The ASM Workbench

A Tool Environment for Computer-Aided Analysis and Validation of Abstract State Machine Models

Dissertation

von

Giuseppe Del Castillo

Schriftliche Arbeit zur Erlangung des Grades
eines Doktors der Naturwissenschaften

Fachbereich Mathematik / Informatik und Heinz Nixdorf Institut
Universität Paderborn

Paderborn, 2000

Gutachter:

1. Prof. Dr. Franz J. Rammig, Universität Paderborn
2. Prof. Dr. Hans Kleine Büning, Universität Paderborn

Acknowledgements

The completion of this thesis was dependent on several people, to whom I wish to express my gratitude here.

First of all, my thanks go to the supervisors Prof. Franz J. Rammig and Prof. Hans Kleine Büning for their efforts in reviewing this work.

Prof. Rammig deserves a special mention for his generous support and continuous encouragement during the last five years. The members of his working group should also be mentioned for their contribution to a nice working atmosphere.

Moreover, I wish to thank Prof. Egon Börger for inciting me to start this work and for his valuable advice, as well as the colleagues with whom I cooperated, in particular Uwe Glässer, Wolfram Hardt and Kirsten Winter, for numerous stimulating discussions and for the fruitful and enjoyable collaboration.

Thanks also to Peter Päppinghaus and Joachim Schmid, who employed the ASM Workbench in the FALKO project and provided very useful feedback for improving the tools.

Contents

Introduction

Gurevich's *Abstract State Machines* (*ASMs*) [45] constitute a simple but powerful state-based formal method for specifying and modelling software and hardware systems. The strength of the ASM method is the combination of *algebras* (*first-order structures*) and *transition systems*. Algebras are used to represent the primitive data and control structures of a system, as well as its instantaneous configurations (*states*), while the system *behaviour* is formalized by means of transition systems, whose states are given by the aforementioned algebras.

Existing case studies, which include specifications of programming language semantics, architectures, protocols, and embedded systems, demonstrate that, in principle, the ASM method is applicable to a wide range of applications (see [13] and [56]). However, like any other formal method, ASMs need appropriate tool support in order to be applied, in practice, to concrete specification and modelling tasks. Tools should support, for instance, simulation (i.e., execution) of ASMs, various kinds of static and dynamic analysis, verification, and generation of documentation and/or code from high-level ASM specifications.

Two main issues arise when trying to equip ASMs with comprehensive tool support.

1. A concrete *specification language* that can be processed by the tools has to be defined. In fact, the definition of Abstract State Machines in [45] includes a language of transition rules for specifying state transitions, but does not include any language constructs for specifying data structures, functions, constraints, and so on. Clearly, an ASM-based specification language must provide expressive means to deal with these features, if it has to be supported by tools.

2. A common *tool infrastructure* has to be provided to ease the task of implementing different ASM tools, as needed for simulation, analysis, verification, and so on. Simple tools, such as ASM interpreters, can be (and have been) developed by means of *ad hoc* solutions, e.g, by extending an existing language and programming environment (such as Prolog [6]). However, a comprehensive tool set, consisting of several interoperable tools providing basic functionalities and designed to be easily extensible by additional components providing special features, requires such an "infrastructure" (or "architecture"). The infrastructure should include reusable modules

implementing internal and external representations of syntactic and semantic entities, basic operations to deal with them, generic transformation schemes, and so on.

This thesis contributes to the solution of the above issues by the definition of an ASM-based specification language, called *ASM-SL*, and of a tool architecture based on this language, the *ASM Workbench*. Another relevant contribution of this thesis is a transformation of ASMs into a form amenable for verification by means of model checking, which has been defined and implemented as a part of the ASM Workbench. The applicability of the specification language and of the related techniques and tools to concrete engineering problems is demonstrated in the last part of the thesis by means of a case study.

According to the structure sketched above, this thesis consists of the following three parts.

1. *Language*. In this part the ASM-based specification language ASM-SL is defined. After a short summary of the basic definitions concerning ASMs (Chapter 1), ASM-SL is informally introduced through a few simple examples (Chapter 2). The following Chapters 3, 4, and 5, contain the formal definition of ASM-SL, consisting of its syntax, type system, and dynamic semantics, respectively.

2. *Tools*. In this part the most relevant features of the ASM Workbench are presented. Chapter 6 deals with the tool environment in general, with particular emphasis on the tool infrastructure mentioned above. Chapter 7 describes in detail the model checking interface.

3. *Applications*. This part deals with an application of the language and tools presented in this thesis to the integration of heterogeneous system models. Chapter 8 describes the general integration approach. Chapter 9 illustrates this approach through a case study from the application domain of automated manufacturing, where heterogeneous models for a distributed material flow system are integrated using ASMs as a semantic "meta-model", such that properties of interest can then be verified with the help of the mentioned model checking interface.

Finally, the thesis is concluded by a summary of the relevant results and experiences as well as by an outlook to possible further developments of the work presented here.

Notational Conventions

Throughout this thesis, standard structures and notations from discrete mathematics and computer science (sets, lists, etc.) are used without further mention. With respect to the notations which are not completely standard, the following conventions are established.

Lists For lists, we basically follow Standard ML. List constructors are "::" (infix, right-associative) and "nil". The form "$[x_1, x_2, \ldots, x_n]$" is equivalent to "$x_1 :: x_2 :: \ldots :: x_n :: \mathrm{nil}$", while "[ ]" is equivalent to "nil". List concatenation, i.e., the "append" function, is usually written in infix notation (right-associative infix operator "@", or as a synonym, "++").

Finite Sets If A is a set, "Fin A" denotes the set of finite subsets of A. Set notation is otherwise standard.

Finite Maps If A and B are sets, "$A \xrightarrow{\text{fin}} B$" denotes the set of *finite maps* (partial functions with finite domain) from A to B. The domain and range of a finite map M are denoted by "dom M" and "ran M", respectively. Finite maps are often enumerated explicitly in the form "$\{x_1 \mapsto y_1, \ldots, x_n \mapsto y_n\}$" ($n \geq 0$); in particular, "{ }" is the empty map (which is also denoted by "$\emptyset$", sometimes). The map comprehension notation $\{\, x \mapsto e_x \mid \phi_x \,\}$ stands for the finite map M with domain $\{\, x \mid \phi_x \,\}$ and whose value on this domain is given by $M(x) = e_x$. If M_1 and M_2 are two finite maps, then:

- "$M_1 \cup M_2$" denotes the *map union* of M_1 and M_2, which is undefined if there is an $x \in \mathrm{dom}\ M_1 \cap \mathrm{dom}\ M_2$ such that $M_1(x) \neq M_2(x)$; otherwise it is the set union of M_1 and M_2.
- "$M_1 \oplus M_2$" denotes the *map override* operation (M_1 "modified by" M_2), which yields a finite map with domain $\mathrm{dom}\ M_1 \cup \mathrm{dom}\ M_2$ and values
$$(M_1 \oplus M_2)(x) = \text{if } x \in \mathrm{dom}\ M_2 \text{ then } M_2(x) \text{ else } M_1(x).$$

If M is a finite map and A is a finite set, then:

- "$M|_A$" denotes M restricted to A:
$$M|_A = \{\, x \mapsto M(x) \mid x \in A \,\}.$$
- "$M \setminus A$" denotes M restricted to $\mathrm{dom}\ M \setminus A$:
$$M \setminus A = M|_{(\mathrm{dom}\ M \setminus A)} = \{\, x \mapsto M(x) \mid x \notin A \,\}.$$

If M_1, M_2 and M_3 are finite maps, the following properties hold:

- $(M_1 \oplus M_2) \oplus M_3 = M_1 \oplus (M_2 \oplus M_3)$.
- If $\mathrm{dom}\ M_1 \cap \mathrm{dom}\ M_2 = \emptyset$: $M_1 \oplus M_2 = M_2 \oplus M_1$.
- If A is a set such that $A \subseteq \mathrm{dom}\ M_2$: $(M_1 \setminus A) \oplus M_2 = M_1 \oplus M_2$.

These properties—which could be easily proved—will be used for the proofs in Appendix D.

Part I

Language

Chapter 1

Abstract State Machines

Abstract State Machines (*ASMs*), formerly known as *Evolving Algebras* (*EAs*), introduced by Yuri Gurevich in [44, 45], constitute a powerful and elegant method for mathematical modelling of discrete dynamic systems: they combine transition systems, which are used for modelling dynamic aspects of a system, i.e., its behaviour, with first-order structures (algebras), used to model static aspects such as data types.

In this chapter we recall the basic notions of Abstract State Machines, as defined in [45, 46, 10]. The presentation given here follows the original definition of [45] with a few exceptions: *(i)* we consider a strongly typed version of ASMs, where states are *multi-sorted* algebras; *(ii)* we introduce the notion of *derived* function, in addition to static, dynamic, and external functions; *(iii)* constructs which involve variables ranging over a given set are defined in the style of [10] instead of [45], i.e., the ranges are specified by finite sets (which are elements of the algebra's base set) instead of using predicates.

This chapter is structured as follows. We first present the ASM computational model (Sect. 1.1). Then we recall the syntax and semantics of the most essential ASM constructs (Sect. 1.2). We conclude by introducing (in Sect. 1.3) the ASM-SL notation, which extends the basic ASM language and is the source language for the ASM Workbench tool environment described in Chapter 6. (The rest of Part I contains a complete description of ASM-SL.)

1.1 Computational Model

1.1.1 Runs

Abstract State Machines define a state-based computational model, where computations (*runs*) are finite or infinite sequences of states $\{S_i\}$, obtained from a given *initial state* S_0 by repeatedly executing *transitions* δ_i:

$$S_0 \xrightarrow{\delta_1} S_1 \xrightarrow{\delta_2} S_2 \ \ldots \xrightarrow{\delta_n} S_n \ \ldots$$

1.1.2 States

The *states* are algebras over a given *signature* Σ (Σ-*algebras* for short). A signature Σ consists of a set of *basic types* (or *sorts*) and a set of *function names*, each function name f coming with a fixed arity n and type $T_1 \dots T_n \to T$, where the T_i and T are basic types (written $f : T_1 \dots T_n \to T$, or simply $f : T$ if $n = 0$). A Σ-algebra (or state) S consists of: *(i)* a nonempty set $\mathcal{T}^S$ for each basic type T (the *carrier set* of T), and *(ii)* a function $\mathbf{f}_S : \mathcal{T}_1^S \times \dots \times \mathcal{T}_n^S \to \mathcal{T}^S$ for each function name $f : T_1 \dots T_n \to T$ in Σ (the *interpretation* of function name f in state S).

Function names in Σ can be declared as *static*, *dynamic*, *external*, or *derived*. The intended meaning of this function classification is as follows.

- **Static functions**. Static function names have the same (fixed) interpretation in each computation state: that is, static functions never change during a run. They typically represent primitive operations of the system, such as operations of an abstract data type (in software specifications) or combinational logic blocks (in hardware specifications).

- **Dynamic functions**. The interpretation of dynamic function names can be altered by the transition occurring in a given computation step (as explained below): that is, dynamic functions change during a run as a result of the specified system's behaviour. They represent the internal state of the system.

- **External functions**. The interpretation of external function names—in each state—is determined by the environment: that is, changes in external functions which take place during a run are not controlled by the system, rather they reflect (uncontrollable) environmental changes.

- **Derived functions**. The interpretation of derived function names—in each state—is function of the interpretation of the dynamic and external function names in the same state: that is, derived functions depend (possibly) on the internal state and on the environmental situation.[1] They represent the view of the system state as accessible to an external observer.

In the literature, synonyms for the function kinds listed above can be found: *controlled* for dynamic, *monitored* for external, and *dependent* for derived. From a methodological point of view, the discussed function classification corresponds to an *open system view*, as depicted in Fig. 1.1.

As a notational convention, we omit explicit mention of the state S when no ambiguity arises: in particular, we write $\mathcal{T}$ instead of $\mathcal{T}^S$ for carrier sets and $\mathbf{f}$ instead of $\mathbf{f}_S$ for static functions, as they never change during a run.

[1] Note that derived functions in ASMs are analogous to the output of a Mealy machine, which is function of the input and of the internal machine state.

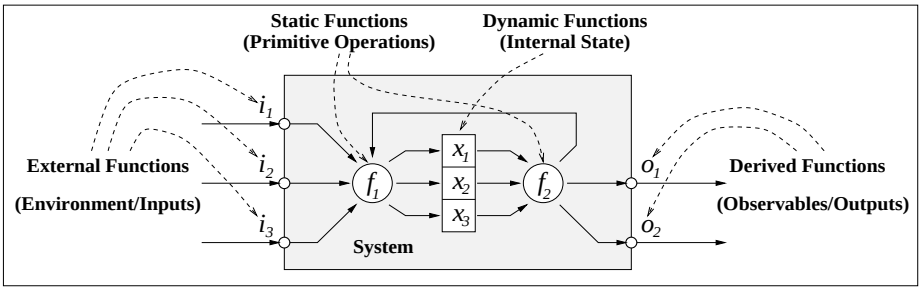


Figure 1.1: Open System View

Standard Types and Functions It is assumed that a few standard types and functions are always defined. In particular, every signature Σ must contain at least a basic type *BOOL* with static nullary function names (constants) *true*, *false* : *BOOL*, the usual boolean operations ($\wedge$, $\vee$, etc.), and the equality symbol $=$. Predicates over $T_1 \dots T_n$ are thus represented as functions of type $T_1 \dots T_n \rightarrow BOOL$ (also called *relations*). In addition, there is a constant *undef* : T for every basic type T except *BOOL*, as well as a type $SET(T)$ of finite sets over T for every basic type T, with the usual set operations ($\cap$, $\cup$, etc.). All standard names listed above have fixed standard interpretations.

1.1.3 Locations

If $f : T_1 \dots T_n \rightarrow T$ is a dynamic or external function name, a pair $l = (f, \overline{x})$ with $\overline{x} \in \mathcal{T}_1 \times \dots \times \mathcal{T}_n$ is called a *location* (then, the *type* of l is T and the *value* of l in a state S is given by $\mathbf{f}_S(\overline{x})$). Note that, within a given run, two states S_i and S_j are equal iff the values of all locations in S_i and S_j are equal (i.e., they coincide iff they coincide on all locations).

1.1.4 Transitions

Transitions transform a state S into its successor state S' by changing the interpretation of some dynamic function names on a finite number of points (i.e., by updating the values of a finite number of *locations*).

More precisely, the transition transforming S into S' results from firing a finite *update set* Δ at S. *Updates* are of the form $((f, \overline{x}), y)$, where $(f, \overline{x})$ is the location to be updated and y the value to be written in that location (f must be a dynamic function name). In the state S' resulting from firing Δ at S the carrier sets are unchanged and, for each function name f:

$$\mathbf{f}_{S'}(\overline{x}) \;=\; \begin{cases} y & \text{if } ((f, \overline{x}), y) \in \Delta \\ \mathbf{f}_S(\overline{x}) & \text{otherwise.} \end{cases}$$

The update set Δ—which depends on the state S—is determined by evaluating

in S a distinguished closed[2] *transition rule* P, called the *program*.[3] Note that the above definition is only applicable if Δ does not contain *conflicting updates*, i.e., any updates $((f, \overline{x}), y)$ and $((f, \overline{x}), y')$ with $y \neq y'$ (i.e., if Δ is *consistent*).

1.2 The Basic ASM Language

1.2.1 Terms

Terms are defined as in first-order logic: *(i)* if $f : T_1 \dots T_n \to T$ is a function name in Σ, and t_i are terms of type T_i (for $i = 1, \dots, n$), then $f(t_1, \dots, t_n)$ is a term of type T (written $t : T$)[4]; *(ii)* a variable v (of a given type T) is a term. The meaning of a term $t : T$ in a state S and environment[5] ρ is a value $S_\rho(t) \in \mathcal{T}$ defined by:

$$S_\rho(t) \;=\; \begin{cases} \mathbf{f}_S(S_\rho(t_1), \dots, S_\rho(t_n)) & \text{if } t \equiv f(t_1, \dots, t_n) \\ \rho(v) & \text{if } t \equiv v. \end{cases}$$

(For closed terms, we omit explicit mention of ρ: if t is closed, $S(t) = S_\emptyset(t)$.)

As opposed to first-order logic, there is no special notion of formula: boolean terms are used instead. Finite quantifications of the form "$(Q\ v \in A : G)$" (where Q is $\forall$ or $\exists$, $v : T$, and $A : \mathit{SET}(T)$) are also valid boolean terms.

1.2.2 Transition Rules

While terms denote values, transition rules (*rules* for short) denote *update sets*, and are used to define the dynamic behaviour of an ASM: the meaning of a rule R in a state S and environment ρ is an update set $\Delta_{S,\rho}(R)$. (As in the case of terms, if R is closed we omit explicit mention of ρ by defining $\Delta_S(R) = \Delta_{S,\emptyset}(R)$.)

ASM runs starting in a given initial state S_0 are determined by a closed transition rule P declared to be the *program*: each state S_{i+1} ($i \geq 0$) is obtained by firing the update set $\Delta_{S_i}(P)$ at S_i. Visually:

$$S_0 \overset{\Delta_{S_0}(P)}{\longrightarrow} S_1 \overset{\Delta_{S_1}(P)}{\longrightarrow} S_2 \;\dots\; \overset{\Delta_{S_{n-1}}(P)}{\longrightarrow} S_n \;\dots$$

Basic transition rules are the *skip*, *update*, *block*, and *conditional* rules. Additional rules are the *do-forall* (a generalized block rule) and *choose* rules (for non-deterministic choice).[6]

$$\begin{aligned} R ::= \;& \texttt{skip} \mid f(t_1, \dots, t_n) := t \mid R_1 \dots R_n \mid \texttt{if } G \texttt{ then } R_T \texttt{ else } R_F \\ & \mid \texttt{do forall } v \texttt{ in } A \texttt{ with } G \;\; R' \mid \texttt{choose } v \texttt{ in } A \texttt{ with } G \;\; R' \end{aligned}$$

[2] A closed transition rule, like a closed term, is a transition rule without free variables.

[3] In applications of ASMs, the program consists usually of a set (block) of rules, describing system behaviour under different—usually mutually exclusive—conditions.

[4] If $n = 0$ the parentheses are omitted, i.e., we write f instead of $f()$. Moreover, we use the letter G (*guard*) to refer to boolean terms and A for set terms ($A : \mathit{SET}(T)$ for some type T).

[5] An environment is a finite map containing bindings which associate variables to their corresponding values.

[6] The ASM-SL notation, which is the subject of the next few chapters, include more transition rules, such as *let* and *case* rules with pattern matching.

The form "`if` G `then` R" is a shortcut for "`if` G `then` R `else skip`". Omitting "`with` G" in *do-forall* and *choose* rules corresponds to specifying "`with true`".

The semantics of transition rules is as follows:

$$\begin{aligned}
\Delta_{S,\rho}(\,\texttt{skip}\,) &= \{\,\} \\
\Delta_{S,\rho}(\,f(t_1,\ldots,t_n) := t\,) &= \{\,((f,(S_\rho(t_1),\ldots,S_\rho(t_n))),S_\rho(t))\,\} \\
\Delta_{S,\rho}(\,R_1\ \ldots\ R_n\,) &= \textstyle\bigcup_{i=1}^{n} \Delta_{S,\rho}(\,R_i\,) \\
\Delta_{S,\rho}(\,\texttt{if}\ G\ \texttt{then}\ R_T\ \texttt{else}\ R_F\,) &= \begin{cases} \Delta_{S,\rho}(\,R_T\,) & \text{if } S_\rho(G) = \mathbf{true} \\ \Delta_{S,\rho}(\,R_F\,) & \text{otherwise} \end{cases}
\end{aligned}$$

$$\begin{aligned}
\Delta_{S,\rho}(\,\texttt{do forall}\ v\ \texttt{in}\ A\ \texttt{with}\ G\ \ R'\,) \;=\; & \textstyle\bigcup_{x\in X} \Delta_{S,\rho\oplus\{v\to x\}}(\,R'\,) \\
& \text{where } X = \{x \mid x \in S_\rho(A) \wedge S_{\rho\oplus\{v\to x\}}(G) = \mathbf{true}\}.
\end{aligned}$$

Note that executing a block (or a do-forall) rule corresponds to *simultaneous* execution of its subrules[7] and may lead to *conflicts* (inconsistent update sets). Dealing with the semantics of `choose` is more involved, thus we delay it to Chapter 5, which defines the dynamic semantics of the full ASM-SL language (of which the basic ASM language presented in this section is a subset).

1.2.3 Multi-Agent ASMs

Concurrent systems can be modelled in ASMs by the notion of multi-agent ASM (called *distributed ASM* in [45]). The basic idea is that the system consists of more *agents*, identified with the elements of a finite set $AGENT$ (which are actually sort of "agent identifiers"). Each agent $a \in AGENT$ executes its own program $prog(a)$ and can identify itself by means of a special nullary function $self : AGENT$, which is interpreted by each agent a as a.

In [45] several semantic models for multi-agent ASMs are discussed, the most general being *partially ordered runs*. For our purposes, a simple interleaving model (*sequential runs* in the terminology of [45]) is sufficient. This allows us to model concurrent systems in the basic ASM formalism as described above.

In particular, we consider *self* as an external function, whose interpretation $\mathbf{self}_{S_i}$ determines the agent which fires at state S_i. We assume that there is one program P, shared by all agents, possibly performing different actions for different agents, e.g.,

`if` $self = a_1$ `then` $prog(a_1)$
...
`if` $self = a_n$ `then` $prog(a_n)$

where $\{a_1,\ldots,a_n\}$ are the agents and $prog(a_i)$ is the rule to be executed by agent a_i, i.e., the "program" of a_i.[8]

[7] For example, a block rule `a` := `b`, `b` := `a` exchanges `a` and `b`.

[8] All examples of concurrent systems presented in this thesis are written in this style.

1.3 The ASM-SL Notation

In order to equip the ASM method with reasonable tool support, the basic ASM language needs to be extended with additional features, addressing pragmatical issues which arise when dealing with concrete modelling tasks. Therefore, as a source language for the ASM Workbench tool environment (described in Chapter 6), we defined the ASM-SL notation.[9] The additional features of ASM-SL include:

1. a simple but flexible *type system*;
2. constructs to define the types and functions making up ASM states (*data model*);
3. mechanisms to define *interfaces* to the environment, according to the open system view.

The language extensions have been realized by borrowing established concepts from other languages and methods, while trying to keep the language *concise*, *understandable*, and *executable*. In this section, we briefly describe the ASM-SL features and motivate the underlying design decisions (to a detailed language description are dedicated the Chapters 2 to 5).

1.3.1 Type System

ASMs, as defined in [45], are untyped, but there are some good reasons for introducing a type system. Types provide a considerable help in clarifying and exposing the structure of the data model. Moreover, a type-checker may detect trivial errors and inconsistencies very early, before undertaking any simulation or verification.[10]

The type system of ASM-SL is an adaptation of the well-known polymorphic type system of Standard ML [70, 27, 72], which is a reasonable compromise between simplicity and flexibility. Besides the advantage of being very well-known, the possibility of performing type inference allows for concise specifications, as most type annotations can be omitted.[11]

1.3.2 Data Model Specification

The issue of how to specify the universes and functions making up ASM states (*data model* for short) is not considered in [45], which only defines the language for specifying transitions. In fact, there are very good reasons for not prescribing a particular specification language for the data model. The possibility of using

[9] ASM-SL stands for ASM-based Specification Language.

[10] Of course, there are also good arguments in favour of keeping the specification language untyped. Anyway, the type system is orthogonal to the other ASM-SL extensions, which in principle could be interpreted as untyped as well.

[11] Extensions of the ML type system, e.g., by the introduction of *type classes* [84], would not present any additional difficulties.

any existing approach (e.g., algebraic, model-based, or even *ad hoc* notation) for defining the states accounts for much of the flexibility of ASM method: thanks to the use of algebras as states, the data model can always be easily combined with a behavioural model of the system consisting of transition rules.

For the purpose of tool support, however, a fixed notation is clearly needed, unless the tool relies on an external language (often coinciding with the implementation language, which is unsatisfactory for several obvious reasons). Such a specification notation is necessarily the result of a compromise between different design goals. The solution adopted in ASM-SL consists in providing:

- a set of predefined elementary types (booleans, integers, floating-point numbers, strings) and polymorphic types (tuples, lists, finite sets, finite maps) together with a construct for defining arbitrary *freely generated types*, as means of defining the structure of the base set;
- a set of additional language constructs borrowed from functional programming and model-based specification (especially Standard ML [72] and VDM [60]), such as recursive and mutually recursive function definitions, pattern matching, set-theoretic notation, as means of defining functions.

This approach, though less abstract than others (e.g., algebraic specifications), allows nevertheless to model a wide range of applications[12], is quite intuitive, and has the advantage that *executable models* are obtained by constructions. The use of familiar structures and notations from discrete mathematics and computer science ensures that ASM-SL specifications can be easily understood and translated into other languages (programming languages or logics).[13]

1.3.3 Interfaces to the Environment

As ASMs are often used to model embedded and reactive systems, the issue of *interfaces* between system and environment is a crucial one. As means for defining such interfaces, ASM-SL supports the standard ASM mechanism of *external functions*. How external functions are concretely handled depends on the tools: the ASM Workbench's simulator, for instance, considers external functions as *oracles* and communicates with a so-called *oracle process* (an external process to be provided by the user, such a simulation of the environment) in order to fetch the values of the external functions.

Declarations of external functions may come with *constraints*, defining restrictions on their interpretation. Supported are currently *type constraints* and *finiteness constraints* (introduced in Sect. 7.2). The meaning of type constraints is obvious. Finiteness constraints define, for each point of a function's domain,

[12] Of course, other approaches based on specialized (application domain-specific) data models may be more convenient for particular applications. A significant example is given by the *Montages* technique for specifying programming languages and the related tool Gem-Mex [3], which are based on a specialized version of ASMs, where the underlying data model consists essentially of control-/data-flow graphs.

[13] Most theorem provers come with predefined theories and most modern programming languages with constructs or generic class libraries to deal with the mentioned structures.

a finite set of values which the function is allowed to assume on that point. In this way, while it is still possible to see external functions as oracles (for the purpose of simulation), the application of an external function can also be considered as a point of non-deterministic choice among a finite set of alternative values (which is convenient for the purpose of analysis, e.g., by model-checking, see Chapter 7).[14]

1.4 Application Domains

Abstract State Machines have been used successfully for the specification and modelling of various kinds of systems including computer architectures, programming languages, control systems, and protocols. Comprehensive overviews of ASM-related work, which demonstrate the wide range of applications which can be handled by ASMs, can be found in [13, 18].

By examining this literature, it can be observed that applications of ASMs fall naturally into two main classes, which can be characterized as follows.

System-level modelling. This class includes applications where ASMs are used as a specification or modelling language (typically in a very early design phase) in view of the development of concrete software or hardware systems serving a specific purpose. In this scenario, ASMs are typically used to formulate an high-level behavioural model of a concrete design (possibly an existing design, in case of reverse engineering or reengineering applications). This model can be used both as a system description for validation purposes, e.g., to show that the high-level design satisfies given requirements, and as an abstract design specification for the further system development by stepwise refinement, down to a concrete implementation.

Typical representatives of this class of applications are the case studies *Steam Boiler Control* [7] and *Production Cell* [19], where ASMs have been used as specification language for the development of embedded control systems.[15] A reverse engineering application is instead presented in [15], where a high-level ASM model of an existing processor architecture was extracted from a simulator (constituting the most abstract specification available of that architecture) in view of the design of a new version of the processor.

Language-level modelling ("**meta-modelling**"). This class includes, besides several works in the traditional field of programming language semantics (C [47], C++ [85], and Java [21, 20], just to mention a few ones), definitions of the operational semantics of specification and modelling languages which are of particular interest in the field of embedded systems design (like the hardware

[14] Note that, for the purpose of simulation, constraints (of both kinds) can be used to perform run-time checks on the admissibility of the values provided by the oracles.

[15] Both case studies were proposed as reference applications to compare different formal methods. In the case study *Steam Boiler Control* [1], control software to control the level of water in steam-boiler (keeping it within admissible limits) was requested. In the case study *Production Cell* [63], the purpose was to develop automation software to control a typical industrial production cell used in a metal processing factory.

description language VHDL [57][16, 17] and the Specification and Description Language SDL [58][39, 11], which is mainly used for specifying protocols).

In this scenario, ASMs are used as a meta-language to define an "abstract interpreter" or "abstract machine" executing programs or simulating system descriptions expressed in the object language (where, typically, it is assumed that these programs or description are well-formed, as the ASM specification does not deal with static semantics, which must be specified by other means, such as attributed grammars or typing rules). When using ASMs for meta-modelling, the behaviour of concrete systems is not expressed directly by ASM programs. Instead, the given system is described in the object language, and this description is encoded into appropriate data structure in the ASM state, mapped into ASM transition rules, or a combination of both.

Although the case studies mentioned above—among others—demonstrate that it is possible to apply ASMs for system-level modelling, this does not seem the most promising branch. In fact, application domain-specific languages (such as the mentioned SDL and VHDL, Statecharts [50], or synchronous languages [49]), where applicable, provide considerable advantages compared to general-purpose languages, for instance, an intuitive graphical notation (Statecharts, SDL), efficient code generation and verification techniques (synchronous languages), hardware synthesis (VHDL). Furthermore, besides their objective merits, languages standardized by official bodies (like VHDL and SDL) are widely accepted in industry, while more "experimental" formal methods have hope to gain acceptance only if they bring considerable and quantifiable benefits.

On the other hand, ASMs seem to be very appropriate for metamodelling, thanks to the combination of algebras for data type specifications and transition systems for specification of behaviour. In fact, more restricted formalisms—based, for instance, on extended finite state machines of some kind—are in general not expressive enough to represent complex program structures or system descriptions, while other methods, like denotational semantics, have difficulties to deal with systems and languages which are heavily state-based. This appropriateness seems also to be confirmed by the fact that ASMs are being used in a formal semantics definition of SDL-2000 determined to be approved as official formal semantics by the ITU-T ([59], see also [11]).

In this thesis, Abstract State Machines will be considered as a meta-language, not as a system design language.[16] In particular, in the next chapter, we present some simple examples of meta-modelling in order to informally introduce the ASM-SL notation. In Part III (Applications), it will be shown, through a case study, how ASMs and ASM-SL can be used as a meta-model for the integration of heterogeneous embedded system models.

[16] An exception is the dining philosophers example of Chapter 7, which is however only intended to demonstrate the transformation techniques documented in that chapter, not as a real application.

Chapter 2

Some Introductory Examples

In this chapter we present a few examples in order to give an informal introduction to the ASM-SL notation. The examples chosen are quite heterogeneous, as we intend to show the adequacy of the notation in different application domains. However, all of them are examples of language-level modelling (meta-modelling) and, as a side effect, illustrate the corresponding modelling methodology.

First we consider two models of sequential systems, at different levels of abstraction, namely: *(i)* the operational semantics of a simple imperative language (while-language, Sect. 2.1), and *(ii)* a model of the instruction set of a VLIW processor (Sect. 2.2). Then we show how concurrent systems can be modelled by the example of place/transition nets, a simple kind of Petri nets (Sect. 2.3).

2.1 The while-Language

We consider here a simple imperative language (while-language) consisting of expressions and statements. Expressions are free of side effects, such that they can be modelled in a purely statical (i.e., functional) way; the execution of a statement is reflected by an ASM computation step, such that the semantics of a while-program is defined by the corresponding runs. For our purposes, we focus on dynamic semantics and do not specify well-formedness conditions for while-programs (which we assume well-formed and type-correct).

Abstract Syntax The abstract syntax of *expressions* and *statements* is represented by two *freely generated types* `EXPR` and `STMT` (where the type `ID` of identifiers is identified with `STRING` by means of a *type alias*):

```
typealias ID == STRING

freetype EXPR ==
{ Con : INT,
  Var : ID,
  App : ID * [EXPR],
  Let : ID * EXPR * EXPR }
```

```
freetype STMT ==
{ Seq    : [STMT],
  If     : EXPR * STMT,
  While  : EXPR * STMT,
  Assign : ID * EXPR,
  Input  : ID,
  Output : EXPR }
```

Expressions are integer constants, variables, function applications, or `let`-expressions. Statements are sequential compositions, `if`, `while`, assignments, or input/output statements. (The notation `[`T`]` is a shorthand for `LIST(`T`)`).

Semantics of Expressions Possible values of expressions are booleans and integers. This is reflected by a free type with two constructors:

```
freetype VALUE == { Bool : BOOL, Int  : INT }
```

The semantics of built-in functions of the while-language is defined by a *static function* `interpretation : ID * [VALUE] -> VALUE` (whose definition illustrates the use of *pattern matching* over free types by means of the `case` construct[1]):

```
static function interpretation (f, args) ==
  case args of
    [] : case f of
      "true"  : Bool (true) ;
      "false" : Bool (false)
    endcase ;
    [ Bool (x) ] : case f of
      "not" : Bool (not (x))
    endcase ;
    // ...
    [ Int (x), Int (y) ] : case f of
      "+"   : Int (x + y) ;
      // ...
      "="   : Bool (x = y) ;
      // ...
    endcase ;
    [ Bool (x), Bool (y) ] : case f of
      "and" : Bool (x and y) ;
      "or"  : Bool (x or y)
    endcase
  endcase
```

(Note that, in function definitions such as the one above, the function type does not need to be declared explicitly, as it can be inferred by the type-checker.)

The semantics of expressions is defined by another static function, whose arguments are the expression to be evaluated and the *environment*[2] (a finite

[1] The ASM-SL `case` construct is similar to analogous constructs found in functional programming languages like Standard ML [72] or Haskell [54] (see Chapters 4 and 5 for details).

[2] The term "environment", here, comes from the semantics jargon and has obviously nothing to do with the environment as understood in the context of the open system view.

map containing bindings which associate variables to their corresponding values) in which the expression is to be evaluated:

```
static function eval_in_env (E, env) ==
  case E of
    Con (x) : Int (x) ;
    Var (v) : apply (env, v) ;
    App (f, E_list) :
      interpretation (f, [ eval_in_env (E, env) | E in E_list ]);
    Let (x, E1, E2) :
      let E1_value == eval_in_env (E1, env)
      in eval_in_env (E2, override (env, { x -> E1_value }))
      endlet
  endcase
```

(In the above definition two primitive map operations of ASM-SL are used, namely `apply` : `MAP`(T_1, T_2) `*`T_1 `->`T_2, the *functional application* of a finite map to a given argument, and `override` : `MAP`(T_1, T_2) `*` `MAP`(T_1, T_2) `->` `MAP`(T_1, T_2), the *map override* operation. Note also, in the `App` case, the use of *list comprehension* notation.)

Machine State The instantaneous configuration of the machine (including both control state and data) is represented by *dynamic functions*. The control flow is modelled—using the technique of continuations—by three nullary dynamic functions: `curr_stmt` and `curr_cont` (*current statement* and *current continuation*, respectively), and `terminated`, which becomes true when the program terminates. The memory contents are represented by a dynamic function `global_env` associating global variables to the corresponding values, while the output stream is represented by a sequence `output` of output values.

The program to be executed, which the machine's input in the initial state, is represented by an *external function* `program : STMT` (note that `program` is used only once, namely in the initialization of `curr_stmt`). The data read from the input stream when executing input statements is given by another external function `input : VALUE`, to be understood as an *oracle*, consulted whenever the need arises (see, in particular, the rule for the `Input` statement in Table 2.1).

```
external function program   :STMT

dynamic function curr_stmt  :STMT    initially program
dynamic function curr_cont  :[STMT]  initially []
dynamic function terminated :BOOL    initially false

dynamic function global_env :ID -> VALUE
initially MAP_TO_FUN emptymap
```

(The `MAP_TO_FUN` operator is a special ASM-SL construct for the extensional definition of functions, which "converts" a finite map into the corresponding function, see Sect. 5.2.4.)

Finally, a *derived function* `eval_expr` is introduced as a shorthand for the evaluation of expressions within the global environment:

```
derived function eval (t) ==
  eval_in_env (t, FUN_TO_MAP global_env)
```

(The operator FUN_TO_MAP is symmetrical to MAP_TO_FUN in that it converts an extensionally defined function into a finite map.)

Semantics of Statements The semantics of statements is given by the *named transition rules*[3] of Table 2.1 (one for each kind of statement), where the auxiliary transition rule Continue lets the execution advance to the next statement whenever the current statement does not alter the control flow (to achieve this, Continue exploits the information contained in the current continuation):

```
transition Continue ==
  case curr_cont of
    stmt1 :: stmts : curr_stmt := stmt1
                     curr_cont := stmts ;
                [] : terminated := true
  endcase
```

The operational semantics of the while-language is defined by the following rule (the ASM *program*), which performs a case distinction on the current statement and executes the corresponding rule:

```
transition ExecuteStmt ==
  if not (terminated)
  then case curr_stmt of
         Seq (stmt_list) : ExecuteSeq (stmt_list) ;
         While (E, stmt) : ExecuteWhile (E, stmt) ;
         If (E, stmt)    : ExecuteIf (E, stmt) ;
         Assign (v, E)   : ExecuteAssign (v, E) ;
         Input (v)       : ExecuteInput (v) ;
         Output (E)      : ExecuteOutput (E)
       endcase
  endif
```

Note that each ASM step, as defined by the above ASM program, simulates the execution of a statement of the while-program, until terminated becomes true. Thus, a run of the ASM presented here corresponds to a complete run of the simulated while-program.

Encoding of while-Programs Finally note that, in order to be executed by the abstract machine (interpreter) presented here, while-programs have to be encoded into the ASM state. More precisely, the external function program—which determines the interpretation of curr_stmt in the initial state—must provide a value of type STMT, corresponding to the abstract syntax tree (AST) of the while-program to be executed. An example of program encoding is presented in Table 2.2: the left part contains a simple while-program (in concrete while

[3]The *named transition rules* of ASM-SL are often called *macros* in the ASM literature.

```
transition ExecuteSeq (stmt_list) ==
  case stmt_list of
    stmt1 :: stmts :
      curr_stmt := stmt1
      curr_cont := stmts @ curr_cont ;
    [] : Continue
  endcase
```

```
transition ExecuteWhile (E, stmt) ==
  if eval (E) = Bool (true)
  then curr_stmt := stmt
       curr_cont := [ While (E, stmt) ]
                    @ curr_cont
  else Continue
  endif
```

```
transition ExecuteIf (E, stmt) ==
  if eval (E) = Bool (true)
  then curr_stmt := stmt
  else Continue
  endif
```

```
transition ExecuteAssign (v, E) ==
block
  global_env (v) := eval (E)
  Continue
endblock
```

```
transition ExecuteInput (v) ==
block
  global_env (v) := input
  Continue
endblock
```

```
transition ExecuteOutput (E) ==
block
  output := output @ [ eval (E) ]
  Continue
endblock
```

Table 2.1: while-Language: Semantics of Statements

```
{
  input max;
  x := 1;
  while (x <= max)
  {
    if (x mod 2 = 0)
    then output x;
    x := x + 1;
  }
}
```

```
Seq ([
  Input ("max"),
  Assign ("x", Con(1)),
  While ( App ("<=", [ Var("x"), Var("max") ]),
    Seq ([
      If ( App ("=", [ App ("mod",[Var("x"),Con(2)]), Con(0) ]),
           Output (Var("x")) ),
      Assign ("x", App ("+", [ Var("x"), Con(1) ]))
    ]) )
])
```

Table 2.2: while-Language: Encoding of Programs (AST Representation)

syntax), the right part shows the corresponding AST (which is a well-formed and well-typed ASM-SL term of type `STMT`).[4]

2.2 An Instruction Set Model

As an example of ASM-based operational semantics at a lower level of abstraction, we present a model of the instruction set of a VLIW processor. This instruction set is used as a kind of "virtual" machine code for the zCPU, a VLIW processor employed as control unit in the SIMD special-purpose parallel architecture APE100 developed at INFN[5] [4, 5], for which an ASM model at the RT-level also exists [15]. The model presented here first appeared in [32], where the goal was to demonstrate how an ASM model can ease the analysis of dynamic instruction set properties.

[4] Typically, the encoding—together with the necessary well-formedness checks—is accomplished by a parser, realized with standard compiler-construction tools and techniques.

[5] Istituto Nazionale di Fisica Nucleare, the Italian National Institute for Nuclear Physics.

The Instruction Format The instruction set under study—as most RISC instruction sets—consists essentially of register-register arithmetic-logical instructions, load/store instructions for memory access, and branch instructions. We distinguish here two classes of instructions, arithmetic-logical instructions (MAC for short) and all other instructions (IOC for short).[6]

The instruction format is represented by a free type `INSTR`. Auxiliary types `JUMP_COND`, `REG` and `DISP` represent branch conditions[7], register addresses and the displacement field in IOC instructions, respectively:

```
freetype JUMP_COND == { TRUE, FALSE, EQ, NE, LT, LE, GT, GE }
freetype REG       == { R : INT }
freetype DISP      == { Disp : INT }
freetype INSTR ==
{ // arithmetic-logic instructions (MAC)
  ZERO : REG,  FF : REG,  CMP : REG * REG,
  OR  : REG * REG * REG,  AND : REG * REG * REG,
  // ... all other MAC instructions have the same 3-opnd format
  // load/store, branch and special instructions (IOC)
  LD   : REG * REG * DISP,
  ST   : REG * REG * DISP,
  // ...
  JUMP : REG * DISP * JUMP_COND,
  HALT }
```

Machine State The machine state consists of the program memory[8] (`instr`), the program counter (`PMA`: Program Memory Address), the data memory (`mem`), the general-purpose registers (`reg`) and the flags (`Neg`, `Zero`, `Divz`), and is represented by dynamic functions (with the exception of `instr`, which is read-only):

```
external function program : MAP (INT, INSTR)
external function initmem : MAP (INT, INT)

static function instr : INT -> INSTR == MAP_TO_FUN program
dynamic function PMA  : INT          initially 0
dynamic function mem  : INT -> INT   initially MAP_TO_FUN initmem
dynamic function reg  : REG -> INT   initially MAP_TO_FUN emptymap

dynamic function Neg   initially false
dynamic function Zero  initially false
dynamic function Divz  initially false
```

Note that external functions `program` and `initmem` are used to parameterize the initial state (by means of the same technique as in the while-language model).

[6] This distinction was originally motivated by the VLIW architecture of the processor, which provides two independent functional units for executing MAC and IOC instructions.

[7] Note that we abstract from the concrete binary representation of such branch conditions by using symbolic names (constructors).

[8] In the architecture under study, two separate memories are used for the program and for the data, such that instructions and data can be accessed simultaneously. The program memory is read-only, in the sense that it can be written only through a special loading mechanism, not as a result of normal instruction execution.

MAC Instructions The execution of arithmetic-logical instructions is modelled by a rule `MAC_RULE`, which performs a case distinction and executes the appropriate subrule:

```
transition MAC_RULE == case instr (PMA) of
  AND (RR, R1, R2)   : DO_AND (RR, R1, R2) ;
  OR (RR, R1, R2)    : DO_OR (RR, R1, R2) ;
  // ... the same for other MAC instructions
end
```

The semantics of each single instruction is modelled by a corresponding rule, for instance:[9]

```
transition DO_OR (RR, R1, R2) ==
  LogicalInstr (RR, or_fun (reg (R1), reg (R2)))

transition DO_AND (RR, R1, R2) ==
  LogicalInstr (RR, and_fun (reg (R1), reg (R2)))
```

Note that the application of (static) functions `or_fun` and `and_fun`, respectively, is the only difference between the two rules. In fact, there is one such static function for each available arithmetic operation. In this way, static aspects are separated from behavioural aspects, which are described—in the case of logical operations like `AND`, `OR`—by the common rule:

```
transition LogicalInstr (RR, value) == block
  reg (RR) := value
  Neg      := (value < 0)
  Zero     := (value = 0)
end
```

IOC Instructions As in the case of MAC instructions, the main rule is a case distinction:

```
transition IOC_RULE == case instr (PMA) of
  LD (RD, RA, disp)      : DO_LD (RD, RA, disp) ;
  LDA (RD, RA, disp)     : DO_LDA (RD, RA, disp) ;
  ST (RD, RA, disp)      : DO_ST (RD, RA, disp) ;
  // ...
  JUMP (RA, disp, cond) : DO_JUMP (RA, disp, cond) ;
end
```

The rules for load/store instructions are straightforward:

```
transition DO_LD (RD, RA, Disp (d)) ==
  reg (RD) := mem (reg (RA) + d)

transition DO_LDA (RD, RA, Disp (d)) ==
  reg (RD) := reg (RA) + d
```

[9] We show only two examples here, rules for other instructions are similar.

```
transition DO_ST (RD, RA, Disp (d)) ==
  mem (reg(RA) + d) := reg (RD)
```

The rule for branch instructions is slightly more complicated, as it implies testing the branch condition against the flags, which is done with the help of a static function `eval_cond`:[10]

```
static function eval_cond (cond, N, Z) == case cond of
  TRUE  : true ;     FALSE : false ;
  EQ    : Z;         NE    : not (Z) ;
  LE    : N or Z;    // ...
end

transition DO_JUMP (RA, Disp (d), cond) ==
  if eval_cond (cond, Neg, Zero)
  then PMA := reg (RA) + d
  else PMA := PMA + 1 end
```

The PMA Rule Finally, we need a rule to increment the program counter in normal situations, i.e., whenever the current instruction is not `JUMP` or `HALT`:

```
transition INCR_PMA ==
  if not (is_jump_instruction (instr (PMA)))
     and (instr (PMA) != HALT)
  then PMA := PMA + 1 end
```

(The static function `is_jump_instruction` yields true iff its argument is a `JUMP` instruction.)

The Instruction Set Model The overall instruction set model is simply obtained by the synchronous-parallel composition of the above rules into the following ASM program:

```
transition ZCPU ==
  block  MAC_RULE  IOC_RULE  INCR_PMA  end
```

Encoding of Programs Like in the case of the while-language, zCPU machine programs have to be encoded into the ASM state before they can be executed by the ASM presented here. Table 2.3 shows an example of such encoding (the code in the table is a fragment of a simple matrix multiplication program, consisting of three nested loops). Again, the left part of the table contains the program in concrete syntax, while the right part is a well-formed and well-typed ASM-SL term (of type `MAP (INT, INSTR)`), denoting an admissible value for the external function `program`.

[10] The static function `eval_cond` is actually implemented by a combinational logic block in the real processor.

```
            ZERO R0                          { 0 -> ZERO (R(0)),
            LDA R1 R0 #1                       1 -> LDA (R(1), R(0), Disp(1)),

            LDA R8 R0 #2                       2 -> LDA (R(8), R(0), Disp(2)),
            LDA R9 R0 #2                       3 -> LDA (R(9), R(0), Disp(2)),
            LDA R10 R0 #2                      4 -> LDA (R(10), R(0), Disp(2)),

            LDA R4 R0 #1                       5 -> LDA (R(4), R(0), Disp(1)),
for_i_entry: CMP R4 R8                         6 -> CMP (R(4), R(8)),
            JUMP R0 :for_i_exit GT             7 -> JUMP (R(0), Disp(51), GT),

            LDA R5 R0 #1                       8 -> LDA (R(5), R(0), Disp(1)),
for_j_entry: CMP R5 R10                        9 -> CMP (R(5), R(10)),
            JUMP R0 :for_j_exit GT             10 -> JUMP (R(0), Disp(49), GT),

            ...                                ...

            LDA R6 R0 #1                       18 -> LDA (R(6), R(0), Disp(1)),
for_k_entry: CMP R6 R9                         19 -> CMP (R(6), R(9)),
            JUMP R0 :for_k_exit GT             20 -> JUMP (R(0), Disp(47), GT),

            ...                                ...

            ADD R6 R6 R1                       45 -> ADD (R(6), R(6), R(1)),
            JUMP R0 :for_k_entry TRUE          46 -> JUMP (R(0), Disp(19), TRUE),

for_k_exit:  ADD R5 R5 R1                      47 -> ADD (R(5), R(5), R(1)),
            JUMP R0 :for_j_entry TRUE          48 -> JUMP (R(0), Disp(9), TRUE),

for_j_exit:  ADD R4 R4 R1                      49 -> ADD (R(4), R(4), R(1)),
            JUMP R0 :for_i_entry TRUE          50 -> JUMP (R(0), Disp(6), TRUE),

for_i_exit:  HALT                              51 -> HALT }
```

Table 2.3: Instruction Set Model: Encoding of Programs

Instrumentation and Analysis The instruction set model presented here can be very easily "instrumented" in order to collect interesting information about dynamic properties of the instruction set (e.g., relative instruction usage) during simulation.[11] In fact, one simply needs to extend the instruction set model by the necessary functions and transition rules to do the bookkeeping, which can be freely mixed with those constituting the actual model.

For instance, assume that we want to analyse a program's branch behaviour, in order to gain some insight about performance issues related to pipelining. Some measures which are useful in this respect are:

- the proportion of taken branches vs. not taken ones;
- the average number of instructions executed between two taken branches, including not taken branches (*average branch distance*).[12]

[11] Such a simulation-based quantitative analysis of instruction sets is essential, for instance, for an ASIC-designer, in order to determine an efficient instruction mix, to identify which instructions deserve more optimization effort, to properly dimension pipelines, and so on (see, for instance, the discussion in [52], Chapters 3 and 4).

[12] Less taken branches and a longer average branch distance imply that a better performance can be achieved by longer pipelines. This is an example of how the analysis data obtained by performing a sufficient number of simulations (with programs and data representative of the application domain) can support the designer in making sensible design decisions.

```
dynamic function TakenBranches     initially 0
dynamic function NotTakenBranches  initially 0

dynamic function StepsFromLastTakenBranch : INT  initially 0
dynamic function BranchDistanceList : LIST(INT)  initially []

transition BRANCH_BEHAVIOUR_ANALYSIS ==
  case instr (PMA) of
    JUMP (RA, disp, cond) :
      if eval_cond (cond, Neg, Zero)
      then TakenBranches := TakenBranches + 1
           BranchDistanceList := BranchDistanceList
                                 @ [ StepsFromLastTakenBranch ]
           StepsFromLastTakenBranch := 0
      else NotTakenBranches := NotTakenBranches + 1
           StepsFromLastTakenBranch := StepsFromLastTakenBranch + 1
      end
    otherwise :
      StepsFromLastTakenBranch := StepsFromLastTakenBranch + 1
  end
```

Table 2.4: Instruction Set Model: Instrumentation Code

In order to collect the required analysis data, one only has to add some "bookkeeping" functions and transition rules to the instruction set model, in this case: *(i)* dynamic functions `TakenBranches` and `NotTakenBranches` counting the number of taken and not taken branches, respectively; *(ii)* dynamic function `StepsFromLastTakenBranch` and `BranchDistanceList` to keep track of branch distances; and *(iii)* a transition rule `BRANCH_BEHAVIOUR_ANALYSIS`, which actually collects the data.

The bookkeeping is activated by simply including the transition rule collecting the data in the ASM program:

```
transition ZCPU == block
  MAC_RULE  IOC_RULE  INCR_PMA    // instruction set model
  BRANCH_BEHAVIOUR_ANALYSIS       // instrumentation code
end
```

In this way, the analysis data is updated at each step, simultaneously with the execution of one instruction. (Note that the instrumentation code is added in a very modular way, without changing anything in the actual system model, thanks to the ASM semantics underlying the simulation).

At the end of the simulation, all the needed information will be available, including a "branch distance list", providing comprehensive information about the length of all instruction sequences executed without branching. The average branch distance, as well as other interesting measures, can then be derived from this list (some experimental results are presented in [32]).

2.3 Place/Transition Nets

In order to exemplify the (meta-)modelling of concurrent systems in ASM-SL, we consider place/transition nets (P/T nets for short), a simple kind of Petri nets. Although Petri net formalisms are, in several respects, quite different from traditional computer languages (such as high-level imperative languages or machine languages like those discussed above), the general modelling methodology still presents some similarities.

On the one hand, one has to identify an appropriate encoding scheme for concrete system descriptions expressed in the object formalism (e.g., syntax trees for programming languages, graph structures for Petri nets). On the other hand, one has to define a general model of state and behaviour, corresponding to a "universal" abstract machine for the object formalism, which is able to simulate any (properly encoded) concrete description expressed in that formalism.

In this section, after recalling the structural definition of P/T Nets (taken from [78]), we present a corresponding encoding scheme and a behavioural model in the form of a "universal" ASM for P/T nets.

Net Structure A P/T net N is a 6-tuple (P, T, F, K, W, M_0), where

1. $(P, T; F)$ is a net with *places* from a finite set P and *transitions* from a finite set T, i.e. a bipartite directed graph with edges $F \subseteq (P \times T) \cup (T \times P)$;
2. $K : P \to \mathbf{N} \cup \{\infty\}$ is the *capacity* of the places;
3. $W : F \to \mathbf{N} \setminus \{0\}$ is a *weight function* defined on the edges;
4. $M_0 : P \to \mathbf{N} \cup \{\infty\}$ is an *initial marking* satisfying $\forall s \in P : M_0(s) \leq K(s)$.

Encoding of P/T Nets In the corresponding ASM-SL specification, the nodes (places and transitions) are represented by a free type, where places and transitions are generated by the constructors `p` and `t`, respectively, and identified by indices. The net structure as 6-tuple is defined by the type alias `PT_NET` (where `(`T_1`,` $\ldots$ `,`T_n`)` is simply an alternative notation for T_1 `*` $\ldots$ `*` T_n).[13]

```
freetype NODE ==
{ p : INT, (* places *)
  t : INT  (* transitions *) }

typealias EDGE == NODE * NODE
```

[13] In order to deal with infinite capacities and with places with an infinite reserve of tokens (sources), as provided for in the definition, a free type `NAT_PLUS_INF` is introduced, with constructors `Fin:INT` and `Inf` to distinguish between finite quantities and (positive) infinity, as well as arithmetic operations `+#`, `-#` and binary relations `<#`, `<=#`, `>=#`, `>#`, which extend the standard ones in the expected way to deal with `Inf`. Their trivial definition is omitted here.

```
typealias PT_NET ==
( SET(NODE),                  // places
  SET(NODE),                  // transitions
  SET(EDGE),                  // edges
  MAP(NODE, NAT_PLUS_INF),    // place capacities (possibly infinite)
  MAP(EDGE, INT),             // edge weights (always finite)
  MAP(NODE, NAT_PLUS_INF) )   // initial marking of places
```

Clearly, an object of type `PT_NET` represents a valid P/T net only if it satisfies some additional well-formedness conditions, which result from the structural definition above and from the fact that, in our encoding, the distinction between places and transitions is not captured by the type system (in fact, both places and transitions are elements of the same type `NODE`). However, as we focus on behavioural specifications here, we skip the formalization of well-formedness conditions in ASM-SL and simply assume it.[14]

Table 2.5 shows an example: the *Producer-Consumer* net (taken from [78]), represented graphically in the upper part of the table, is encoded in ASM-SL by the static function `N` (of type `PT_NET`) shown in the lower part.

Note that the representation of places and transitions in this example illustrates a modelling pattern constantly recurring in ASM-SL, namely:

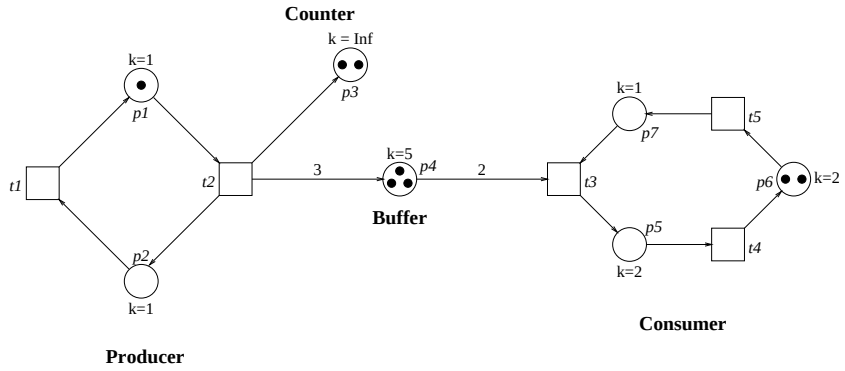
- an infinite supply of elements of some kind, containing all elements of this kind *potentially* needed, i.e., all those which may occur in a possible system description, is defined as a *type* T (here, the type `NODE` of places and transitions);
- the collection of elements *actually* used in a particular system description is restricted to *finite subsets* of T, i.e., elements of type `SET(`T`)` (here, the first two components `P` and `T` of the 6-tuple defining particular a P/T net, see Table 2.5).

Machine State An instantaneous P/T net configuration consists simply of its *marking*, i.e., the tokens contained in its places, represented by a dynamic function `M : NODE -> NAT_PLUS_INF` (see Table 2.6). (Note the use of the `MAP_TO_FUN` operator to extensionally define the initial interpretation of `M` by the finite map `M0`, representing the initial marking.)

Behaviour The ASM-SL specification of P/T net behaviour is shown in Table 2.6. After projection functions to access the single components of the 6-tuple `N` describing the net, the following functions are defined: the above mentioned dynamic function `M` representing the current marking; static functions `pre` and `post`, standing for the preset and the postset of a given transition `t0`; a derived function `active`, corresponding to the transition activation condition.

[14] This is a realistic assumption as, in practice, the well-formedness check is usually done by a front-end which—like the while-language parser in Sect. 2.1—generates the encoding of a given Petri net for the ASM tools.

Example P/T Net: *Producer-Consumer*



Encoding of the *Producer-Consumer* Net in ASM-SL

```
static function N :PT_NET ==
  let P == { p(i) | i in { 1..7 } }
  in let T == { t(i) | i in { 1..5 } }
  in let F == { (p(1),t(2)), (t(2),p(2)), (p(2),t(1)), (t(1),p(1)),
                (t(2),p(3)), (t(2),p(4)), (p(4),t(3)),
                (t(3),p(5)), (p(5),t(4)), (t(4),p(6)), (p(6),t(5)),
                (t(5),p(7)), (p(7),t(3)) }
  in let K == { p(1) -> Fin(1), p(2) -> Fin(1), p(3) -> Inf, p(4) -> Fin(5),
                p(5) -> Fin(2), p(6) -> Fin(2), p(7) -> Fin(1) }
  in let W == override ( { e -> 1 | e in F },
                         { (t(2),p(4)) -> 3, (p(4),t(3)) -> 2 } )
  in let M0 == { i -> Fin(j) | (i,j) in { (p(1),1), (p(2),0), (p(3),5),
                                          (p(4),3), (p(5),0), (p(6),2),
                                          (p(7),0) } }
  in (P, T, F, K, W, M0)
  end end end end end end
```

Table 2.5: P/T Nets: Encoding of the Net Structure

The ASM program is the rule `PT_Net_Step`, which specifies the behaviour of P/T nets using a function `self`, in the style of multi-agent ASMs (see Sect. 1.2.3). The auxiliary rule `PT_Net_Transition` defines the state transition which takes place when a given net transition `t_` is executed.[15]

Assuming an interleaving model of concurrency, the ASM program could be equivalently defined as follows:[16]

```
transition PT_Net_Step ==
  choose self in T with active (self)
    PT_Net_Transition (self)
  end
```

[15] Thanks to the availability of finite sets as a primitive type in ASM-SL, these definitions can be written in a form which is quite close to the corresponding mathematical definitions, as found, for instance, in [78].

[16] Note that `self` is a variable in this case, not an external function.

```
static function P == let (P_,_,_,_,_,_) == N in P_ end
static function T == let (_,T_,_,_,_,_) == N in T_ end
static function F == let (_,_,F_,_,_,_) == N in F_ end
static function K (s_) == let (_,_,_,K_,_,_) == N in apply (K_,s_) end
static function W (e_) == let (_,_,_,_,W_,_) == N in apply (W_,e_) end
static function M0 == let (_,_,_,_,_,M0_) == N in M0_ end

dynamic function M initially MAP_TO_FUN M0

static function pre (t0)  == { s_ | (s_,t_) in F with t_ = t0 }
static function post (t0) == { s_ | (t_,s_) in F with t_ = t0 }

derived function active (t0) ==
  (forall s_ in pre(t0) : (M(s_) -# Fin(W(s_,t0)) >=# Fin(0)))
  and (forall s_ in post(t0) : (M(s_) +# Fin(W(t0,s_)) <=# K(s_)))

transition PT_Net_Transition (t_) == block
  do forall s_ in pre(t_) \ post(t_)
    M(s_) := M(s_) -# Fin(W(s_,t_))
  end
  do forall s_ in post(t_) \ pre(t_)
    M(s_) := M(s_) +# Fin(W(t_,s_))
  end
  do forall s_ in pre(t_) intersect post(t_)
    M(s_) := M(s_) -# Fin(W(s_,t_)) +# Fin(W(t_,s_))
  end
end

external function self :NODE
  with self in T

transition PT_Net_Step ==
  if active (self)
  then PT_Net_Transition (self) end
```

Table 2.6: P/T Nets: Behavioural Specification

Note that, although ASM-SL and the ASM Workbench only support an interleaving model for multi-agent ASMs (realized either by declaring `self` as an external function, as explained in Sect. 1.2.3, or by means of a `choose` rule, as shown above), specifications could in principle be interpreted within other semantic frameworks, such as partially ordered runs (see [45]).[17]

[17]The choice of the interleaving model for ASM-SL is simply motivated by reasons of simplicity. The question, whether and under which conditions interleaving is an appropriate model of concurrency is discussed, for instance, in [65], Chapter 2. However, in the context of this thesis, it is not a question of primary importance.

Chapter 3

Syntax

In this chapter we define the syntax of ASM-SL (in EBNF notation) and, to some extent, its informal semantics. To simplify the language specification, we distinguish between *basic forms* and *derived forms*: basic forms constitute the *core* of ASM-SL, while derived forms can be seen as "macros" and reduced syntactically to basic forms. The semantics of the core language will be defined in Chapters 4 and 5, while the semantics of derived forms results from their expansion into basic forms and from the core language semantics. The complete ASM-SL syntax (basic forms and derived forms with the corresponding expansions) is summarized, in a more concise form, in Appendix C.

3.1 Lexical Conventions

ASM-SL terminals (tokens)—whose lexical structure is described in detail in Appendix A—are: type variables (*typevar*), special constants (*int_const*, *float_const*, *string_const*), identifiers (*id*), and keywords (listed in Appendix A and set in `typewriter` font within grammar productions).[1]

All identifiers belong to the same name space. However, for better readability, we distinguish between identifiers of different kinds by using different terminals in place of *id*, depending on the context, namely: *var_id* for variables, *type_id* for type names (also known as "type constructors"), *fun_id* for function names, *infix_op* for infix operators, *rule_id* for transition rule names.

Most of the grammar symbols—terminals as well as non-terminals—have an abbreviated (one-letter) form, used whenever needed to save space. Moreover, in order to make the notation more suggestive, syntactic objects (e.g., terms) expected to be of a given type in a certain context have an additional short form: for instance, the short form G (for "guard") can be used in contexts where a boolean term is expected. These short forms are listed in Table 3.1.

[1] Thus, all symbols typeset in *italics* in productions, which are neither type variables, nor special constants, nor identifiers, are non-terminals.

Syntactic category	Long form	Short form
Type name	*type_id*	T
Type variable	*typevar*	α
Basic type	*type*	τ
Function type	*func_type*	$\tau_1 \rightarrow \tau_2$
Function name	*fun_id*	f
Bool. function name (*predicate*)	*fun_id* [: ... → `BOOL`]	P
Transition rule name	*rule_id*	r
Variable	*var_id*	v
Term	*term*	t
Boolean term (*guard*)	*term* [`:BOOL`]	G
Term of a list type	*term* [`:LIST(...)`]	L
Term of a set type	*term* [`:SET(...)`]	A
Term of a map type	*term* [`:MAP(...)`]	M
Pattern	*patt*	p
Transition rule	*rule*	R
Definition	*def*	D

Table 3.1: Short Forms for ASM-SL Grammar Symbols

Another notational convention adopted here is to put indices on production symbols whenever needed for further reference.

3.2 Types

The type system of ASM-SL is a restricted and slightly modified version of the polymorphic type system of ML, where only first-order functions are allowed (see Chapter 4 for details). *Basic types*, i.e., types of values, are syntactically distinguished from (first-order) *function types*. The syntax of basic types is:

type	→	*typevar*	type variable
	|	`( )`	empty tuple type
	|	*type* `*` *type* (`*` *type*)*	tuple type (n-ary, $n > 1$)
	|	*type_id*	type construction (nullary)
	|	*type_id* `(` *type* (`,` *type*)* `)`	type construction (n-ary, $n > 0$)

The syntax of function types is:

func_type	→	$type_1$ `->` $type_2$	function type

The following derived forms allow to parenthesize types and provide an alternative notation for tuple types and for the predefined polymorphic types of lists, finite sets, and finite maps:

$$
\begin{array}{rcl}
\texttt{(}\ \mathit{type}\ \texttt{)} & \equiv & \mathit{type} \\
\texttt{(}\ \mathit{type}_1\ \texttt{,}\ \mathit{type}_2\ (\texttt{,}\ \mathit{type}_i)^*\ \texttt{)} & \equiv & \mathit{type}_1\ \texttt{*}\ \mathit{type}_2\ (\texttt{*}\ \mathit{type}_i)^* \\
\texttt{[}\ \mathit{type}\ \texttt{]} & \equiv & \texttt{LIST (}\ \mathit{type}\ \texttt{)} \\
\texttt{\{}\ \mathit{type}\ \texttt{\}} & \equiv & \texttt{SET (}\ \mathit{type}\ \texttt{)} \\
\texttt{\{}\ \mathit{type}\ \texttt{->}\ \mathit{type}\ \texttt{\}} & \equiv & \texttt{MAP (}\ \mathit{type}_1\ \texttt{,}\ \mathit{type}_2\ \texttt{)}
\end{array}
$$

(Note that the syntax of these derived forms recalls the syntax of terms denoting the values of the respective types.)

Examples of Types

Examples of basic types are the following, where in each line different syntactic variants of the same type are shown:

```
LIST (SET (INT)) ≡ [SET (INT)] ≡ LIST ({INT}) ≡ [{INT}]
MAP (STRING, INT) ≡ { STRING -> INT }
STRING * LIST (INT) * SET (BOOL) ≡ (STRING, [INT], {BOOL})
```

3.3 Terms

We distinguish between *basic terms* and *variable-binding terms*, for reasons which will become clear in Sect. 3.4, where the latter are extended to include *pattern matching*.

3.3.1 Basic Terms

Basic terms are—as in first-order logic—variables or function applications, with the addition of a few special forms (special constants, tuple terms, enumerations, conditional terms, and the special operators `FUN_TO_MAP` and `REL_TO_SET`).

Special Constants *Special constants*[2] denote integers, floating-point numbers, and strings (see Appendix A for details on their lexical structure):

term	→	*int_const*	integer constant
	|	*float_const*	floating-point constant
	|	*string_const*	string constant

Variables *Applied occurrences of variables*[3] are terms:

term	→	*var_id*	variable (applied occurrence)

[2] As opposed to "ordinary" constants (which are 0-ary static function names from a finite signature), there are infinitely many special constants.

[3] We distinguish between *applied occurrences* and *binding occurrences* of variables. Binding occurrences bind a given variable within the scope of a syntactic construct, while applied occurrences of that variable are its *uses* in the scope of that construct. As an example, consider the formula "$\forall x : P(x, x)$": the first occurrence of x is a *binding* occurrence of x, while the other two occurrences are *applied* occurrences.

Tuples Tuple terms denote tuples (which are admissible values, see Sect. 5.1.1) and have the following syntax:

term	→	`(` `)`	empty tuple term
	|	`(` *term* (`,` *term*)* `)`	tuple term

Note, however, that the form "`(` *term* `)`" is simply a parenthesized term, equivalent to "*term*".

Function Applications Function application terms are defined as usual, with an additional form for infix operators (see App. A.5):

term	→	*fun_id*	function application (nullary)
	|	*fun_id* `(` *term* (`,` *term*)* `)`	function application (n-ary, $n > 0$)
	|	*term infix_op term*	function application (infixed)

The basic form of function application is "*fun_id* `(` *term* `)`" (unary application), while all other forms are derived. The form in the first production, used when *fun_id* is a nullary function name, is equivalent to "*fun_id* `(())`". When the arity of *fun_id* is $n > 1$, the form in the second production is equivalent to "$f\,(\overline{t})$", where $\overline{t}$ is the tuple term "$(t_1\,,\ldots,t_n)$". The form in the third production is reduced to a basic form as follows:

term$_1$ *infix_op term*$_2$ ≡ `op` *infix_op* `(` *term*$_1$ `,` *term*$_2$ `)`

The expansion of terms containing infix operators, as specified by the above rule, respects the operator precedences and associativities, as defined in App. A.5.

Enumerations Lists, finite sets and finite maps can be denoted by special *enumeration* terms (which are all derived forms). The syntax is as follows. For lists:

term	→	`[` `]`	empty list
	|	`[` *term* (`,` *term*)* `]`	list enumeration (n elements, $n > 0$)

For finite sets:

term	→	{ }	empty set
	|	{ *term* (`,` *term*)* }	set enumeration (n elements, $n > 0$)

For finite maps:

term	→	{ *t* `->` *t* (`,` *t* `->` *t*)* }	map enumeration (n elements, $n > 0$)

(The form "{ }" to denote empty maps is not allowed, because the grammar would be ambiguous otherwise.)

List, set and map enumeration terms are reduced to basic forms according to the following rules:

$$
\begin{aligned}
\texttt{[ ]} &\equiv \texttt{nil} \\
\texttt{[}\ t_1\ (\texttt{,}\ t_i)^*\ \texttt{]} &\equiv t_1\ (\texttt{::}\ t_i)^*\ \texttt{::}\ \texttt{nil} \\
\{\ \} &\equiv \texttt{emptyset} \\
\{\ t_1\ (\texttt{,}\ t_i)^*\ \} &\equiv \texttt{list\_to\_set}\ (\texttt{[}\ t_1\ (\texttt{,}\ t_i)^*\ \texttt{]}) \\
\{\ t_{11}\ \texttt{->}\ t_{12}\ (\texttt{,}\ t_{i1}\ \texttt{->}\ t_{i2})^*\ \} &\equiv \texttt{set\_to\_map}\ (\{\ (t_{11}\,,t_{12})\ (\texttt{,}\ (t_{i1}\,,t_{i2}))^*\ \})
\end{aligned}
$$

(where `nil`, `::`, `emptyset`, `list_to_set`, and `set_to_map` are primitive functions of ASM-SL with the obvious meaning, see also Appendix B).

Intervals List and finite sets over integers can also be defined by means of an interval notation:

term	→	`[` *term* `..` *term* `]`	list interval
	|	{ *term* `..` *term* }	set interval

Interval terms are derived forms, expanded into basic forms as follows:

`[` $term_1$ `..` $term_2$ `]`	≡	`list_interval (` $term_1$ `,` $term_2$ `, 1 )`
{ $term_1$ `..` $term_2$ }	≡	`set_interval (` $term_1$ `,` $term_2$ `, 1 )`

(where `list_interval` and `set_interval` are primitive functions of ASM-SL, see Appendix B).

Conditionals For notational convenience, there is a special *conditional term*, with a syntax similar to the conditional rule:

term	→	`if` *term* `then` *term* (`elseif` *term* `then` *term*)* [`else` *term*] `endif`	conditional term

The basic form of the conditional term is

`if` G `then` t_T `else` t_F `endif`.

Other conditional terms are reduced to this form according to the following rules:

- Omitting the `else` clause is equivalent to specifying `else undef`
- `Elseif` clauses are transformed into nested conditionals:

  `if` G_1 `then` t_1 `elseif` G_2 `then` t_2 `...` `endif` ≡
  ≡ `if` G `then` t_1 `else if` G_2 `then` t_2 `...` `endif endif`

`FUN_TO_MAP` and `REL_TO_SET` The special operator `FUN_TO_MAP` allows to "convert" a function $f : \tau \to \tau'$, into a finite map (i.e., a value of type `MAP(`τ`,`τ'`)`) constituting a representation of f. Similarly, `REL_TO_SET` converts a boolean function $P : \tau \to$ `BOOL` into a finite set (i.e., a value of type `SET(`τ`)`) representing P. The syntax is as follows:

term	→	`FUN_TO_MAP` f	function to finite map conversion
	|	`REL_TO_SET` P	relation to finite set conversion

Note the following restriction: `FUN_TO_MAP` and `REL_TO_SET` can only be applied to extensionally defined functions and relations[4], which are finite by construction.

[4]Extensionally defined functions and relations are those functions and relations defined by means of the `MAP_TO_FUN` and `SET_TO_REL` operators, respectively (see Sect. 3.6 and 5.2.4).

Examples of Basic Terms

Examples of basic terms built out of special constants, variables, tuples, and function applications:

```
1 + 2 * f (x) ≡ op + (1, op * (2, f (x)))
(1 + 2) * f (x) ≡ op * (op + (1, 2), f (x))
(x div y, x mod y) ≡ (op div (x, y), op mod (x, y))
```

Examples of enumeration terms, with the corresponding expansions:

```
[] ≡ nil
[1, 2] ≡ 1::2::nil
{ [], [1], [1,2] } ≡ list_to_set (nil::(1::nil)::(1::2::nil)::nil)
{ (0,0) -> 0, (1,0) -> 1, (0,1) -> 1, (1,1) -> 1 } ≡
   ≡ set_to_map ({ ((0,0),0), ((1,0),1), ((0,1),1), ((1,1),1)) })
```

Example of interval terms, with the corresponding expansions:

```
[ 1..4 ] ≡ list_interval (1, 4, 1)
{ 0..n-1 } ≡ set_interval (0, n-1, 1)
```

Example of conditional term:

```
if x > 0 then 1
elseif x = 0 then 0
else 0-1 endif
```

For an example of term involving the `MAP_TO_FUN` operator, see the definition of function `eval` in Sect. 2.1 (the `REL_TO_SET` operator is analogous).

3.3.2 Variable-Binding Terms

Variable-binding terms are `let`-terms, comprehension terms, and finite quantifications.

Let-Terms `Let`-terms allow to evaluate a term $term_2$ with a given variable bound to the value resulting from the evaluation of another term $term_1$:

term	→	`let` *var_id* `==` $term_1$ `in` $term_2$ `endlet`	let term

Comprehension Terms Following [10], set comprehension can be used as a construct to denote finite sets. Additionally, map and list comprehensions are also supported:

term	→	{ t `|` v `in` A [`with` G] }	set comprehension
		{ t_1 `->` t_2 `|` v `in` A [`with` G] }	map comprehension
		[t `|` v `in` L [`with` G]]	list comprehension

The term following `in` must denote a finite set, in the case of set and map comprehensions, or a list, in the case of list comprehensions. The term following

`with` must be boolean. Omitting the `with` clause is equivalent to specifying `with true`.

The meaning of the comprehension notation can be defined in terms of the basic constructs as follows. A list comprehension `[` t_v `|` v `in` L `with` G_v `]` is equivalent to a term F`(L)`, where F is a fresh function name, whose interpretation is given by a recursive definition:

```
F(L)  ≡  if L = [] then []
         else let v == hd(L)
              in if G_v then t_v ::  F(tl(L))
                        else F(tl(L))
         endif endlet endif
```

(where the subscript v in t_v, G_v should simply suggest that t and G usually depend on v, while `hd` and `tl` are primitive ASM-SL functions to select the head and the tail of a list, respectively).

Set and map comprehensions can then be expressed as derived forms according to the following equivalences:

```
{ t | v in A with G }  ≡
   ≡  list_to_set ([ t | v in set_to_list(A) with G ])

{ t1 -> t2 | v in A with G }  ≡
   ≡  set_to_map ({ (t1,t2) | v in A with G })
```

(`list_to_set`, `set_to_list`, and `set_to_map` are primitive ASM-SL functions, see Appendix B).

Finite Quantifications Finite quantifications, where the variable range is given by (a term denoting) a finite set, are admissible boolean terms:

term	→	`(` `exists` v `in` A [`:` G] `)`	finite existential quantification
	|	`(` `forall` v `in` A [`:` G] `)`	finite universal quantification

Omitting the condition "`:` G" is equivalent to specifying "`: true`".

Finite quantifications are derived forms and can be expressed in terms of set comprehensions as follows:

```
( exists v in A : G )  ≡  ({ v | v in A with G } != { })
( forall v in A : G )  ≡  ({ v | v in A with G } = A )
```

Examples of Variable-Binding Terms

Example of `let`-term:

```
let succ_x == x + 1
in succ_x * succ_x
endlet
```

Examples of comprehension terms (for lists, sets, and maps, respectively):

```
[ f(x) | x in [ 0..2*n-1 ] with x mod 2 = 0 ]
{ 2 * x | x in { 0..n-1 } }
{ x -> x * x | x in { 2, 3, 5, 7, 11 } }
```

Note that, according to the translation scheme introduced here, the list comprehension term from the example above corresponds to a term `F([ 0..2*n-1 ])`, where `F` is recursively defined as follows:

```
static function F(L) ==
  if L = [] then []
  else let x == hd (L)
       in if x mod 2 = 0
          then f(x) :: F(tl(L))
          else F(tl(L))
  endif endlet endif
```

Examples of finite quantifications, with the corresponding expansions:

```
( exists x in A : x mod 2 = 0 )   ≡
  ≡   { x | x in A with x mod 2 = 0 } != {}
( forall x in A : y >= x )   ≡
  ≡   { x | x in A with y >= x } = A
```

3.4 Patterns and Pattern Matching

A form of *pattern matching* as found, for instance, in functional languages (such as ML or Haskell), is supported. This allows for more concise and readable specifications, especially when tuples and free types are intensively used.

3.4.1 Patterns

From the syntactic point of view, patterns can be considered as a special case of terms, where—essentially—only constructors are allowed as function names.[5] However, as the semantics of patterns is very different from the semantics of terms (see Sect. 5.2.1 and 5.2.2, respectively), it is more convenient to define them as two distinct syntactic categories.

As soon as the language is extended with pattern matching, patterns can occur in any place where a binding occurrence of a variable can occur.[6] More precisely, in all productions for variable-binding terms, the *var_id* standing for the variable's binding occurrence in the variable-binding term is replaced by *patt*. The syntax of patterns is as follows.

Variables Variables are patterns. They possibly come with a type constraint (if no type constraint is specified, the variable type is inferred from the context):

patt	→	*var_id*	variable (binding occurrence)
	|	(*var_id* : *type*)	idem, with type constraint

Note that any occurrence of a variable in a pattern is a *binding occurrence*.

[5] Note that constructors include not only constructors of free type values, but also special constants and tuple constructors.

[6] In fact, a binding occurrence of a variable is a special case of pattern.

Placeholders A placeholder (`_`) in a pattern has the same meaning as a variable (it matches anything), but no variable is bound as a result of the match:

patt	→	`_`	placeholder

Other Patterns Besides variables and placeholders, the following are patterns:

patt	→	*int_const*	integer constant
	|	*float_const*	floating-point constant
	|	*string_const*	string constant
	|	`( )`	empty tuple pattern
	|	`(` *patt* (`,` *patt*)* `)`	tuple pattern
	|	*fun_id*	constructor application (nullary)
	|	*fun_id* `(` *patt* (`,` *patt*)* `)`	constructor application (n-ary, $n > 0$)
	|	*patt infix_op patt*	constructor application (infixed)
	|	`[ ]`	list pattern (empty)
	|	`[` *patt* (`,` *patt*)* `]`	list pattern (non-empty)

The expansions of derived forms (parenthesized patterns, n-ary applications with $n \neq 1$, infixed applications, and list patterns) are analogous with those defined for the corresponding terms.

Syntactic Restrictions on Patterns

- Only constructors may occur as function names in application patterns.[7]
- No variable may occur twice in a pattern.

Examples of Patterns

The following are examples of well-formed patterns:

```
true
[]
x :: y
(x, 1, [z])
```

Note that the following, instead, are not well-formed patterns:

```
x + y
(x, 1, [x])
```

The first one is not well-formed because the `+` operator—as opposite to the list constructor `::`—is not a constructor, the second one because the variable `x` occurs twice. (In general, a pattern with more occurrences of the same variable can be replaced by a pattern with distinct variables, which are then tested for equality.)

[7] Constructors include the built-in constants `true`, `false`, `undef`, list constructors `::` (infix) and `nil`, and any constructors declared by the user within free type definitions (Sect. 3.8.1).

3.4.2 Case-Terms

The main construct involving pattern matching is the `case`-construct, which is quite similar—both syntactically and semantically—to the omonymous construct found in functional languages.[8] The syntax of `case`-terms is:

term → `case` *term* `of` case term
patt `:` *term*
(`;` *patt* `:` *term*)*
[`;` `otherwise` *term*]
`endcase`

To simplify matters, we consider

`case` t_0 `of` p `:` t_1 `;` `otherwise` t_2 `endcase`

as the basic form of the `case`-term (in a similar way as we did for conditionals). Other `case`-terms are reduced to nested basic forms according to the following rules:

- An omitted `otherwise` clause corresponds to `otherwise undef`
- Sequences of pattern-term pairs are resolved as follows:

`case` t_0 `of` p_1 `:` t_1 `;` p_2 `:` t_2 ... `endcase` ≡
≡ `case` t_0 `of` p_1 `:` t_1 `;` `otherwise`
`case` t_0 `of` p_2 `:` t_2 `;` `otherwise`
...
`endcase`
`endcase`

`Case` is the basic construct to deal with pattern matching: other constructs involving pattern matching will be considered as derived forms and expressed in terms of `case`.

Let-Terms `Let`-terms extended with patterns are translated into `case`-terms according to the following scheme:

`let` p `==` t_1 `in` t_2 `endlet` ≡ `case` t_1 `of` p `:` t_2 `endcase`

As a result, if the pattern p does not match the value denoted by t_1, the value of the `let`-term is *undef*.[9]

Comprehensions and Quantifications The definition of list comprehensions in terms of an auxiliary recursive function F (introduced in Sect. 3.3.2) must be refined to deal with pattern matching. In particular, the comprehension range consists of those elements of the list L which match the pattern p, in addition to satisfying the condition G.

[8] More precisely, there are two `case`-constructs in ASM-SL, the `case`-term (discussed here) and the `case`-rule (introduced in Sect. 3.5).

[9] However, as `let` is mostly used to match simple variables or tuples (where pattern matching can not fail), the evaluation of a well-typed `let`-term usually does not yield *undef*.

More precisely, a list comprehension `[` $t_{v(p)}$ `|` p `in` L `with` $G_{v(p)}$ `]` is equivalent to F`(`L`)`, with F defined as:

```
F(L)  ≡  if L = [] then []
         else case hd(L) of
                 p :  if G_v(p)
                      then t_v(p) ::  F(tl(L))
                      else F(tl(L)) endif ;
                 otherwise F(tl(L))
              endcase
         endif
```

(The subscript $v(p)$ in $t_{v(p)}$, $G_{v(p)}$ should suggest that t and G may depend on the variables $v(p)$ bound by the pattern p).

The translation schemes for set and map comprehensions, defined in terms of list comprehensions, are unchanged, as well as those for finite quantifications (with every binding occurrence of a variable v replaced by a pattern p).

Examples of Variable-Binding Terms with Pattern Matching

Example of `case`-term (equivalent to "`tl(L)`"):

```
case L of
  x :: xs : xs ;
  []      : undef
endif
```

Example of `let`-term:

```
let [x] == one_element_list in f (x) endlet
```

Note that the `let`-term above evaluates to *undef* if the number of elements in `one_element_list` is not one (otherwise it extracts the element from the list and applies `f` to it), as the `let`-term expands to

```
case one_element_list of
  [x] : f (x) ;
  otherwise undef
endcase
```

Example of comprehension term:

```
[ x + y | (x, y) in list_of_pairs with x < y ]
```

The list comprehension above, according to the translation scheme introduced here, corresponds to "`F(list_of_pairs)`", where `F` is defined as follows:

```
static function F(L) ==
  if L = [] then []
  else case hd(L) of
         (x, y) : if x < y then (x + y) :: F(tl(L))
                           else F(tl(L)) endif;
         otherwise F(tl(L))
       endcase
  endif
```

3.5 Transition Rules

With respect to transition rules, ASM-SL follows [45, 46], with the following exceptions:

- The ranges of `do forall` and `choose` rules are defined by finite sets (possibly restricted by a boolean term), and not by arbitrary predicates.[10]
- The pattern-matching `case` rule has been introduced into the language, as free types and pattern matching are an integral part of ASM-SL. Moreover, the syntax and semantics of rules is influenced by the pervasive use of pattern matching in ASM-SL: as already discussed in Sect. 3.4 in relation to terms, patterns can occur in every place where a binding occurrence of a variable can occur (in particular, in variable-binding rules such as `do forall` and `choose`).
- Motivated by the widespread informal use of "macros" in many papers which use ASMs as a specification or modelling language, constructs for defining and using parameterized *named transition rules* have been added to the language. In particular, *named rule application* is a form of transition rule in ASM-SL (just as function application is a form of term).
- The `import` rule is not supported.

Basic Rules Basic transition rules are skip, update, block, and conditional rules. Block rules come in two (fully equivalent) syntactic forms, an implicit one, where any sequence of at least two rules is considered as a block, and an explicit one, where the rules constituting the block are delimited by `block ... endblock` (the explicit form is sometimes needed to avoid ambiguities which arise in some contexts). The syntax of basic rules is otherwise standard.

rule	→	`skip`	skip rule
	|	*term* `:=` *term*	update rule
	|	*rule rule* $(rule)^*$	block rule (implicit)
	|	`block` $(rule)^*$ `endblock`	block rule (explicit)
	|	`if` *term* `then` *rule* $(\texttt{elseif}\ term\ \texttt{then}\ rule)^*$ $[\,\texttt{else}\ rule\,]$ `endif`	conditional rule

Just as conditional terms, conditional rules have a ternary basic form, to which arbitrary conditional rules can be reduced (see "Conditionals" in Sect. 3.3.1).

[10] This change was inspired by [10], which introduces a variant of ASMs where finite sets are elements of the base set and the ranges of `do forall` rules are finite sets. For practical applications, this variant of ASMs is much more useful than the original definition in [45]: in fact, if one strictly follows the definition of [45], the problem of determining the range of `do forall` and `choose` rules is—in general—undecidable.

Do-Forall and Choose Rules Variable-binding rules are the synchronous-parallel `do forall` rule (formerly known as `var` rule [45]), and the `choose` rule, which expresses non-deterministic choice.

rule	→	`do forall` p `in` A [`with` G] *rule* `enddo`	do-forall rule
	|	`choose` p `in` A [`with` G] *rule* `endchoose`	choose rule

(Omitting the `with` clause is equivalent to specifying `with true`.)

Case and Let Rules `Case` and `let` rules have the same syntactic structure as the corresponding terms:

rule	→	`case` *term* `of`
		patt `:` *rule*
		(`;` *patt* `:` *rule*)*
		[`; otherwise` *rule*]
		`endcase`
			case rule

rule	→	`let` *patt* `==` *term* `in` *rule* `endlet`	let rule

The same considerations regarding basic and derived forms and the corresponding reductions also apply (see Sect. 3.4.2), whereas omitting the `otherwise` clause in a `case` rule corresponds to specifying `otherwise skip`.

Named Rule Application The syntax of named rule application is similar to the syntax of function application (except that there is no infixed application).

rule	→	*rule_id*	named rule application (nullary)
	|	*rule_id* `(` *term* (`,` *term*)* `)`	named rule application (n-ary, $n > 0$)

Similar rules as for function application terms are applied to reduce derived forms (n-ary applications with $n \neq 1$) to the basic form *rule_id* `(`*term*`)`.

Syntactic Restrictions on Transition Rules

- The term on the left-hand side of an update rule must be an application term. That is, an update rule is always of the form

  f [`(` t_1 `,` ... `,` t_n `)`] `:=` t

  Moreover, f must be a dynamic function name.

Examples of Transition Rules

See ASM programs in the examples of Chapter 2.

3.6 Function Expressions

Function expressions denote (first-order) functions and only occur, as an auxiliary construct, within function definitions. More precisely, they occur on the right-hand side of definitions of the form

attr `function` *fun_id* `==` *fexpr*,

where *attr* $\in$ { `static`, `dynamic`, `derived`, `external` } is the function attribute. Function expressions allow the definition of functions either intensionally (by λ-abstraction) or extensionally (by means of the special operators `MAP_TO_FUN` and `SET_TO_REL`, which convert a finite map or a finite set into a function or relation, respectively).[11]

fexpr	$\rightarrow$	`fn ( ) ->` t	lambda-term (nullary)
	$\mid$	`fn (` p `(, ` p`)`* `) ->` t	lambda-term (n-ary, $n > 0$)
	$\mid$	`MAP_TO_FUN` M	finite map to function conversion
	$\mid$	`SET_TO_REL` A	finite set to relation conversion

The basic form of lambda-term is `fn (` p `) ->` t. Other lambda-terms are reduced to the basic form according to the following rules:

$$\begin{array}{rcll} \texttt{fn ( ) ->}\ t & \equiv & \texttt{fn (()) ->}\ t & \\ \texttt{fn (}\ p_1, \ldots, p_n\ \texttt{) ->}\ t & \equiv & \texttt{fn (}\ (p_1, \ldots, p_n)\ \texttt{) ->}\ t & (n > 1) \end{array}$$

Note also that, usually, lambda-terms are not written explicitly, but result from the following expansions of derived forms for function definitions:

attr `function` f `==` t $\equiv$
$\equiv$ *attr* `function` f `==fn ( ) ->` t

attr `function` f `(` $p_1, \ldots, p_n$ `) ==` t $\equiv$
$\equiv$ *attr* `function` f `==fn (` $p_1, \ldots, p_n$ `) ->` t $(n \geq 1)$

(where *attr* $\in$ { `static`, `derived` }, as only static and derived functions can be defined intensionally).

3.7 Transition Expressions

In analogy to function expressions, *transition expressions* are an auxiliary construct, which occurs on the right-hand side of named rule definitions of the form

`transition` *rule_id* `==` *texpr*.

There is only one kind of transition expression, the so called *lambda-rule*:

texpr	$\rightarrow$	`tn ( ) ->` R	lambda-rule (nullary)
	$\mid$	`tn (` p `(, ` p`)`* `) ->` R	lambda-rule (n-ary, $n > 0$)

(The analogies with lambda-terms are obvious. In particular, the same considerations regarding derived forms apply.)

[11] See Sect. 5.2.4 for more details.

3.8 Definitions

3.8.1 Type Definitions

There are two kinds of type definitions, namely definitions of *type aliases* and of *freely generated types* (*free types* for short):

def	→	`typealias` T [`(` α (`,` α)* `)`] `==` *type*	type alias definition
	|	`freetype` T [`(` α (`,` α)* `)`] `==` { *fun_id* [`:` *type*] (`,` *fun_id* [`:` *type*])* }	free type definition

Both constructs allow to define polymorphic types. However, while type aliases are only a syntactic shorthand[12], free type definitions introduce new types, distinct from any other type existing prior to those definitions.

Free types can be recursive and mutually recursive. In order to overcome the *definition before use* limitation (see Sect. 3.9 below), there is a special syntax for simultaneous definition of free types:

def → `freetypes` {
  T [`(` α (`,` α)* `)`] `==` { *fun_id* [`:` *type*] (`,` *fun_id* [`:` *type*])* }
  T [`(` α (`,` α)* `)`] `==` { *fun_id* [`:` *type*] (`,` *fun_id* [`:` *type*])* }
  (T [`(` α (`,` α)* `)`] `==` { *fun_id* [`:` *type*] (`,` *fun_id* [`:` *type*])* })*
}

The keyword `datatype(s)` can be used as a synonym for `freetype(s)`.

Examples of Type Definitions

Several type definitions are shown in the examples of Chapter 2. However, no simultaneous free type definitions can be found there. The following example shows definitions of types representing the abstract syntax of a simple language, where the type `DECL` of declarations and the type `STMT` of statements are mutually recursive: in fact, a procedure declaration contains a statement (the procedure body), while a block statement contains a list of local declarations.

```
typealias VALUE   == INT
typealias VAR_ID  == STRING
typealias PROC_ID == STRING

freetypes {
  DECL == { VarDecl  : VAR_ID * VALUE,
            ProcDecl : PROC_ID * LIST(VAR_ID) * STMT }

  STMT == { Assignment : VAR_ID * VALUE,
            ProcCall   : PROC_ID * LIST(VALUE),
            Block      : LIST(DECL) * LIST(STMT) }
}
```

[12] In fact, type aliases are syntactically expanded by the type checker.

3.8.2 Function Definitions

Function definitions—besides indicating whether a function is *static*, *dynamic*, *derived*, or *external* (i.e., the *function kind*)—bind function names to the corresponding function expressions defining their interpretation (or, in the case of dynamic functions, their initialization, i.e., their interpretation in the initial state). Optionally, a type constraint for the function can be specified (for external functions, it *must* be specified, as there is no function expression from which the function type can be inferred):[13]

def → `static function` *fun_id* [`:` *func_type*] `==` *fexpr*
| `derived function` *fun_id* [`:` *func_type*] `==` *fexpr*
| `dynamic function` *fun_id* [`:` *func_type*] `initially` *fexpr*
| `external function` *fun_id* `:` *func_type*

The following restrictions apply:

- The definition of a `static` function may be given by means of any function expression. All function names occurring in the function expression must be static.
- The definition of a `derived` function must be intensional (i.e., *fexpr* must be a lambda-term, as defined in Sect. 3.6).
- The initialization of a `dynamic` function must be specified extensionally (i.e., *fexpr* must be a `MAP_TO_FUN` or `SET_TO_REL` expression, as defined in Sect. 3.6).

Derived Forms for Function Definitions

1. `static function` f `==` t
   ≡ `static function` f `== fn ( ) ->` t
2. `static function` f `(` p_1 (`,` p_i)* `)` `==` t
   ≡ `static function` f `== fn (` p_1 (`,` p_i)* `) ->` t
3. `dynamic function` f `initially` t
   ≡ `dynamic function` f `initially MAP_TO_FUN { ( ) ->` t `}`

The derived forms for `static` function definitions (1. and 2. above, see also Sect. 3.6) apply to definitions of `derived` functions as well.

Simultaneous Function Definitions Intensional function definitions can be mutually recursive. The syntax for simultaneous function definitions is the following:

[13] Additionally, so-called *finiteness constraints* (which will be introduced in Chapter 7) can be specified for dynamic and external functions. They essential to perform certain kinds of analyses and transformations of ASMs (e.g., as needed for model checking).

def → (`static` | `derived`) `functions` {
fun_id `==` *fexpr*
fun_id `==` *fexpr*
(*fun_id* `==` *fexpr*)*
}

Infix Operators The function being defined can be declared as an infix operator (with corresponding associativity and precedence). The concrete notation for infix operators is explained in App. A.5.

Examples of Function Definitions

Example of intensional function definition:

```
static function list_length (L) ==
  case L of
    []      : 0 ;
    _ :: xs : 1 + list_length (xs)
  endcase
```

Example of extensional function definition:

```
static function three_bit_and ==
  MAP_TO_FUN {
    (0,0,0) -> 0,  (0,0,1) -> 0,  (0,1,0) -> 0, (0,1,1) -> 0,
    (1,0,0) -> 0,  (1,0,1) -> 0,  (1,1,0) -> 0, (1,1,1) -> 1
  }
```

Example of simultaneous function definitions:

```
static functions {
  eval_expr (E, env) ==
    case E of
      Con (x)         : Int (x) ;
      Var (v)         : apply (env, v) ;
      App (f, E_list) :
        interpretation (f, eval_expr_list (E_list, env)) ;
      Let (x, E1, E2) :
        eval_expr (E2, override (env, { x -> eval_expr (E1, env) }))
    endcase

  eval_expr_list (E_list, env) ==
    case E_list of
      []      : [] ;
      E :: Es : eval_expr (E, env) :: eval_expr_list (Es, env)
    endcase
}
```

Note that the function `eval_expr` above has the same meaning as `eval_in_env` presented in Sect. 2.1: however, mutual recursion is used instead of a list comprehension in order to evaluate the arguments of an application term "`App(...)`".

Example of infix operator definition:

```
static function op_l 5 ++ (x, y) == (x + y) mod 10
```

3.8.3 Named Rule Definitions

Named rule definitions are similar to function definitions. They bind rule names to the corresponding (possibly parameterized) transition expressions:

def $\rightarrow$ `transition` *rule_id* [`:` *type*] `==` *texpr*

In analogy to function definitions, the following derived forms are defined:

$$\texttt{transition}\ r\ \texttt{==}\ R \ \equiv\ \texttt{transition}\ r\ \texttt{==}\ \texttt{tn ( ) ->}\ R$$
$$\texttt{transition}\ r\ \texttt{(}\ p_1\ \texttt{(,}\ p_i\texttt{)}^*\ \texttt{)}\ \texttt{==}\ R \ \equiv\ \texttt{transition}\ r\ \texttt{==}\ \texttt{tn (}\ p_1\ \texttt{(,}\ p_i\texttt{)}^*\ \texttt{) ->}\ R$$

(See Table 2.1 in the previous chapter for examples of named rule definitions.)

3.9 Specifications

An ASM-SL specification is a sequence

$$spec \ \equiv\ D_1 \ldots D_n$$

of definitions (of types, functions, and named rules, collectively referred to as *entities* below). The specification is typed and evaluated ("processed" for short) within an *initial context*, containing information about the primitive ASM-SL types and functions, as listed in Appendix B.[14]

The processing of each definition extends the current context by adding information related to the entities specified in that definition. The processing of a definition sequence (such as a specification) is carried out sequentially. The entities to which a definition refers must be available in the context in which the definition is processed (*definition before use* requirement). At the end, the context will include the relevant information about all entities defined in the ASM specification. After this preliminary phase ("loading"), the context is fixed: only at this point it is possible to evaluate terms, execute transition rules, perform analyses and transformations of the specification, and so on.[15]

Finally, note that there is no ASM-SL construct to specify the ASM *program*. The user can decide, from time to time, which one of the named rules of the current context should be considered as the program. As this information is not part of an ASM-SL specification, the supporting tools have to provide some means to allow the user to identify the program.

[14] The context is essentially a "database" of entities, which contains—for instance—signature information.

[15] In fact, it would not be very sensible to allow context changes, for instance, between two steps of an ASM runs.

Chapter 4

Type System

In this chapter, we introduce the ASM-SL type system and define *typing rules* for all its constructs, except those which are completely standard. Type checking, according to the rules presented here, must always be performed before any further processing of ASM-SL specifications is carried out, as in the dynamic semantics (Chapter 5) it is assumed that programs are well-typed.

4.1 Basic Notions

The ASM-SL type system is a restricted version of the polymorphic type system of the functional language ML [72]. Types τ can be constructed by means of:[1]

- *type variables* `'a`, `'b`, ... for expressing polymorphism (i.e., standing for "any type");
- *tuple types* (cartesian products), written "τ_1 `*` ... `*` τ_n" for $n \geq 2$, which include the type "`()`" of the empty tuple ($n = 0$);
- *type constructors* (*type names*) of a given arity, such as the nullary `BOOL` and `INT` (elementary types), the unary `SET(` $\cdot$ `)` for (polymorphic) finite sets, the binary `MAP(`$\cdot, \cdot$`)` for finite maps, etc.

Any type constructed by means of type parameters, cartesian products and type constructors is a *basic type*. Additionally, $\tau' \to \tau$ is a *function type* for any basic types τ' and τ (no higher-order functions are allowed).[2] Types containing no type variables are called *monomorphic types* (as opposite to polymorphic types).

We do not give further details about the type system here, as it is well known (see [70, 27, 73] for details). We only mention the points where the ASM-SL type system differs from the ML type system.

[1] Note that types have a similar structure as first-order terms: type variables correspond to variables, type constructors to function names.

[2] From a strictly formal point of view, all functions are unary in ASM-SL: however, we consider functions defined over τ_1 `*` ... `*` τ_n, $n \neq 1$, as n-ary functions. A type constraint of the form $a : \tau$ for a nullary function a is a shorthand for a : `()` $\to \tau$.

Restriction to First-Order Higher-order functions are not allowed. Moreover, as functions are not values, function types can be syntactically distinguished from basic types (see also Sect. 3.2). In fact, they never appear in the same phrase context: function types may occur only as type constraints for function names, basic types only as type constraints for variables. Furthermore, there is no need for "equality types", as all basic types in ASM-SL admit equality.[3]

Treatment of *undef* The definition of Abstract State Machines in [45], based on total algebras, prescribes that every signature must contain a static nullary function name *undef* (interpreted as a distinguished element of the base set), in order to imitate partial functions.[4] In a typed version of ASMs based on a polymorphic type system, it seems natural to consider *undef* as a polymorphic function "`undef : () -> 'a`". However:

- `undef` should not be defined on `BOOL`, as we want to consider boolean functions as predicates of classical first-order logic;
- `undef` should not be defined on tuple types: in fact, tuple types are used (among other things) to define n-ary functions and, if we would allow `undef` as a legal n-tuple, the arity of functions could be broken.

The obvious solution to this problem is to distinguish between types which admit `undef` and types which do not (just as ML does with equality types). Types which admit `undef` will be called *u-types* for short. U-types are all types except `BOOL` and tuple types (including the unit type "`()`"). Type variables which can only be matched by u-types will be called *u-type variables* and, in ASM-SL programs, lexically distinguished from ordinary type variables by prefixing `'u` to them (e.g., `'a` is an ordinary type variable, while `'u'a` is a u-type variable). Note that u-type variables are u-types, while ordinary type variables are not.

Now, if we define the type of *undef* as "`undef : () -> 'u'a`" and introduce a type generalization relation which deals properly with u-type variables (see Sect. 4.2 below), the problem of dealing with *undef* is solved.

4.2 Typing Rules

Well-typedness conditions for ASM-SL phrases (i.e., their "static semantics") are specified—as usual in the literature, e.g. [81, 72]—by means of *typing rules*, a special kind of inference rules which allow to derive valid typing judgments for all (and only for) well-typed phrases [23, 73]. As the intention here is to document

[3] In ML, the notion of *equality types* is introduced in order to avoid the undecidable problem of testing function equality: for instance, if f and g are function names, the term $f = g$ is rejected by the ML type checker, even if f and g are of the same type, because function types are not equality types, i.e., their elements can not be tested for equality.

[4] For instance, division can be defined as a total function by setting $x/0 = \mathit{undef}$ for each x. Moreover, in the original (untyped) definition of ASMs, the interpretation of x/y should also be *undef* if x is not a number or y is not a number.

the language rather than to discuss the type inference problem, we specify a set of rules which allow to derive every well-typed phrase together with any type admissible for that phrase, instead of rules which specify a type inference algorithm computing the most general type for that phrase. Type inference can be performed by standard unification-based algorithms [27, 73]. Before presenting the typing rules, we have to introduce a few notational conventions and briefly recall some basic definition concerning polymorphic type systems.

Basic Definitions

Notational Conventions The greek letters Σ and π are used to denote *signatures* and *type environments*, respectively. Signatures are finite maps associating function names and rule names to corresponding function types (or, more precisely, type schemes—see below). Type environments are finite maps associating variables to corresponding basic types.[5]

Well-Formed Types We assume that all types τ occurring in typing rules or mentioned in the text are well-formed. A type $T(\tau_1, \ldots, \tau_n)$ is well-formed iff n is the arity of type constructor T.

Type Schemes Within signatures, polymorphic functions are associated to *type schemes*, i.e., universally quantified types.[6] For example, type schemes for the list constructors "`::`" and "`nil`" ("`[]`") are "∀`'a . 'a * ['a] -> ['a]`" and "∀`'a . () -> ['a]`", respectively. Intuitively, this means that a function name can assume a different type each time it is applied, provided that this type is an *instance* of the corresponding type scheme (this intuition is formalized by the notion of type generalization, see below). On the contrary, a free occurrence of a type variable stands for a *particular* type (and every other free occurrence of that type variable also stands for the same type). For example, consider the terms

$$\begin{aligned} t_1 &\equiv \texttt{[] :: []} \\ t_2 &\equiv \texttt{let x == [] in x :: x endlet}. \end{aligned}$$

The term t_1, which denotes a list containing an empty list, is well-typed, its most general type being `[['a]]` (list of list of `'a`): this can be obtained by assigning types `() -> ['a]` to the first occurrence of `[]` and `() -> [['a]]` to the second, as both are valid instances of the type scheme for `[]`. On the contrary, the term t_2 is not well-typed: in fact, there is no type assignment $\{\texttt{x} \mapsto \tau\}$ in whose context the term `x :: x` is well-typed (note that none of the above instances of the type scheme for `[]`, i.e., neither $\tau \equiv$ `['a]` nor $\tau \equiv$ `[['a]]`, leads to a valid

[5] As signatures and type environments are finite maps, all finite map operations mentioned under "Notational Conventions" in the introduction are defined on them.

[6] Note that type schemes are necessary at the meta-level for a proper definition of the typing rules, but are never written explicitly in programs (there is not even a syntax for them). They come into existence by building the *closure* of a type: in ASM-SL, this happens in the context of function and named rule definitions.

instance of the type scheme for `::`, as the occurrence of `'a` in τ is free and can not be further substituted).

Type Generalization A type scheme $\sigma \equiv \forall\alpha_1 \ldots \alpha_n.\tau$ *generalizes* a type τ' (or, equivalently, τ' *is an instance of* σ), written $\sigma \succ \tau$, if $\tau' = \tau\{\alpha_i/\tau_i\}_{i=1}^n$ (i.e., τ' is obtained by replacing each α_i by a type τ_i in τ) and τ_i is a u-type if α_i is a u-type variable.[7]

Type Closure The *closure* of a type τ, written "Clos τ", is the type scheme $\forall\alpha_1 \ldots \alpha_n.\tau$, where $\{\alpha_1, \ldots, \alpha_n\}$ is the set of all type variables occurring in τ.[8]

Typing Rules

The typing rules for all syntactic categories of ASM-SL are listed in tables in the following pages. Note that there is a typing rule for each construct of the language, such that the typing rule which has to be applied to derive the type of a given phrase is always determined.

Note also that some rules are to be considered as rule schemata. For instance, consider the rule for determining the type of an integer constant in a term (rule TERM_{int} in Table 4.2): there, "$\Sigma, \pi \vdash$ *int_const* : `INT`" stands for all its (infinitely many) instances "$\Sigma, \pi \vdash 0$: `INT`", "$\Sigma, \pi \vdash 1$: `INT`", etc.

Patterns Typing judgments for patterns are formulated in the context of a signature Σ and associate to a (well-typed) pattern p a pair consisting of a type environment π (assigning a type to each variable bound by p) and the type of p. This can be expressed concisely by the notation $\Sigma \vdash p : (\pi, \tau)$, which expresses the "type" of the typing judgments.[9]

Typing rules for all basic forms of patterns are given in Table 4.1. Note that, in the typing rule ($\text{PATT}_{\text{tuple}}$), the map union is always defined, because of the syntactic restriction that no variable occurs twice in a pattern (see Sect. 3.4.1).

Terms For terms t, given a context consisting of a signature Σ and a type environment π (assigning a type to each variable occurring free in t), the type τ of t is derived. That is, typing judgments have the form $\Sigma, \pi \vdash t : \tau$.

Typing rules for all basic forms of terms are given in Table 4.2. For the convenience of the reader, Table 4.3 also includes typing rules for comprehension terms and quantifications, although these are not basic forms and the corresponding typing rules could be derived from the typing rules for basic forms.

[7] This definition is an adapted version of the corresponding definition from [72].

[8] This definition is a special case (*total* closure) of the corresponding definition from [72]: due to the structure of ASM-SL specifications, the more general definition is not needed.

[9] Note that—as opposite to terms and rules—typing judgments for patterns do not depend on a type environment, as all occurrences of variables within a pattern are binding occurrences.

(PATT_{int})	$\dfrac{}{\Sigma \vdash \mathit{int\_const} : (\{\}, \texttt{INT})}$
($\text{PATT}_{\text{float}}$)	$\dfrac{}{\Sigma \vdash \mathit{float\_const} : (\{\}, \texttt{FLOAT})}$
($\text{PATT}_{\text{string}}$)	$\dfrac{}{\Sigma \vdash \mathit{string\_const} : (\{\}, \texttt{STRING})}$
($\text{PATT}_{\text{placeholder}}$)	$\dfrac{}{\Sigma \vdash \_ : (\{\}, \tau)}$ for any type τ
(PATT_{var})	$\dfrac{}{\Sigma \vdash \mathit{var\_id} : (\{\mathit{var\_id} \mapsto \tau\}, \tau)}$ for any type τ
($\text{PATT}_{\text{constr\_var}}$)	$\dfrac{}{\Sigma \vdash \texttt{(}\ \mathit{var\_id}\ \texttt{:}\ \tau\ \texttt{)} : (\{\mathit{var\_id} \mapsto \tau\}, \tau)}$
($\text{PATT}_{\text{tuple}}$)	$\dfrac{\Sigma \vdash p_i : (\pi_i, \tau_i) \qquad (i = 1, \ldots, n)}{\Sigma \vdash \texttt{(}\ p_1\ \texttt{,}\ \ldots\ \texttt{,}\ p_n\ \texttt{)} : (\bigcup_{i=1}^{n} \pi_i,\ \tau_1\ \texttt{*}\ \ldots\ \texttt{*}\ \tau_n)}$
($\text{PATT}_{\text{appl}}$)	$\dfrac{\Sigma(\mathit{fun\_id}) \succ \tau' \to \tau \qquad \Sigma \vdash p : (\pi, \tau')}{\Sigma \vdash \mathit{fun\_id}\ \texttt{(}\ p\ \texttt{)} : (\pi, \tau)}$

Table 4.1: Typing Rules for Patterns — $\Sigma \vdash p : (\pi, \tau)$

Transition Rules Typing of transition rules has basically the same structure as typing of terms. In fact, for the purpose of type checking, rules can be considered as terms of a special type.[10] Thus, to simplify matters, we introduce here a type "$\Box$" of transition rules (which, like type schemes, only exists at the meta-level, but not in the ASM-SL language).

Typing rules for all basic forms of transition rules are given in Table 4.4. It is easy to see that the typing rules ($\text{RULE}_{\text{appl}}$), ($\text{RULE}_{\text{cond}}$) and ($\text{RULE}_{\text{case}}$) are special cases of the corresponding rules for terms. Therefore, in the sequel of this chapter, as for each construct involving transition rules there will be a corresponding term-based construct (lambda-rule/lambda-term, named rule definition/function definition), we will skip the former and only discuss the latter.

Function/Transition Expressions Table 4.5 shows the typing rules for function expressions. The context does not include a type environment, as function expressions always occur at the *top-level* (functions can not be defined locally, e.g., in the scope of a `let` construct).

The typing rule for lambda-abstraction is standard. In addition, there are two typing rules for the special ASM-SL operators `MAP_TO_FUN` and `SET_TO_REL`, which allow extensional function definitions: note that the range of the finite map in a `MAP_TO_FUN` expression must be a u-type, because the resulting function yields *undef* for all arguments outside the (finite) domain of the map. The typing of lambda-rules (which are the only kind of transition expressions) is a special case of the typing of lambda-terms, as discussed above.

[10] However, it is convenient to keep terms and transition rules syntactically separate, as they are very different with respect to dynamic semantics (see Chapter 5).

(TERM_{int})	$\dfrac{}{\Sigma, \pi \vdash \textit{int\_const} : \texttt{INT}}$
($\text{TERM}_{\text{float}}$)	$\dfrac{}{\Sigma, \pi \vdash \textit{float\_const} : \texttt{FLOAT}}$
($\text{TERM}_{\text{string}}$)	$\dfrac{}{\Sigma, \pi \vdash \textit{string\_const} : \texttt{STRING}}$
(TERM_{var})	$\dfrac{\pi(\textit{var\_id}) = \tau}{\Sigma, \pi \vdash \textit{var\_id} : \tau}$
($\text{TERM}_{\text{tuple}}$)	$\dfrac{\Sigma, \pi \vdash t_i : \tau_i \qquad (i = 1, \ldots, n)}{\Sigma, \pi \vdash \texttt{(}\, t_1 \texttt{,} \ldots \texttt{,}\, t_n \,\texttt{)} : \tau_1 \texttt{ * } \ldots \texttt{ * } \tau_n}$
($\text{TERM}_{\text{appl}}$)	$\dfrac{\Sigma(\textit{fun\_id}) \succ \tau' \to \tau \qquad \Sigma, \pi \vdash t : \tau'}{\Sigma, \pi \vdash \textit{fun\_id}\ \texttt{(}\, t \,\texttt{)} : \tau}$
($\text{TERM}_{\text{cond}}$)	$\dfrac{\Sigma, \pi \vdash G : \texttt{BOOL} \qquad \Sigma, \pi \vdash t_T : \tau \qquad \Sigma, \pi \vdash t_F : \tau}{\Sigma, \pi \vdash \texttt{if}\ G\ \texttt{then}\ t_T\ \texttt{else}\ t_F\ \texttt{endif} : \tau}$
($\text{TERM}_{\text{case}}$)	$\dfrac{\Sigma, \pi \vdash t_0 : \tau_0 \quad \Sigma \vdash p : (\pi', \tau_0) \qquad \Sigma, \pi \oplus \pi' \vdash t_1 : \tau \qquad \Sigma, \pi \vdash t_2 : \tau}{\Sigma, \pi \vdash \texttt{case}\ t_0\ \texttt{of}\ p\ \texttt{:}\ t_1\ \texttt{; otherwise}\ t_2\ \texttt{endcase} : \tau}$
($\text{TERM}_{\text{fun\_to\_map}}$)	$\dfrac{\Sigma(\textit{fun\_id}) = \tau' \to \tau \qquad \tau \text{ is a u-type}}{\Sigma, \pi \vdash \texttt{FUN\_TO\_MAP}\ \textit{fun\_id} : \texttt{MAP(}\tau'\texttt{,}\tau\texttt{)}}$

Table 4.2: Typing Rules for Terms — $\Sigma, \pi \vdash t : \tau$

Definitions The typing rules for *type definitions* are completely standard (see, for instance, [72]), thus we omit mentioning them here.

In the typing rules for *function definitions* (shown in the Table 4.6), we have to distinguish between *intensional* and *extensional* definitions. As opposite to extensional definitions, recursion is allowed within intensional definitions, i.e., the function name on the left-hand side of the definition must be included in the context (signature) used for typing the right-hand side. The typing rule for simultaneous function definitions is obviously a straightforward generalization of ($\text{FUNDEF}_{\text{intens}}$). For external functions, the corresponding signature is simply determined by taking the (mandatory) type constraint, as there is no function expression from which the function type could be inferred.

The typing rule for *named function definitions* is similar to ($\text{FUNDEF}_{\text{intens}}$), except that, in its premise, the context is given by the signature Σ alone, as named rule definitions can not be recursive.

Finally, a definition sequence is also a definition. The typing of such definition sequences takes place "sequentially", as shown by the following rule:

(DEFSEQ) $$\frac{\Sigma \vdash D_1 : \Sigma_1 \qquad \Sigma \cup \Sigma_1 \vdash D_2 \ldots D_n : \Sigma_{2\ldots n}}{\Sigma \vdash D_1 \ldots D_n : \Sigma_1 \cup \Sigma_{2\ldots n}}$$

Note that the use of $\cup$ instead of $\oplus$ for merging subsignatures implies that each name can be used only once (i.e., no redefinitions are allowed).[11]

[11] As opposite to binding occurrences of variables within patterns, which "mask" bindings of the same variable which possibly already exist in the given environment.

$$\frac{\Sigma \vdash p : (\pi_p, \tau_p) \qquad \Sigma, \pi \vdash L : \texttt{LIST}(\tau_p) \qquad \Sigma, \pi \oplus \pi_p \vdash G : \texttt{BOOL},\ t : \tau}{\Sigma, \pi \vdash \texttt{[}\ t\ \texttt{|}\ p\ \texttt{in}\ L\ \texttt{with}\ G\ \texttt{]}\ :\ \texttt{LIST}(\tau)}$$

$$\frac{\Sigma \vdash p : (\pi_p, \tau_p) \qquad \Sigma, \pi \vdash A : \texttt{SET}(\tau_p) \qquad \Sigma, \pi \oplus \pi_p \vdash G : \texttt{BOOL},\ t : \tau}{\Sigma, \pi \vdash \{\ t\ \texttt{|}\ p\ \texttt{in}\ A\ \texttt{with}\ G\ \}\ :\ \texttt{SET}(\tau)}$$

$$\frac{\Sigma \vdash p : (\pi_p, \tau_p) \quad \Sigma, \pi \vdash A : \texttt{SET}(\tau_p) \quad \Sigma, \pi \oplus \pi_p \vdash G : \texttt{BOOL},\ t : \tau,\ t' : \tau' \quad (\tau' \text{ u-type})}{\Sigma, \pi \vdash \{\ t\ \texttt{->}\ t'\ \texttt{|}\ p\ \texttt{in}\ A\ \texttt{with}\ G\ \}\ :\ \texttt{MAP}(\tau, \tau')}$$

$$\frac{\Sigma \vdash p : (\pi_p, \tau_p) \qquad \Sigma, \pi \vdash A : \texttt{SET}(\tau_p) \qquad \Sigma, \pi \oplus \pi_p \vdash G : \texttt{BOOL}}{\Sigma, \pi \vdash \texttt{(}\ \texttt{exists/forall}\ p\ \texttt{in}\ A\ \texttt{:}\ G\ \texttt{)}\ :\ \texttt{BOOL}}$$

Table 4.3: Typing Rules for Comprehension Terms (Derived Forms)

Specifications As already mentioned in Sect. 3.9, ASM-SL specifications consist of sequences

$$\mathit{spec} \equiv D_1 \ldots D_n$$

of definitions (of types, functions, and named rules). One or more specifications can be loaded to extend the ASM-SL *initial context* (see Sect. 3.9). For the purpose of type checking only a part of the context is of interest, namely the signature. If the current signature is denoted by Σ_{curr}, then the static semantics of loading a specification *spec* can be expressed operationally by an ASM rule

$$\Sigma_{\text{curr}} := \Sigma_{\text{curr}} \cup \Sigma_\Delta,$$

where Σ_Δ is the signature obtained by typing *spec* in the current signature, i.e., $\Sigma_{\text{curr}} \vdash \mathit{spec} : \Sigma_\Delta$. Note that, initially, $\Sigma_{\text{curr}} = \Sigma_0$, where Σ_0 is the signature containing the primitive ASM-SL types and functions, as listed in Appendix B.

$$(\text{RULE}_{\text{skip}}) \quad \frac{}{\Sigma, \pi \vdash \texttt{skip} : \square}$$

$$(\text{RULE}_{\text{update}}) \quad \frac{\Sigma(\mathit{fun\_id}) = \tau' \to \tau \qquad \Sigma, \pi \vdash t' : \tau' \qquad \Sigma, \pi \vdash t : \tau}{\Sigma, \pi \vdash \mathit{fun\_id}\ \texttt{(}\ t'\ \texttt{)} := t\ :\ \square}$$

$$(\text{RULE}_{\text{block}}) \quad \frac{\Sigma, \pi \vdash R_i : \square \qquad (i = 1, \ldots, n)}{\Sigma, \pi \vdash R_1\ \ldots\ R_n\ :\ \square}$$

$$(\text{RULE}_{\text{appl}}) \quad \frac{\Sigma(\mathit{rule\_id}) \succ \tau \to \square \qquad \Sigma, \pi \vdash t : \tau}{\Sigma, \pi \vdash \mathit{rule\_id}\ \texttt{(}\ t\ \texttt{)}\ :\ \square}$$

$$(\text{RULE}_{\text{cond}}) \quad \frac{\Sigma, \pi \vdash G : \texttt{BOOL} \qquad \Sigma, \pi \vdash R_T : \square \qquad \Sigma, \pi \vdash R_F : \square}{\Sigma, \pi \vdash \texttt{if}\ G\ \texttt{then}\ R_T\ \texttt{else}\ R_F\ \texttt{endif}\ :\ \square}$$

$$(\text{RULE}_{\text{case}}) \quad \frac{\begin{array}{l}\Sigma, \pi \vdash t : \tau \\ \Sigma \vdash p : (\pi', \tau) \qquad \Sigma, \pi \oplus \pi' \vdash R_1 : \square \qquad \Sigma, \pi \vdash R_2 : \square\end{array}}{\Sigma, \pi \vdash \texttt{case}\ t\ \texttt{of}\ p\ \texttt{:}\ R_1\ \texttt{;}\ \texttt{otherwise}\ R_2\ \texttt{endcase}\ :\ \square}$$

$$(\text{RULE}_{\text{choose/do-forall}}) \quad \frac{\Sigma \vdash p : (\pi_p, \tau_p) \qquad \Sigma, \pi \vdash A : \texttt{SET(}\tau_p\texttt{)} \qquad \Sigma, \pi \oplus \pi_p \vdash G : \texttt{BOOL},\ R : \square}{\Sigma, \pi \vdash \texttt{choose/do forall}\ p\ \texttt{in}\ A\ \texttt{with}\ G\ \ R\ \ \texttt{endchoose/enddo}\ :\ \square}$$

Table 4.4: Typing Rules for Transition Rules — $\Sigma, \pi \vdash R : \square$

$$(\text{FEXPR}_{\text{lambda}}) \quad \frac{\Sigma \vdash p : (\pi_p, \tau_p) \qquad \Sigma, \pi_p \vdash t : \tau_t}{\Sigma \vdash \texttt{fn (}\ p\ \texttt{) ->}\ t\ :\ \tau_p \to \tau_t}$$

$$(\text{FEXPR}_{\text{map\_to\_fun}}) \quad \frac{\Sigma, \{\,\} \vdash t : \texttt{MAP(}\tau'\texttt{,}\tau\texttt{)} \qquad \tau \text{ is a u-type}}{\Sigma \vdash \texttt{MAP\_TO\_FUN}\ t\ :\ \tau' \to \tau}$$

$$(\text{FEXPR}_{\text{set\_to\_rel}}) \quad \frac{\Sigma, \{\,\} \vdash t : \texttt{SET(}\tau\texttt{)}}{\Sigma \vdash \texttt{SET\_TO\_REL}\ t\ :\ \tau \to \texttt{BOOL}}$$

Table 4.5: Typing Rules for Function Expressions — $\Sigma \vdash \mathit{fexpr} : \tau_1 \to \tau_2$

$$(\text{FUNDEF}_{\text{intens}}) \quad \frac{\Sigma \oplus \{\, f \mapsto (\tau' \to \tau)\,\} \vdash \texttt{fn (}p\texttt{) ->}\, t\ :\ \tau' \to \tau}{\Sigma \vdash \texttt{static/derived function}\ f\,[\texttt{:}\tau' \to \tau] \texttt{ == fn (}p\texttt{) ->}\, t\ :\ \{\, f \mapsto \text{Clos}\ (\tau' \to \tau)\,\}}$$

$$(\text{FUNDEF}_{\text{extens}}) \quad \text{For } \mathit{fexpr} \equiv \texttt{MAP\_TO\_FUN}\ M \text{ or } \mathit{fexpr} \equiv \texttt{SET\_TO\_REL}\ A\text{:} \quad \frac{\Sigma \vdash \mathit{fexpr}\ :\ \tau' \to \tau \qquad \tau' \to \tau \text{ is monomorphic}}{\Sigma \vdash \texttt{static/dynamic function}\ f\,[\texttt{:}\tau' \to \tau]\ \texttt{==/initially}\ \mathit{fexpr} : \{f \mapsto (\tau' \to \tau)\}}$$

$$(\text{FUNDEF}_{\text{extern}}) \quad \frac{\tau' \to \tau \text{ is monomorphic}}{\Sigma \vdash \texttt{external function}\ f\ :\ \tau' \to \tau\ :\ \{f \mapsto (\tau' \to \tau)\}}$$

Table 4.6: Typing Rules for Function Definitions — $\Sigma \vdash D : \Sigma'$

Chapter 5

Dynamic Semantics

In this chapter, we present the dynamic semantics of ASM-SL, corresponding to the *evaluation* phase which follows the type-checking.[1] After introducing the necessary semantic domains (Sect. 5.1), we define the semantics of each syntactic category by specifying the mapping of each syntactic construct into the appropriate semantic domain (Sect. 5.2).

In order to conveniently define the type of the semantic mappings, we also introduce domains for each syntactic category of ASM-SL (*syntactic domains*), with the same name as the corresponding non-terminal of the ASM-SL grammar, but in "small caps" font (e.g., PATT for patterns, TERM for terms, etc.).

5.1 Semantic Domains

5.1.1 Values

All values of ASM-SL are elements of a *base set* $\mathcal{U}$ (also called universe of discourse, or superuniverse). The interpretation of ground basic types is given by subsets of $\mathcal{U}$ (*carrier sets*). The base set $\mathcal{U}$ is defined as

$$\begin{aligned}\mathcal{U} \;\cong\; & \mathbf{BOOL} + \mathbf{INT} + \mathbf{FLOAT} + \mathbf{STRING} + \{\,\mathbf{undef}\,\} \\ & + \mathcal{U}^* + \mathrm{Fin}\,\mathcal{U} + \mathcal{U} \xrightarrow{\text{fin}} (\mathcal{U} \setminus \{\,\mathbf{undef}\,\}) \\ & + \bigcup_{C \in \mathcal{C}} \{\,\langle C(x)\rangle \mid x \in \mathcal{U}\,\}\end{aligned}$$

where the + operator stands for disjoint union, and:

- A^* denotes the set of finite sequences (tuples) over A, which includes the empty sequence () and where sequences of length 1 are identical with the corresponding elements of A;

[1] The dynamic semantics relies on the assumption that ASM-SL phrases are well-formed and well-typed, i.e., their evaluation is performed only after their successful type-checking.

Built-in types:

$$
\begin{aligned}
\mathcal{T}[\![\,\texttt{BOOL}\,]\!] &= \mathbf{BOOL} = \{\,\mathbf{false}, \mathbf{true}\,\} \\
\mathcal{T}[\![\,\texttt{INT}\,]\!] &= \mathbf{INT} \cup \{\,\mathbf{undef}\,\} \\
\mathcal{T}[\![\,\texttt{FLOAT}\,]\!] &= \mathbf{FLOAT} \cup \{\,\mathbf{undef}\,\} \\
\mathcal{T}[\![\,\texttt{STRING}\,]\!] &= \mathbf{STRING} \cup \{\,\mathbf{undef}\,\} \\
\mathcal{T}[\![\,\texttt{()}\,]\!] &= \mathcal{U}^0 = \{\,()\,\} \\
\mathcal{T}[\![\,\tau_1\,\texttt{*}\,\ldots\,\texttt{*}\,\tau_n\,]\!] &= \{\,(x_1,\ldots,x_n) \mid x_i \in \mathcal{T}[\![\tau_i]\!],\, i \leq 1 \leq n\,\} \subseteq \mathcal{U}^n \\
&\qquad (\text{for } n \geq 2) \\
\mathcal{T}[\![\,\texttt{SET(}\tau\texttt{)}\,]\!] &= (\text{Fin}\ \mathcal{T}[\![\tau]\!]) \cup \{\,\mathbf{undef}\,\} \\
\mathcal{T}[\![\,\texttt{MAP(}\tau\texttt{,}\tau'\texttt{)}\,]\!] &= (\mathcal{T}[\![\tau]\!] \xrightarrow{\text{fin}} \mathcal{T}[\![\tau']\!]) \cup \{\,\mathbf{undef}\,\}
\end{aligned}
$$

User-defined free types:

$$\mathcal{T}[\![\,T(\tau_1,\ldots,\tau_n)\,]\!] = \mathbf{T}(\mathcal{T}[\![\tau_1]\!],\ldots,\mathcal{T}[\![\tau_n]\!])$$

Table 5.1: Interpretation of Ground Basic Types

- $\mathcal{C}$ is the collection of all possible *constructor names*, which includes `undef`, the list constructors `::` and `nil` and all constructor names possibly declared within definitions of free types.[2]

In ASM-SL, all states (i.e., any state of any run of any ASM-SL specification) have $\mathcal{U}$—as defined above—as their universe of discourse. However, due to the restrictions imposed by the type system, not all elements of $\mathcal{U}$ are reachable, i.e., can be denoted by some term: for instance, while all finite sets with elements from $\mathcal{U}$ are elements of $\mathcal{U}$ themselves, only finite sets with elements of the same type are reachable elements.

Note that all issues related to typing have been handled in the previous chapter on type system, such that there is no need to refer to types in the discussion of dynamic semantics. However, for the convenience of the reader, we present the intended interpretation of (ground) basic types in Table 5.1, defined by a mapping $\mathcal{T} : \textsc{Type} \to \mathcal{P}(\mathcal{U})$, which associates to each ground basic type its corresponding carrier set. (In the equation for user-defined free types, $\mathbf{T} : (\mathcal{P}(\mathcal{U}))^n \to \mathcal{P}(\mathcal{U})$ denotes the interpretation of type constructor T.[3])

As the base set $\mathcal{U}$ includes the set of all *values* admissible for ASM-SL terms, we also use the name *VALUE* as a synonym for $\mathcal{U}$.

5.1.2 States

As the base set is fixed (see above), the notion of state in ASM-SL slightly differs from the corresponding one in [45, 46]. In particular, the interpretation

[2] `True` and `false` are also constructors, but are considered as elements of the carrier set **BOOL**.

[3] The interpretation of T, in turn, depends on the names and types of the value constructors specified in the type definition for T (we omit the details here).

of function names is sufficient to identify a state (while the base set is not explicitly mentioned, as it is always $\mathcal{U}$). In our notation, this can be expressed by defining the semantic domain of states as follows:

$$STATE = \textsc{FunId} \xrightarrow{\text{fin}} (VALUE \to VALUE)$$

Note that the set of functions making up the states consists of the primitive ASM-SL functions, as listed in Appendix B, and of those introduced by the user by means of ASM-SL function definitions.

5.1.3 Environments

An environment is a *finite map* containing bindings which associate variables to their corresponding values:

$$ENV = \textsc{VarId} \xrightarrow{\text{fin}} VALUE$$

Environments are denoted by the letter ρ in the sequel of this section.

5.1.4 Locations and Updates

According to the definitions introduced in Sect. 1.1, the notions of *location*, *update* and *update set* are represented by the following semantic domains:

$$\begin{aligned} LOCATION &= \textsc{FunId} \times VALUE \\ UPDATE &= LOCATION \times VALUE \\ UPDATE\_SET &= \text{Fin } UPDATE \end{aligned}$$

Note that it is not necessary to define locations as $\textsc{FunId} \times VALUE^*$ (which would be more close to the definition of location given in Sect. 1.1.3), as the set $\mathcal{U}^*$ of all tuples of elements of $\mathcal{U}$ is, by definition, a subset of $\mathcal{U}$.

5.2 Semantic Mappings

The semantic mappings are defined for the different syntactic domains in a compositional style: for better readability, the language constructs whose semantics is to be defined are enclosed in $[\![\cdot]\!]$, followed by the other arguments of the semantic mapping. For instance, we define *evaluation* of terms (in a given state and environment) by a function $\mathcal{E} : \textsc{Term} \to STATE \times ENV \to VALUE$, and, for a given term t, state S, and environment ρ, write $\mathcal{E}[\![t]\!](S, \rho)$.

5.2.1 Patterns

The semantics of pattern matching is formalized by the mapping

$$\mathcal{M} : \textsc{Patt} \to VALUE \to ENV \cup \{\text{FAIL}\},$$

(where $\mathcal{M}$ stands for *match*), defined in Table 5.2. $\mathcal{M}$ yields an environment binding the pattern variables to appropriate values, if the pattern and the value match, or the special result FAIL, if they do not match.

$$\mathcal{M}[\![\ \mathit{int\_const}\ ]\!](x) = \begin{cases} \{\,\} & \text{if } x = \textbf{int\_const} \\ \text{FAIL} & \text{otherwise} \end{cases}$$

$$\mathcal{M}[\![\ \mathit{float\_const}\ ]\!](x) = \begin{cases} \{\,\} & \text{if } x = \textbf{float\_const} \\ \text{FAIL} & \text{otherwise} \end{cases}$$

$$\mathcal{M}[\![\ \mathit{string\_const}\ ]\!](x) = \begin{cases} \{\,\} & \text{if } x = \textbf{string\_const} \\ \text{FAIL} & \text{otherwise} \end{cases}$$

$$\mathcal{M}[\![\ \_\ ]\!](x) = \{\,\} \quad \text{for any } x \in \mathcal{U}$$

$$\mathcal{M}[\![\ \mathit{var\_id}\ ]\!](x) = \{\ \mathit{var\_id} \mapsto x\ \} \quad \text{for any } x \in \mathcal{U}$$

$$\mathcal{M}[\![\ (p_1, \ldots, p_n)\ ]\!](x) = \begin{cases} \bigcup_{i=1}^{n} \mathcal{M}[\![\ p_i\ ]\!](x_i) & \text{if } x = (x_1, \ldots, x_n) \in \mathcal{U}^n \text{ and} \\ & \quad \mathcal{M}[\![\ p_i\ ]\!](x_i) \neq \text{FAIL for all } i \in \{1, \ldots, n\} \\ \text{FAIL} & \text{otherwise} \end{cases}$$

$$\mathcal{M}[\![\ \mathit{fun\_id}\ (\ p\ )\ ]\!](x) = \begin{cases} \mathcal{M}[\![\ p\ ]\!](x') & \text{if } x = C(x') \text{ and } C \equiv \mathit{fun\_id} \\ \text{FAIL} & \text{otherwise} \end{cases}$$

Table 5.2: Semantics of Pattern Matching

Examples

The semantic mapping $\mathcal{M}$, applied to the example patterns of Chapter 3 and to some values of appropriate type, produces the following results:

$$\begin{aligned} \mathcal{M}[\![\ \texttt{true}\ ]\!](\,\textbf{true}\,) &= \{\,\} \\ \mathcal{M}[\![\ \texttt{true}\ ]\!](\,\textbf{false}\,) &= \text{FAIL} \\ \mathcal{M}[\![\ \texttt{[]}\ ]\!](\,[]\,) &= \{\,\} \\ \mathcal{M}[\![\ \texttt{[]}\ ]\!](\,[1,2,3]\,) &= \text{FAIL} \\ \mathcal{M}[\![\ \texttt{x :: y}\ ]\!](\,[]\,) &= \text{FAIL} \\ \mathcal{M}[\![\ \texttt{x :: y}\ ]\!](\,[1,2,3]\,) &= \{\,\texttt{x} \mapsto 1,\ \texttt{y} \mapsto [2,3]\,\} \\ \mathcal{M}[\![\ \texttt{(x, 1, [z])}\ ]\!](\,(0,1,\textbf{false})\,) &= \text{FAIL} \\ \mathcal{M}[\![\ \texttt{(x, 1, [z])}\ ]\!](\,(0,1,[\textbf{false}])\,) &= \{\,\texttt{x} \mapsto 0,\ \texttt{z} \mapsto \textbf{false}\,\} \end{aligned}$$

5.2.2 Terms

The semantics of terms is formalized by the evaluation mapping

$$\mathcal{E} : \textsc{Term} \to \mathit{STATE} \times \mathit{ENV} \to \mathit{VALUE},$$

defined in Table 5.3. The notation $(S, \rho) \models G$ is used as an abbreviation for $\mathcal{E}[\![\ G\ ]\!](S, \rho) = \textbf{true}$, if G is a boolean term.

Examples

As the semantics of basic terms is quite straightforward, we show here only examples of variable-binding terms, in particular `case`-terms and comprehension terms.

- Example of `case`-term:

  $t \equiv$
  ```
  case L of
    x :: xs : xs ;
    []      : undef
  endif
  ```

  Assume that S is a state where the nullary function name L is interpreted as $\mathbf{L}_S = [1, 2, 3]$. The value $\mathbf{L}_S$ matches the pattern "`x :: xs`" and, according to the definition in Table 5.3, $\rho' = \{\, \mathtt{x} \mapsto 1,\ \mathtt{xs} \mapsto [2, 3] \,\}$. Thus, $\mathcal{E}[\![\, t \,]\!](S, \rho) = \mathcal{E}[\![\, \mathtt{xs} \,]\!](S, \rho \oplus \rho') = (\rho \oplus \rho')(\mathtt{xs}) = [2, 3]$.

- Example of list comprehension term (see Sect. 3.4.2, page 41):

  $t \equiv$ `[ x + y | (x, y) in list_of_pairs with x < y ]`

  Assume that S is a state where the nullary function name *list_of_pairs* is interpreted as $\mathbf{list\_of\_pairs}_S = [(1, 4), (2, 1), (3, 10)]$. The evaluation of t corresponds to evaluating a term "`F(list_of_pairs)`", where F is defined as in Sect. 3.4.2, page 41. Then:

  $$\begin{aligned}
  & \mathcal{E}[\![\, t \,]\!](S, \rho) \ = \\
  = \ & \mathcal{E}[\![\, \mathtt{F(list_of_pairs)} \,]\!](S, \rho) \ = \\
  = \ & \mathbf{F}_S([(1, 4), (2, 1), (3, 10)]) \ = \\
  = \ & 5 :: \mathbf{F}_S([(2, 1), (3, 10)]) \ = \\
  = \ & 5 :: \mathbf{F}_S([(3, 10)]) \ = \\
  = \ & 5 :: 13 :: \mathbf{F}_S([]) \ = \\
  = \ & 5 :: 13 :: [] \ = \\
  = \ & [5, 13].
  \end{aligned}$$

5.2.3 Transition Rules

Following [46], we first define the semantics of deterministic transition rules, then we refine this semantics to deal with nondeterministic rules (`choose` rules). The deterministic semantics is formalized by a mapping from rules to update sets:[4]

$$\Delta : \textsc{Rule} \to \mathit{STATE} \times \mathit{ENV} \to \mathit{RULE\_DENOTATION}$$

where $\mathit{RULE\_DENOTATION} \equiv \mathit{UPDATE\_SET} \cup \{\text{ERROR}\}$. Note that we consider only consistent update sets as proper update sets, otherwise the evaluation of a rule results in an error.[5] This is reflected by the following

[4] Note that defining the semantics of rules by means of *update sets* (as in [45]) instead of *actions* (as in [46]) is adequate here, as ASM-SL does not include `import` rules: see [46] for the definition of actions and of the corresponding sum operator.

[5] *Consistent* update sets, according to [45], are those which do not contain conflicting updates (see also Sect. 1.1.4).

$$
\begin{array}{rcl}
\mathcal{E}[\![\ \mathit{int\_const}\ ]\!](S,\rho) & = & \mathbf{int\_const} \\
\mathcal{E}[\![\ \mathit{float\_const}\ ]\!](S,\rho) & = & \mathbf{float\_const} \\
\mathcal{E}[\![\ \mathit{string\_const}\ ]\!](S,\rho) & = & \mathbf{string\_const} \\
\mathcal{E}[\![\ \mathit{var\_id}\ ]\!](S,\rho) & = & \rho(\mathit{var\_id}) \\
\mathcal{E}[\![\ \texttt{(}\ t_1\ \texttt{,}\ \ldots\ \texttt{,}\ t_n\ \texttt{)}\ ]\!](S,\rho) & = & (\mathcal{E}[\![\ t_1\ ]\!](S,\rho), \ldots, \mathcal{E}[\![\ t_n\ ]\!](S,\rho)) \\
\mathcal{E}[\![\ \mathit{fun\_id}\ \texttt{(}\ t\ \texttt{)}\ ]\!](S,\rho) & = & \mathbf{fun\_id}_S(\mathcal{E}[\![\ t\ ]\!](S,\rho)) \\
\mathcal{E}[\![\ \texttt{if}\ G\ \texttt{then}\ t_T\ \texttt{else}\ t_F\ \texttt{endif}\ ]\!](S,\rho) & = & \begin{cases} \mathcal{E}[\![\ t_T\ ]\!](S,\rho) & \text{if } (S,\rho) \models G \\ \mathcal{E}[\![\ t_F\ ]\!](S,\rho) & \text{otherwise} \end{cases}
\end{array}
$$

$$
\mathcal{E}[\![\ \texttt{case}\ t_0\ \texttt{of}\ p\texttt{:}t_1\texttt{;}\ \texttt{otherwise}\ t_2\ \texttt{endcase}\ ]\!](S,\rho) =
$$

$$
= \begin{cases} \mathcal{E}[\![\ t_1\ ]\!](S,\rho \oplus \rho') & \text{if } \rho' \neq \text{FAIL} \\ \mathcal{E}[\![\ t_2\ ]\!](S,\rho) & \text{otherwise} \end{cases}
$$

where $\rho' = \mathcal{M}[\![\ p\ ]\!](\mathcal{E}[\![\ t_0\ ]\!](S,\rho))$

Table 5.3: Semantics of Terms

definition of sum of update sets (synchronous-parallel composition), used in Table 5.4 to define the semantics of block and `do forall` rules:

$$\Delta_1 + \Delta_2 = \begin{cases} \Delta_1 \cup \Delta_2 & \text{if } \Delta_i \neq \text{ERROR } (i = 1,2) \text{ and } \Delta_1 \cup \Delta_2 \text{ is consistent} \\ \text{ERROR} & \text{otherwise.} \end{cases}$$

The nondeterministic semantics is expressed in terms of update set families (i.e., non-empty finite sets of update sets): thus, we introduce a new mapping

$$\boldsymbol{\Delta} : \textsc{Rule} \rightarrow \mathit{STATE} \times \mathit{ENV} \rightarrow \mathit{RULE\_DENOTATION}$$

where *RULE_DENOTATION* is refined to Fin (*UPDATE_SET* $\cup$ {ERROR}). The sum operator is extended to update set families as follows:

$$\boldsymbol{\Delta}_1 + \boldsymbol{\Delta}_2 \quad = \quad \{\, \Delta_i + \Delta_j \mid \Delta_i \in \boldsymbol{\Delta}_1,\ \Delta_j \in \boldsymbol{\Delta}_2 \,\}$$

The nondeterministic semantics obviously generalizes the deterministic one, in the sense that $\boldsymbol{\Delta}[\![\ R\ ]\!](S,\rho) = \{\, \Delta[\![\ R\ ]\!](S,\rho) \,\}$ for any deterministic rule R (i.e., for any rule R not containing any `choose` as a subrule).

The definitions of the Δ and $\boldsymbol{\Delta}$ mappings are shown in Table 5.4. For `choose` and `do forall` rules, which share the same syntactic structure, the following abbreviation is used:

$$\begin{array}{rl} \text{range}_{S,\rho}(p, A, G) \ \equiv \ \{\, x \mid x \in \mathcal{E}[\![\ A\ ]\!](S,\rho) & \text{with } \mathcal{M}[\![\ p\ ]\!](x) \neq \text{FAIL}, \\ & \text{such that } (S, \rho \oplus \mathcal{M}[\![\ p\ ]\!](x)) \models G \,\} \end{array}$$

Note that, beyond the need for a special treatment of empty ranges in `choose` rules, the only difference between the semantics of `choose` and `do forall` is the composition operation applied (union for `choose`, sum for `do forall`).

Finally observe that, in the definition of the semantics of named rule applications ("*rule_id* `(` *t* `)`"), the interpretation of a rule name *rule_id* in a given

state S (which is a function from *VALUE* to *RULE_DENOTATION*) is denoted by $\mathbf{rule\_id}_S$, in analogy the interpretation of function names (for details on the precise meaning of $\mathbf{rule\_id}_S$, see Sect. 5.2.4 below, under "Transition Expressions").

In the sequel of this chapter, whenever the distinction between the deterministic and the nondeterministic case is not important, we denote the semantics of transition rules simply by Δ (which can be replaced by $\mathbf{\Delta}$ if preferred, if the interpretation of *RULE_DENOTATION* is also changed accordingly).

Example: Deterministic Semantics

Consider the rule

$R \equiv$
```
do forall x in {0,1}
   do forall y in {0,1}
      f (x, y) := 2*x + y
   enddo
enddo
```

For conciseness, we refer to the subrule "`f (x, y) := 2*x + y`" as R'.

Then, according to the deterministic semantics, as defined in Table 5.4, the update set denoted by this rule is:

$$
\begin{aligned}
& \Delta[\![\, R \,]\!](S, \emptyset) \;= \\
= \;& \textstyle\sum_{x \in \{0,1\}} \sum_{y \in \{0,1\}} \Delta\,[\![\, R' \,]\!]\,(S, \{\mathtt{x} \mapsto x,\, \mathtt{y} \mapsto y\}) \\
= \;& (\,\Delta\,[\![\, R' \,]\!]\,(S, \{\mathtt{x} \mapsto 0,\, \mathtt{y} \mapsto 0\}) + \Delta\,[\![\, R' \,]\!]\,(S, \{\mathtt{x} \mapsto 0,\, \mathtt{y} \mapsto 1\})\,) \;\; + \\
+ \;& (\,\Delta\,[\![\, R' \,]\!]\,(S, \{\mathtt{x} \mapsto 1,\, \mathtt{y} \mapsto 0\}) + \Delta\,[\![\, R' \,]\!]\,(S, \{\mathtt{x} \mapsto 1,\, \mathtt{y} \mapsto 1\})\,) \\
= \;& (\,\{\,((\mathtt{f}, (0,0)),\, 0)\,\} + \{\,((\mathtt{f}, (0,1)),\, 1)\,\}\,) \;\; + \\
+ \;& (\,\{\,((\mathtt{f}, (1,0)),\, 2)\,\} + \{\,((\mathtt{f}, (1,1)),\, 3)\,\}\,) \\
= \;& \{\,((\mathtt{f}, (0,0)),\, 0)\,\} + \{\,((\mathtt{f}, (0,1)),\, 1)\,\} + \{\,((\mathtt{f}, (1,0)),\, 2)\,\} + \{\,((\mathtt{f}, (1,1)),\, 3)\,\} \\
= \;& \{\,((\mathtt{f}, (0,0)),\, 0),\, ((\mathtt{f}, (0,1)),\, 1),\, ((\mathtt{f}, (1,0)),\, 2),\, ((\mathtt{f}, (1,1)),\, 3)\,\}
\end{aligned}
$$

Example: Nondeterministic Semantics

Consider the rule

$R \equiv$
```
do forall x in {0,1}
   choose y in {0,1}
      do forall z in {0,1}
         f (x, y, z) := 4*x + 2*y + z
      enddo
   endchoose
enddo
```

Again, we refer to the inner subrule "`f (x, y, z) := 4*x + 2*y + z`" as R'.

According to the nondeterministic semantics, as defined in Table 5.4, the semantics of this rule is an update set family, determined as follows:

$$
\begin{aligned}
& \boldsymbol{\Delta}[\![\, R \,]\!](S,\emptyset) \;= \\
= \;& \textstyle\sum_{x\in\{0,1\}} \bigcup_{y\in\{0,1\}} \sum_{z\in\{0,1\}} \Delta\,[\![\, R' \,]\!]\,(S, \{\mathtt{x}\mapsto x,\, \mathtt{y}\mapsto y,\, \mathtt{z}\mapsto z\}) \\
= \;& (\,(\,\boldsymbol{\Delta}[\![R']\!](S,\{\mathtt{x}\mapsto 0,\mathtt{y}\mapsto 0,\mathtt{z}\mapsto 0\}) + \boldsymbol{\Delta}[\![R']\!](S,\{\mathtt{x}\mapsto 0,\mathtt{y}\mapsto 0,\mathtt{z}\mapsto 1\})\,) \cup \\
& \;\;(\,\boldsymbol{\Delta}[\![R']\!](S,\{\mathtt{x}\mapsto 0,\mathtt{y}\mapsto 1,\mathtt{z}\mapsto 0\}) + \boldsymbol{\Delta}[\![R']\!](S,\{\mathtt{x}\mapsto 0,\mathtt{y}\mapsto 1,\mathtt{z}\mapsto 1\})\,)\,) + \\
& (\,(\,\boldsymbol{\Delta}[\![R']\!](S,\{\mathtt{x}\mapsto 1,\mathtt{y}\mapsto 0,\mathtt{z}\mapsto 0\}) + \boldsymbol{\Delta}[\![R']\!](S,\{\mathtt{x}\mapsto 1,\mathtt{y}\mapsto 0,\mathtt{z}\mapsto 1\})\,) \cup \\
& \;\;(\,\boldsymbol{\Delta}[\![R']\!](S,\{\mathtt{x}\mapsto 1,\mathtt{y}\mapsto 1,\mathtt{z}\mapsto 0\}) + \boldsymbol{\Delta}[\![R']\!](S,\{\mathtt{x}\mapsto 1,\mathtt{y}\mapsto 1,\mathtt{z}\mapsto 1\})\,)\,) \\
= \;& (\,(\,\{\,\{\,((\mathtt{f},(0,0,0)),\,0\,)\,\}\,\} + \{\,\{\,((\mathtt{f},(0,0,1)),\,1\,)\,\}\,\}\,) \cup \\
& \;\;(\,\{\,\{\,((\mathtt{f},(0,1,0)),\,2\,)\,\}\,\} + \{\,\{\,((\mathtt{f},(0,1,1)),\,3\,)\,\}\,\}\,)\,) + \\
& (\,(\,\{\,\{\,((\mathtt{f},(1,0,0)),\,4\,)\,\}\,\} + \{\,\{\,((\mathtt{f},(1,0,1)),\,5\,)\,\}\,\}\,) \cup \\
& \;\;(\,\{\,\{\,((\mathtt{f},(1,1,0)),\,6\,)\,\}\,\} + \{\,\{\,((\mathtt{f},(1,1,1)),\,7\,)\,\}\,\}\,)\,) \\
= \;& (\,\{\,\{\,((\mathtt{f},(0,0,0)),\,0\,),\,((\mathtt{f},(0,0,1)),\,1\,)\,\}\,\} \cup \\
& \;\;\{\,\{\,((\mathtt{f},(0,1,0)),\,2\,),\,((\mathtt{f},(0,1,1)),\,3\,)\,\}\,\}\,) + \\
& (\,\{\,\{\,((\mathtt{f},(1,0,0)),\,4\,),\,((\mathtt{f},(1,0,1)),\,5\,)\,\}\,\} \cup \\
& \;\;\{\,\{\,((\mathtt{f},(1,1,0)),\,6\,),\,((\mathtt{f},(1,1,1)),\,7\,)\,\}\,\}\,) \\
= \;& \{\,\{\,((\mathtt{f},(0,0,0)),0),((\mathtt{f},(0,0,1)),1)\,\},\,\{\,((\mathtt{f},(0,1,0)),2),((\mathtt{f},(0,1,1)),3)\,\}\,\} + \\
& \{\,\{\,((\mathtt{f},(1,0,0)),4),((\mathtt{f},(1,0,1)),5)\,\},\,\{\,((\mathtt{f},(1,1,0)),6),((\mathtt{f},(1,1,1)),7)\,\}\,\} \\
= \;& \{\,\{\,((\mathtt{f},(0,0,0)),0),((\mathtt{f},(0,0,1)),1),((\mathtt{f},(1,0,0)),4),((\mathtt{f},(1,0,1)),5)\,\}, \\
& \;\;\{\,((\mathtt{f},(0,0,0)),0),((\mathtt{f},(0,0,1)),1),((\mathtt{f},(1,1,0)),6),((\mathtt{f},(1,1,1)),7)\,\}, \\
& \;\;\{\,((\mathtt{f},(0,1,0)),2),((\mathtt{f},(0,1,1)),3),((\mathtt{f},(1,0,0)),4),((\mathtt{f},(1,0,1)),5)\,\}, \\
& \;\;\{\,((\mathtt{f},(0,1,0)),2),((\mathtt{f},(0,1,1)),3),((\mathtt{f},(1,1,0)),6),((\mathtt{f},(1,1,1)),7)\,\}\,\}
\end{aligned}
$$

This means that executing the rule R corresponds to firing an update set chosen non-deterministically among the four in the update set family $\boldsymbol{\Delta}[\![\, R \,]\!](S,\emptyset)$.

5.2.4 Definitions

The semantics of definitions is discussed here only very shortly, as it is for a large part completely standard. In particular, the semantics of type and function definitions is essentially the same as in functional programming languages (in particular, in a purely functional subset of ML). We define in more detail and more formally only those forms of definitions which are peculiar features of ASM-SL (extensional function definitions, named rules).

Definitions (in General) In general, definitions are phrases of the form *id* == *rhs* (whereas derived forms may present parameters on the left-hand side). The semantics is very simple: the evaluation of a definition simply binds the identifier *id* to the semantic object denoted by the right-hand side *rhs*, such that the corresponding information becomes available in the current context (see Sect. 3.9).[6] The difference between the different kinds of definitions depends only on the nature of the right-hand side.

[6]Note that there are no semantic issues related to the choice whether a static or a dynamic scoping rule is applied in a sequence of definition, because such a sequence can never contain two definitions of the same identifier (see typing rule DEFSEQ in Sect. 4.2).

In the case of types, the semantics is the same as in ML, thus we skip a discussion of type definitions. For functions there are some particularities, while named rules are a special ASM feature: thus, the semantics of function expressions and transition expressions (which occur on the right-hand side of definitions of functions and named rules, respectively, see Sects. 3.6 and 3.7) are briefly discussed below.

Function Expressions Function expressions occur on the right-hand side of function definitions. We distinguish between *intensional* and *extensional* definitions of functions.

- Intensional definitions are specified by λ-abstraction. In this case, the right-hand side is a function expression of the form "`fn (` p `) ->` t", with the obvious meaning.

- Extensional definitions are specified by the special ASM-SL operators `MAP_TO_FUN` and `SET_TO_REL`. The function expression "`MAP_TO_FUN` M" denotes a function whose graph is specified by the finite map M (and interpreted as **undef** outside the domain of M), while "`SET_TO_REL` A" denotes a relation (i.e., a boolean function) interpreted as **true** on all elements of the finite sets A and as **false** elsewhere.[7]

The semantics of function expressions can be formalized by the mapping

$$\mathcal{FE} \quad : \quad \text{FExpr} \rightarrow \mathit{STATE} \rightarrow (\mathit{VALUE} \rightarrow \mathit{VALUE}),$$

which is defined as follows:

$$\begin{array}{lcl} \mathcal{FE}[\![\,\texttt{fn (}\,p\,\texttt{) -> }\,t\,]\!](S) & = & \lambda x.\mathcal{E}[\![\,t\,]\!](S,\,\mathcal{M}[\![\,p\,]\!](x)) \\ \mathcal{FE}[\![\,\texttt{MAP\_TO\_FUN }\,M\,]\!](S) & = & \lambda x.\mathbf{apply}(\mathcal{E}[\![\,t\,]\!](S,\{\}),x) \\ \mathcal{FE}[\![\,\texttt{SET\_TO\_REL }\,A\,]\!](S) & = & \lambda x.\mathbf{member}(x,\mathcal{E}[\![\,t\,]\!](S,\{\})) \end{array}$$

(where *apply* and *member* are the names of two primitive ASM-SL functions, see Appendix B).

If f is a static or derived function name bound to a function expression *fexpr* in the current context (by a function definition of the form "f `==` *fexpr*"), then its interpretation $\mathbf{f}_S$ in a given state S corresponds to $\mathcal{FE}[\![\,\mathit{fexpr}\,]\!](S)$.[8]

If f is the name of a dynamic function, initialized through a function definition of the form "f `initially` *fexpr*", then its interpretation $\mathbf{f}_{S_0}$ in the initial state S_0 is given by $\mathcal{FE}[\![\,\mathit{fexpr}\,]\!](S_0)$ (while $\mathbf{f}_{S_i}$ for $i > 0$ obviously depends on $\mathbf{f}_{S_0}$ and on the sequence of transitions performed during the given run.)

[7] Extensional definitions are mainly intended for the initialization of dynamic functions (i.e., to define their interpretation in the initial state), but can also be used for the definition of static functions.

[8] Actually, the situation is slightly more complicated in the case of recursive definitions (recursion is allowed in intensional definitions). However, as their semantics in ASM-SL is exactly the same as in functional languages, we skip formal details here (see, for instance, [43], for a precise definition of the semantics of recursive definitions).

Transition Expressions Similar to function expressions, which occur on the right-hand side of function definitions, transition expressions occur on the right-hand side of named rule definitions. There is only one form of transition expression, namely the so-called λ-rule "`tn (` p `) ->` t", which allows to define parameterized rules (the analogies with the λ-term "`fn (` p `) ->` t" are evident).[9] The corresponding semantic mapping

$$\mathcal{TE} \quad : \quad \text{TExpr} \rightarrow \mathit{STATE} \rightarrow (\mathit{VALUE} \rightarrow \mathit{RULE\_DENOTATION})$$

is defined as follows:

$$\mathcal{TE}[\![\,\texttt{tn (}\,p\,\texttt{) ->}\;R\,]\!](S) \quad = \quad \lambda x.\Delta[\![\,R\,]\!](S,\,\mathcal{M}[\![\,p\,]\!](x)).$$

If *rule_id* is the name of a rule bound in the current context to a transition expression *texpr* (by a named rule definition of the form "*rule_id* `==` *texpr*"), then its interpretation $\mathbf{rule\_id}_S$ in a given state S corresponds to $\mathcal{TE}[\![\,\mathit{texpr}\,]\!](S)$ (see also semantics of named rule applications in Table 5.4).

[9] Note that λ-rules are usually not written explicitly, but obtained by the syntactic expansion of derived forms of named rule definitions (see Sect. 3.8.3).

Deterministic Semantics

$$\Delta[\![\, \texttt{skip} \,]\!](S,\rho) \;=\; \emptyset$$

$$\Delta[\![\, \mathit{fun\_id}\ \texttt{(}\ t_L\ \texttt{)} := t_R \,]\!](S,\rho) \;=\; \{\, (\ (\mathit{fun\_id}, \mathcal{E}[\![\, t_L \,]\!](S,\rho)),\ \mathcal{E}[\![\, t_R \,]\!](S,\rho)\ )\,\}$$

$$\Delta[\![\, R_1\ \ldots\ R_n \,]\!](S,\rho) \;=\; \sum_{i=1}^{n} \Delta[\![R_i]\!](S,\rho)$$

$$\Delta[\![\, \texttt{if}\ G\ \texttt{then}\ R_T\ \texttt{else}\ R_F\ \texttt{endif} \,]\!](S,\rho) \;=\; \begin{cases} \Delta[\![R_T]\!](S,\rho) & \text{if } (S,\rho) \models G \\ \Delta[\![R_F]\!](S,\rho) & \text{otherwise} \end{cases}$$

$$\Delta[\![\, \texttt{case}\ t_0\ \texttt{of}\ p\texttt{:}R_1\texttt{;}\ \texttt{otherwise}\ R_2\ \texttt{endcase} \,]\!](S,\rho) \;=$$
$$= \begin{cases} \Delta[\![\, R_1 \,]\!](S, \rho \oplus \rho') & \text{if } \rho' \neq \text{FAIL} \\ \Delta[\![\, R_2 \,]\!](S,\rho) & \text{otherwise} \end{cases}$$
where $\rho' = \mathcal{M}[\![\, p \,]\!]\,(\mathcal{E}[\![\, t_0 \,]\!](S,\rho))$

$$\Delta[\![\, \texttt{do forall}\ p\ \texttt{in}\ A\ \texttt{with}\ G\ \ R\ \ \texttt{enddo} \,]\!](S,\rho) \;=$$
$$= \sum_{x \in X} \Delta[\![\, R \,]\!](S, \rho \oplus \mathcal{M}[\![\, p \,]\!](x))$$
where $X = \text{range}_{S,\rho}(p, A, G)$.

$$\Delta[\![\, \mathit{rule\_id}\ \texttt{(}\ t\ \texttt{)} \,]\!](S,\rho) \;=\; \mathbf{rule\_id}_S(\mathcal{E}[\![\, t \,]\!](S,\rho))$$

Nondeterministic Semantics

For `skip` and update rules ($R \equiv$ `skip` or $R \equiv \mathit{fun\_id}\texttt{(}t_L\texttt{)} := t_R$):

$$\mathbf{\Delta}[\![\, R \,]\!](S,\rho) = \{\, \Delta[\![\, R \,]\!](S,\rho) \,\}$$

For block, conditional, `case`, and `do forall` rules:

replace Δ by $\mathbf{\Delta}$ in equations for the deterministic case
(note the different interpretation of + on update set families).

For `choose` rules:

$$\mathbf{\Delta}[\![\, \texttt{choose}\ p\ \texttt{in}\ A\ \texttt{with}\ G\ \ R\ \ \texttt{endchoose} \,]\!](S,\rho) \;=$$
$$= \begin{cases} \{\, \emptyset \,\} & \text{if } X = \emptyset \\ \bigcup\limits_{x \in X} \mathbf{\Delta}[\![\, R \,]\!](S, \rho \oplus \mathcal{M}[\![\, p \,]\!](x)) & \text{otherwise} \end{cases}$$
where $X = \text{range}_{S,\rho}(p, A, G)$.

Table 5.4: Semantics of Transition Rules

Part II

Tools

Chapter 6

The ASM Workbench

As a framework for the systematic development of tools for Abstract State Machines, we developed the *ASM Workbench*. This chapter describes the basic concepts of its architecture and its functionalities. We begin by discussing the situation which motivated the development of the Workbench (Sect. 6.1). Then we describe the tool architecture (Sect. 6.2), the basic functionalities of its kernel (Sect. 6.3) and the generic mechanisms it provides (Sect. 6.4). Finally, in Sect. 6.5 we refer to an industrial case study carried out with the help of the ASM Workbench and, in Sect. 6.6, give an overview of related work (existing ASM tools).

6.1 Motivation

There is general agreement on the fact that appropriate tool support is needed in order to gain practical advantages from the application of formal methods to concrete specification and modelling tasks. Useful tools include:

- syntax-directed and graphical editors;
- static checkers (e.g., syntax and type checkers);
- prototyping/simulation/animation tools and code generators;
- verification tools (model checkers, proof checkers, theorem provers).

This consideration obviously applies to Abstract State Machines too. Potentially useful ASM tools can be roughly subdivided into the following two classes:

1. tools supporting a user in the process of developing ASM specifications, such as editors, static analysers, interpreters, debuggers: they assist the specifier in formalizing informal descriptions of systems or requirements and improving such formalizations as the result of an iterative process, which involves experimentation as an important means of validation;

2. tools translating ASM specifications into other languages in order to allow the processing of these specifications by means of other tools, e.g. code generators transforming ASM specifications into programs or transformation tools mapping them into the language of a particular logic: tools of this kind enable the use of state-of-the-art compilers and verification tools to produce efficient executable code or to verify properties of the specifications, respectively.

Note that, while the former tools (*interactive development tools*) are helpful during the formalization phase, the latter (*transformation tools*) come into play after it, when the ASM models formalizing the problem at hand are completed.

The current state of the art of tool support for Abstract State Machines—despite the efforts of many people and the improvements achieved during the last years (see Sect. 6.6 for an account of the existing ASM tools)—is not yet completely satisfactory.

On one hand, there are several ASM simulators (mostly interpreter-based), which usually—besides the possibility of executing ASM specifications—do not provide any other interesting feature.[1] With the exception of the XASM tool [2], which includes a debugging feature, these simulators provide very restricted possibilities of user interaction, therefore they are not very convenient as interactive development tools. Moreover, many of them rely on the implementation language in order to define certain parts of the specifications, such as static function definitions: this forces the user to write programs in this language to define particular aspects of a specification (at a level of abstraction which may be too low, e.g. C code) and presupposes the availability of a development environment for the implementation language.

On the other hand, there are essentially no trasformation tools. Although, for instance, some successful approaches to mechanical verification of properties of ASM specifications [79, 86, 90] by theorem provers (KIV [77] and PVS [75]) or model checkers (SMV [66]) employ transformations of ASMs that are quite general and thus subject to be automated, these transformations have not been implemented, probably due—among other things—to the relatively high amount of implementation work needed (for the case studies considered in the papers the transformations have been applied by hand). However, the implementation effort could be drastically reduced if an appropriate framework/library for the development of ASM tools were available.

In response to the situation described above, we developed the *ASM Workbench* (*ASM-WB* for short), a tool environment intended to support both the development process and transformations of ASMs, based on a open architecture providing some basic functionalities and easily extensible.

[1] A notable exception is the Gem-Mex tool [3], which includes a graphical editor for control/data-flow diagrams and documentation generation features: however, the Gem-Mex tool supports the special ASM-based methodology of *Montages* [3] for specifying programming languages, not ASMs in their general form.

6.2 Architecture

The main characteristics of the ASM Workbench architecture are its *kernel* (a set of program modules implemented in the functional language Standard ML [72, 71, 76]) and a set of *exchange formats* (textual representations of the kernel's data structures, used for importing and exporting ASM-related information). The existing components (tools) of the ASM Workbench—implemented either as part of the kernel or as additional modules—include a *type-checker*, an interpreter-based *simulator*, a graphical user interface (*browser/debugger*), and an interface to the *model-checker* SMV (for checking properties of a particular class of ASMs).

The Kernel The essential functionalities of the ASM Workbench are contained in its *kernel*, which consists of:

- a collection of *data structures* representing *syntactic objects* as well as *semantic objects* of ASM specifications (collectively denoted as *ASM objects*). Syntactic objects are, for instance, terms, transition rules, and signatures, while semantic objects include *values* (i.e., the elements of the ASM *superuniverse* or *base set*), states, update sets, runs, etc.;
- a collection of *functions* to process the ASM objects, to be used as building blocks for the construction of more complex functions up to complete tool components. The kernel provides, for instance, functions to type-check a term or rule, to evaluate a term to a value or a transition rule to an update set, to fire an update set in a given state, and so on.

The kernel is implemented as a set of Standard ML *modules*, each module corresponding to a relevant data structure (e.g., abstract syntax trees, signatures) or functionality (e.g., type-checker, evaluator). Moreover, there is a collection of auxiliary modules which can be reused for further tool development, including modules for importing and exporting ASM objects (see "Exchange Formats" below), a module for accessing the ASM-SL definition database, and modules to ease the implementation of structural-inductive transformations defined over the ASM abstract syntax (see Sect. 6.4 below).

Thanks to a feature of the Standard ML compiler, it is possible to export an executable image of the ML compiler itself (usually called `sml-asm`) containing the precompiled and preloaded kernel. In this way `sml-asm`, besides constituting a first—not very friendly—user interface for the ASM Workbench, provides immediate access to the data structures and functions of the ASM-WB kernel, which can be directly used for the development of further—tightly coupled—tool modules. (Table 6.1 shows an example session with `sml-asm`).

Exchange Formats Most data structures of the kernel come with corresponding textual representations (*exchange formats*), to allow importing and exporting ASM-related information: this enables loosely coupled tool integration. For instance, one could use the ASM-WB parser and type checker as front-end for a

```
- ASM.load_file' "while_language_ast.asm";
val it =
  ["ID","EXPR","STMT","display_list","display_expr","display_stmt",
   "display_stmt_list"] : string list
- ASM.load_file' "while_language.asm";
val it =
  ["VALUE","interpretation","eval_expr_in_env","program","curr_stmt",
   "curr_cont",...] : string list
- ASM.eval_term' "eval_expr";
[interactive definition]:1: [1(1)-1(10)]:
type check error -- function eval_expr : EXPR -> VALUE has argument of type ()
uncaught exception Error
- ASM.eval_term' "curr_stmt";
val it = CELL ("Seq",[CELL (#,#)]) : ASM_Domains.VALUE
- print (ASM.show_value it);
Seq ([ Input ("max"), Assign ("x", Con (1)), While (App ("<=", [ Var ("x"), Var
("max") ]), Seq ([ If (App ("=", [ App ("mod", [ Var ("x"), Con (2) ]), Con (0)
]), Output (Var ("x"))), Assign ("x", App ("+", [ Var ("x"), Con (1) ])) ])) ])
val it = () : unit
- ASM.set_program "Interpreter";
val it = () : unit
- ASM.step ();
val it = () : unit
- print (ASM.show_value (ASM.eval_term' "curr_stmt"));
Input ("max")
val it = () : unit
- ...
```

Table 6.1: Example Session with `sml-asm`

transformation tool. The transformation tool could take as input the annotated AST exported by the type-checker: in this way, the transformation tool would not need to care about the ASM-SL concrete syntax and could rely on the fact that its input corresponds to a well-formed and type-correct ASM specification (where all "syntactic sugar" has been eliminated). Such an approach can contribute to a considerable reduction of the overall implementation effort, while not requiring any knowledge of the internal workings of the ASM Workbench.

6.3 Basic Functionalities

The basic ASM-WB functionalities include the type checker, the simulator, and the GUI. The interface to SMV is an extension, described in detail in Chapter 7.

Type Checker A type checker, implementing the type system discussed in Chapter 4, is part of the ASM-WB kernel. Its core is an efficient implementation of the well-known unification-based type inference algorithm [27], and can be used—even independently from the ASM interpreter—to check that ASM-SL specifications are well-typed and to infer their signatures. It also performs other simple static checks, e.g., it checks that static functions are never updated. The result of successful type checking is an abstract syntax tree (AST) with type

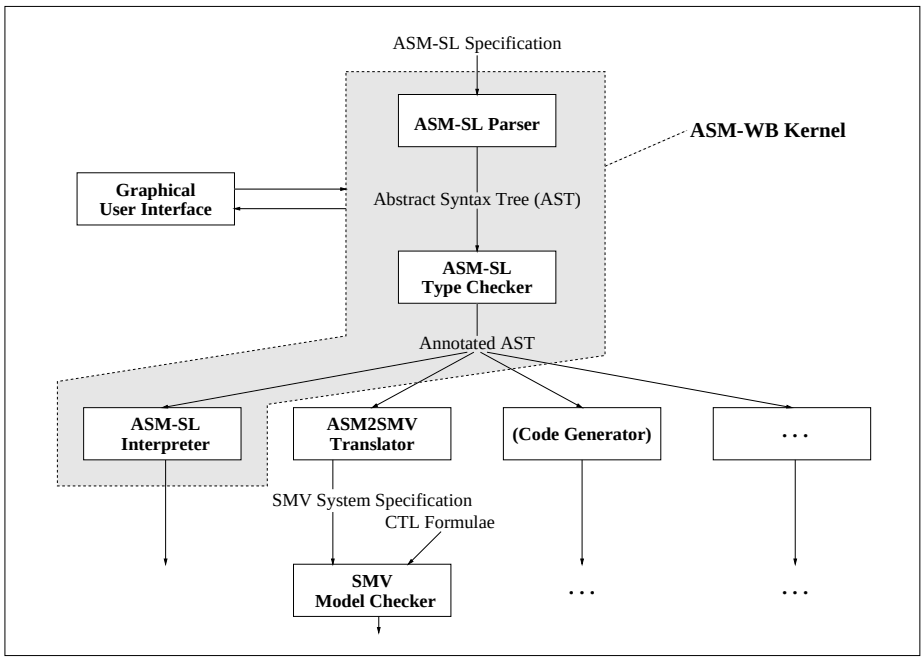


Figure 6.1: The ASM Workbench Tool Environment

annotations (see Fig. 6.1).

Simulator A simple interpreter, which allows to simulate Abstract State Machines described in ASM-SL notation, is also part of the kernel. It is based on: *(i)* an evaluator, containing functions—defined inductively over the ASM-SL abstract syntax trees—which compute values from terms and update sets from rules[2], and *(ii)* a module defining operations to manipulate runs, while keeping track of the computation history, such that computation steps can be retracted (*backward step* feature).

A peculiar feature of the simulator is the implementation of *external functions*: to increase the flexibility of the simulator, the mechanism to fetch values of external functions is not fixed *a priori*. Instead, this task is delegated to an external process, the so-called *oracle process*. The interaction between the oracle and the interpreter works as follows: the oracle waits for input from the interpreter; whenever the interpreter needs some external function value, it sends the function name and the arguments (in a form complying to the appropriate exchange format) to the oracle, thus activating it, and waits for the result; at this point, the oracle may perform any actions needed in order to compute or retrieve the requested value and, when finished, sends the result back to the interpreter, thus reactivating it, and waits for the next query.

[2]Essentially, the evaluator implements the denotational semantics of terms and rules.

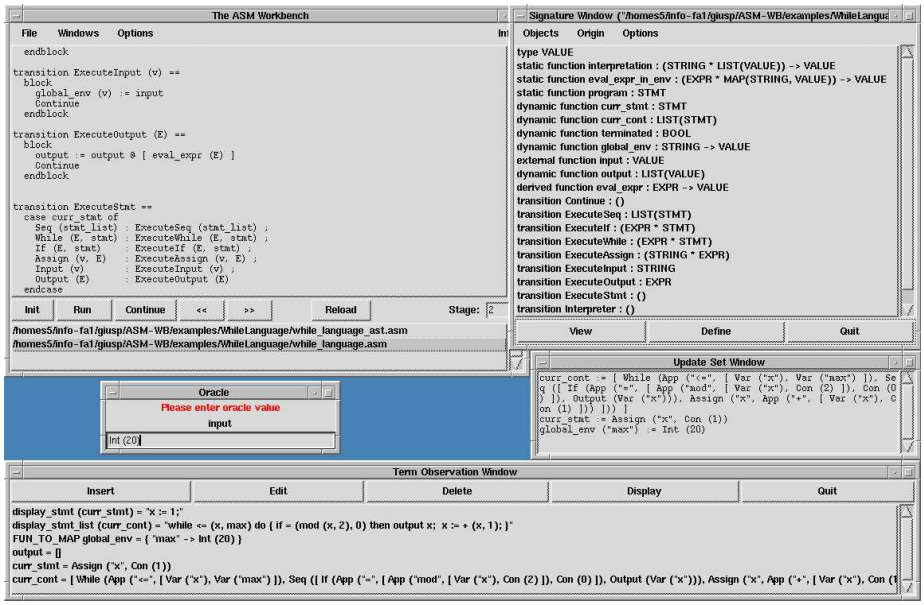


Figure 6.2: The ASM Workbench's Graphical User Interface

Graphical User Interface Finally, the ASM Workbench provides a Graphical User Interface (GUI), which presents the information contained in ASM-SL specifications (e.g., signatures) in an orderly form and allows the user to control the simulation and inspect its results (e.g., by performing single steps forward or backward, setting breakpoints, observing the values of some terms, etc.). In this sense, the ASM Workbench's GUI can be seen as a "browser" and "debugger" of ASM specifications.

The GUI also provides a default oracle, which simply invites the user to enter a value of the appropriate type. Despite its simplicity, the default oracle proves to be useful to make the first experiments with ASM models. Later, it can be replaced by more sophisticated (application-specific) oracles implementing realistic simulation scenarios.

A snapshot of the ASM Workbench's GUI is shown in Fig. 6.2 (the example specification is the while-language semantics from Chapter 2). It is possible to see: in the main window—the large one on the left—the ASM-SL rules for the semantics of statements (`while_language.asm`), the debugger controls, and the list with the specification files; in the signature window—above on the right—the signature inferred for the definitions from `while_language.asm`; in the oracle window—the small one under the main window—a value that the user just entered for the external function `input`; in the update set window—under the signature window—the update set fired in the last step; and finally, in the term observation window—below—a part of the ASM state, identified by a set of terms selected by the user.

6.4 Generic Mechanisms

In addition to the essential functionalities mentioned in the previous section, the ASM-WB kernel includes a few modules implementing generic mechanisms which should ease further tool developments, namely: *structural induction* over the ASM-SL abstract syntax and its extension to polymorphic *annotated ASTs*.

6.4.1 Structural Induction

The ideas underlying the structural induction module go back to some elementary notions of algebraic specification, which can be directly implemented in ML (or any other functional language with a similar type system). A slightly different presentation of induction-based programming can also be found in [67], where so-called "generalized fold operators" are presented as a general principle to structure functional programs. In this section, we first briefly recall the relevant notions of algebraic specification, following the notation of [87]. Then, we describe how they are implemented in the ASM Workbench. Finally, in order to exemplify the technique, we present two applications (evaluation and substitution).

Basic Notions A *signature* Σ is a pair $\langle S, F\rangle$, where S is a set of *sorts* and F is a set of *function symbols* equipped with a mapping $type : F \to S^* \times S$ (where the notation $f : \overline{s} \to s$ is often used to denote $f \in F$ with $type(f) = \langle \overline{s}, s\rangle$). Constants (of sort S) are represented by nullary function symbols $c : () \to s$, written $c : s$ for short. If $\Sigma = \langle S, F\rangle$ is a signature[3], then:

- For an S-sorted set of variables X (disjoint from the set F of function symbols), the set $T(\Sigma, X)_s$ of *terms* of sort s (with variables in X) is the least set containing: *(i)* every $x \in X_s$, and *(ii)* every $f(t_1, \ldots, t_n)$ where $f : s_1, \ldots, s_n \to s \in F$ and t_i $(i = 1, \ldots, n)$ is a term in $T(\Sigma, X)_{s_i}$. Moreover, $T(\Sigma) =_{\text{def}} T(\Sigma, \emptyset)$ is the corresponding set of *ground* terms.[4]

- A Σ-*algebra* A consists of an S-sorted family of nonempty carrier sets $\{A_s\}_{s\in S}$ (also denoted by A) and a total function $f^A : A_{s_1} \times \ldots \times A_{s_n} \to A_s$ for each $f : s_1, \ldots, s_n \to s \in F$. Note that ground terms form a Σ-algebra $T(\Sigma)$ with carrier sets $\{T(\Sigma)_s\}_{s\in S}$ and $f^{T(\Sigma)}(t_1, \ldots, t_n) = f(t_1, \ldots, t_n)$ for each $f : s_1, \ldots, s_n \to s \in F$ and $t_i \in T(\Sigma)_{s_i}$ $(i = 1, \ldots, n)$.

- If A and B are two Σ-algebras, a Σ-*homomorphism* $h : A \to B$ is a family of maps $\{h_s : A_s \to B_s\}_{s\in S}$ such that, for each $f : \overline{s} \to s \in F$ and each $a_i \in A_{s_i}$ $(i = 1, \ldots, n)$, $h_s(f^A(a_1, \ldots, a_n)) = f^B(h_{s_1}(a_1), \ldots, h_{s_n}(a_n))$. (Intuitively, a Σ-homomorphism maps the data types of one Σ-algebra to those of another in such a way that the operations are preserved.)

[3] We consider here only *sensible* signatures, i.e., signatures which admit at least one ground term for each sort.

[4] As usual, if $c : s$ is a constant name, one writes c as a shorthand for the term $c()$.

Note that there exists a unique Σ-homomorphism $h : T(\Sigma) \to A$ from the ground term algebra $T(\Sigma)$ to any other Σ-algebra A, defined by

$$h_s(f(t_1, \ldots, t_n)) = f^A(h_{s_1}(t_1), \ldots, h_{s_n}(t_n)) \tag{6.1}$$

for each $f : s_1, \ldots, s_n \to s \in F$ and $t_1 \in T(\Sigma)_{s_1}, \ldots, t_n \in T(\Sigma)_{s_n}$ (corresponding to the interpretation of ground terms in A: for every $t \in T(\Sigma)$, $h(t) = t^A$).

It is well known that the (abstract) syntax of a formal language, described by a context-free grammar G, can be represented by a signature Σ_G, containing a sort for each syntactic category of the language (i.e., for each non-terminal of G) and a function name for each language construct (i.e., for each production of G), such that each phrase of the language corresponds to a ground term of the appropriate sort [8, 74]. Tables 6.2 and 6.3 illustrate this construction for a subset of ASM-SL, which we use as a running example in the rest of this section.

If the language phrases are encoded into the ground term algebra in this way, the equations (6.1) correspond to a program transformation scheme defined by induction over the structure of phrases. In order to unambiguously specify a particular transformation scheme, it suffices to specify the algebra A, i.e., its carriers sets A_s and all functions f^A for $f \in F$.

A notable example of program transformation expressible in terms of the algebraic construction sketched above is the dynamic semantics of ASM-SL, as specified in Chapter 5. In this case, the *semantic domains* of Chapter 5 correspond to carrier sets A_s of the algebra A, while the *semantic mappings* correspond to the components h_s of the Σ-homomorphism h defined by the equations (6.1). Although the ASM-SL semantics has been defined in Chapter 5 by means of explicit recursion, it could also be defined as Σ_G-homomorphism according to the scheme illustrated above. This leads to more concise definitions, as one needs to specify the functions f^A, but not the recursion scheme implicit in the definition of Σ-homomorphism (equations 6.1). This technique is exemplified below for a small subset of ASM-SL.

A Minimal Example To exemplify the technique, we first consider a very small subset of ASM-SL consisting only of ground terms (function applications, tuple terms, and conditional terms).

The grammar G specifying the concrete syntax for this language fragment is the following:

term	→	*fun_id* `(` *term* `)`
	|	`(` *term* `,` `...` `,` *term* `)`
	|	`if` *term* `then` *term* `else` *term* `endif`

From a strictly formal point of view, this is not a proper context-free grammar (because of the second production, concerning tuple terms). However, to avoid unnecessary overhead, it is convenient to use an informal notation for sequences, which is more concise and readable and could be easily formalized, if needed.

As mentioned above, the abstract syntax of a formal language can be represented by a signature Σ_G, systematically derived from the grammar G that

defines the concrete syntax of the language. As Σ_G must contain a sort for each syntactic category (non-terminal) of G, the signature Σ_G for our example has two sorts `FUN_ID` and `TERM` for function names and terms, respectively.[5] For each language construct there is a function symbol $f : \overline{s} \to s$ in Σ_G, whose type is extracted from the corresponding production of G as follows:

- $\overline{s}$ is the sequence of sorts corresponding to non-terminals occurring on the right-hand side of the production.
- s is the sort corresponding to the non-terminal on the left-hand side, i.e., the syntactic category of the construct under consideration.

In our example, we call the relevant function symbols `AppTerm`, `TupleTerm`, and `CondTerm`, obviously standing for application term, tuple term, and conditional term. Then, the signature Σ_G extracted from the grammar G above will be the following.

Sorts:	`FUN_ID`
	`TERM`
Function symbols:	`AppTerm` : `FUN_ID`, `TERM` → `TERM`
	`TupleTerm` : `LIST(TERM)` → `TERM`
	`CondTerm` : `TERM`, `TERM`, `TERM` → `TERM`

Note again that, for convenience, we freely use sequences (lists) in order to handle tuple terms.

Now ASM-SL terms—whose syntax is specified by the grammar G—can be easily encoded into the ground term algebra $T(\Sigma_G)$. In particular, if we denote the encoding of a phrase α as $\overline{\alpha}$:

$$\begin{array}{rcl} \overline{f\ \texttt{(}\ t\ \texttt{)}} & = & \texttt{AppTerm (}\ \overline{f}\texttt{,}\ \overline{t}\ \texttt{)} \\ \overline{\texttt{(}\ t_1\ \texttt{,}\ \ldots\ \texttt{,}\ t_n\ \texttt{)}} & = & \texttt{TupleTerm ([}\ \overline{t_1}\texttt{,}\ \ldots\texttt{,}\ \overline{t_n}\ \texttt{])} \\ \overline{\texttt{if}\ G\ \texttt{then}\ t_1\ \texttt{else}\ t_2\ \texttt{endif}} & = & \texttt{CondTerm (}\ \overline{G}\texttt{,}\ \overline{t_1}\texttt{,}\ \overline{t_2}\ \texttt{).} \end{array}$$

For instance, the term $t_{\text{ex}} \equiv$ `if a then b else f(a,b) endif` will be encoded as

```
t_ex ≡ CondTerm (
         AppTerm ("a", TupleTerm ([])),
         AppTerm ("b", TupleTerm ([])),
         AppTerm ("f", TupleTerm ([ AppTerm ("a", TupleTerm ([])),
                                    AppTerm ("b", TupleTerm ([])) ]))
       )
```

assuming that the sort `FUN_ID` of function names has been identified with a sort of strings, denoted by corresponding string constants.[6] (Table 6.2 describes the syntax encoding for a larger ASM-SL subset, including also patterns and rules.)

[5] Actually, *fun_id* is a terminal symbol in the ASM-SL grammar. However, as different function names are distinguished from each other by means of an appropriate attribute (the actual function name), the grammar symbol *fun_id* must be treated like a non-terminal in this context. Keywords like "`(`", "`)`", "`,`" and "`if`" are treated as proper terminals, instead.

[6] To understand the encoding, one should also recall that "`if a then b else f(a,b) endif`" is a short form for "`if a() then b() else f(a(),b()) endif`", where "`()`" is the empty tuple.

p	$\rightarrow$ *var_id*	$\mapsto$ `VarPatt(`*var_id*`)`
	$\mid$ *fun_id* `(` p' `)`	$\mapsto$ `AppPatt(`*fun_id*`,`p'`)`
	$\mid$ `(` p_1 `,` $\ldots$ `,` p_n `)`	$\mapsto$ `TuplePatt([`p_1`,`$\ldots$`,`p_n`])`
t	$\rightarrow$ *var_id*	$\mapsto$ `VarTerm(`*var_id*`)`
	$\mid$ *fun_id* `(` t' `)`	$\mapsto$ `AppTerm(`*fun_id*`,`t'`)`
	$\mid$ `(` t_1 `,` $\ldots$ `,` t_n `)`	$\mapsto$ `TupleTerm([`t_1`,`$\ldots$`,`t_n`])`
	$\mid$ `if` G `then` t_1 `else` t_2 `endif`	$\mapsto$ `CondTerm(`G`,`t_1`,`t_2`)`
	$\mid$ `case` t_0 `of` p `:` t_1 `; otherwise` t_2 `endcase`	$\mapsto$ `CaseTerm(`t_0`,`p`,`t_1`,`t_2`)`
	$\mid$ `{` t `|` p `in` U `with` G `}`	$\mapsto$ `SetCompr(`t`,`p`,`U`,`G`)`
R	$\rightarrow$ `skip`	$\mapsto$ `SkipRule`
	$\mid$ *fun_id* `(` t `)` `:=` t'	$\mapsto$ `UpdateRule((`*fun_id*`,`t`),`t'`)`
	$\mid$ $R_1 \ldots R_n$	$\mapsto$ `BlockRule([`R_1`,`$\ldots$`,`R_n`])`
	$\mid$ `if` G `then` R_1 `else` R_2 `endif`	$\mapsto$ `CondRule(`G`,`R_1`,`R_2`)`
	$\mid$ `case` t_0 `of` p `:` R_1 `; otherwise` R_2 `endcase`	$\mapsto$ `CaseRule(`t_0`,`p`,`R_1`,`R_2`)`
	$\mid$ `do forall` p `in` U `with` G R' `enddo`	$\mapsto$ `DoForallRule(`p`,`U`,`G`,`R'`)`

Table 6.2: Term-Encoding of ASM-SL Abstract Syntax (Patterns, Terms, Rules)

As already mentioned at the beginning of this section, there is a unique Σ_G-homomorphism $h : T(\Sigma_G) \rightarrow A$ from the ground term algebra $h : T(\Sigma_G)$ to any other Σ-algebra A. The homomorphism h satisfies the equations (6.1), which also provide a way to compute h, provided that the functions f^A are known. Here, we extend h to deal with lists in a standardized way (as needed, for instance, to deal with tuple terms in our example):

$$h_{\mathtt{LIST}(s)}([\,t_1,\ldots,t_n\,]) = [\,h_s(t_1),\ldots,h_s(t_n)\,] \tag{6.2}$$

Then, in the example, h will be as follows:

$$\begin{array}{lcl} h = \{\ h_{\mathtt{FUN\_ID}} : T(\Sigma_G)_{\mathtt{FUN\_ID}} \rightarrow A_{\mathtt{FUN\_ID}}, \\ \quad\ h_{\mathtt{TERM}} : T(\Sigma_G)_{\mathtt{TERM}} \rightarrow A_{\mathtt{TERM}}\ \} \end{array}$$

where

$$\begin{array}{lcl} h_{\mathtt{TERM}}(\mathtt{AppTerm}(f,\,t)) & = & \mathtt{AppTerm}^A\,(h_{\mathtt{FUN\_ID}}(f), h_{\mathtt{TERM}}(t)) \\ h_{\mathtt{TERM}}(\mathtt{TupleTerm}([\,t_1,\ldots,t_n\,])) & = & \mathtt{TupleTerm}^A\ (h_{\mathtt{LIST(TERM)}}([\,t_1,\ldots,t_n\,])) \\ & = & \mathtt{TupleTerm}^A\ ([\,h_{\mathtt{TERM}}(t_1),\ldots,h_{\mathtt{TERM}}(t_n)\,]) \\ h_{\mathtt{TERM}}(\mathtt{CondTerm}(G,\,t_1,\,t_2)) & = & \mathtt{CondTerm}^A\,(h_{\mathtt{TERM}}(G), h_{\mathtt{TERM}}(t_1), h_{\mathtt{TERM}}(t_2)) \end{array}$$

Note that $h_{\mathtt{TERM}}$ depends only on the choice of the functions $\mathtt{AppTerm}^A$, $\mathtt{TupleTerm}^A$ and $\mathtt{CondTerm}^A$ in the target algebra A.

To conclude this first example, we finally show how the semantics of terms can be defined by means of the technique discussed here. For simplicity, we consider the interpretation of ASM-SL function names as fixed.[7] First, the carrier sets of the target algebra A must be chosen. As the interpretation of a function name is a function, the carrier set for `FUN_ID` is given by the set of functions over

[7] The next example will show how to deal with states and environments.

```
datatype NAME =
  IntConst of int
| FunId of string

datatype PATT =
  VarPatt of string
| AppPatt of NAME*PATT
| TuplePatt of PATT list

datatype TERM =
  VarTerm of string
| AppTerm of NAME*TERM
| TupleTerm of TERM list
| CondTerm of TERM*TERM*TERM
| CaseTerm of TERM*PATT*TERM*TERM
| SetCompr of TERM*PATT*TERM*TERM

datatype RULE =
  SkipRule
| UpdateRule of (NAME*TERM)*TERM
| BlockRule of RULE list
| CondRule of TERM*RULE*RULE
| CaseRule of TERM*PATT*RULE*RULE
| DoForallRule of PATT*TERM*TERM*RULE
```

Table 6.3: Implementation of $T(\Sigma_G)$—(Abstract Syntax Trees)

the base set *VALUE*. The semantics of a term, instead, is a value, therefore the carrier set for `TERM` coincides with *VALUE*. Then, the functions $\texttt{AppTerm}^A$, $\texttt{TupleTerm}^A$ and $\texttt{CondTerm}^A$, which uniquely determine $h_{\texttt{TERM}}$, have to be defined. The component $h_{\texttt{FUN\_ID}}$ of the homomorphism h, instead, is left abstract here, and we simply assume that it yields the appropriate interpretations of the function names. The target algebra A is as follows:

$$
\begin{aligned}
A_{\texttt{FUN\_ID}} &= \mathit{VALUE} \to \mathit{VALUE} \\
A_{\texttt{TERM}} &= \mathit{VALUE} \\
\texttt{AppTerm}^A\,(f,\, t) &= f(t) \\
\texttt{TupleTerm}^A\,([\,t_1, \ldots, t_n\,]) &= (t_1, \ldots, t_n) \\
\texttt{CondTerm}^A\,(G,\, t_1,\, t_2) &= \begin{cases} t_1 & \text{if } G = \mathbf{true} \\ t_2 & \text{otherwise.} \end{cases}
\end{aligned}
$$

The component $h_{\texttt{TERM}}$ of the Σ_G-homomorphism induced by A defines the semantics of ASM-SL terms. According to the terminology of Chapter 5, $h_{\texttt{TERM}}$ is a semantic mapping, while the carrier sets $A_{\texttt{FUN\_ID}}$ and $A_{\texttt{TERM}}$ are semantic domains.

Implementation All the above constructions can be easily implemented in ML and, in fact, they are an essential part of the ASM Workbench kernel.

In particular, the ground term algebra $T(\Sigma_G)$ representing the abstract syntax trees (ASTs) for ASM-SL is realized by appropriate `datatype` declarations.[8] Table 6.3 shows the implementation of the ASTs for a larger subset of ASM-SL than the one of the previous example. Note that the syntactic category `FUN_ID` is replaced by `NAME`, in order to accommodate integer constants and other possible special forms of identifiers. After these type declarations, ASM-SL phrases can be represented as ML terms: for instance, the term $\overline{t_{\text{ex}}}$ from the example above (p. 79) is a valid ML term representing t_{ex}.

The structural-inductive program transformations based on the algebraic constructions discussed above are implemented in a modular way. This means

[8] ML datatypes are the same as ASM-SL free types.

that it is not necessary to specify the whole target algebra in order to define a transformation scheme. Instead, it is possible to define only the parts of the target algebra that are relevant to handle the syntactic categories of interest (e.g., only terms or rules). Correspondingly, there are a number of functions implementing the parts of the Σ_G-homomorphism relevant for the different syntactic domains.[9] More precisely:

1. For each sort s in the signature Σ_G, i.e., for each syntactic category, there is a record type "S_TARGET_ALGEBRA" collecting all functions f^A of the target algebra A with range A_s. Each field of the record contains one of the functions f^A, corresponding to a language construct of sort s.

2. For each sort s in Σ_G, there is an higher-order function "s_induction" implementing the component h_s of the Σ_G-homomorphism h, i.e., a generic structural-inductive transformation for the syntactic category s. Parameters of "s_induction" are, in addition to the specification of the target algebra for sort s (a record of type "S_TARGET_ALGEBRA"), a number of functions h_s' that specify how to transform subphrases of sort $s' \neq s$.[10]

Types corresponding to the carrier sets A_s of the target algebra must not be declared explicitly. Instead, "S_TARGET_ALGEBRA" and "s_induction" have polymorphic types, whose type parameters correspond to the carrier sets A_s.

For instance, for the minimal example including only ground terms, the record type for the target algebra is the following:

```
type ('fun_id, 'term) TERM_TARGET_ALGEBRA =
{ AppTerm   : 'fun_id * 'term -> 'term,
  TupleTerm : 'term list -> 'term,
  CondTerm  : 'term * 'term * 'term -> 'term }
```

The fields AppTerm, TupleTerm and CondTerm of the record should be filled with the respective functions AppTerm^A, TupleTerm^A and CondTerm^A, depending on the transformation to be realized. As an example, the target algebra which defines term semantics is represented by the following record:

```
val term_evaluation_algebra =
  { AppTerm   = fn (f, t) => f t,
    TupleTerm = fn ts => TUPLE ts,
    CondTerm  = fn (G, t1, t2) => if G then t1 else t2 }
```

Note that "val n = x" is a construct to name a value x as n, "fn p => t" is just the ML syntax for "$\lambda p.t$", and "TUPLE" is a function which constructs a tuple value $(x_1, \ldots, x_n) \in \mathit{VALUE}$ out of a list of values $[\,x_1, \ldots, x_n\,]$ (see also the implementation of the base set VALUE at the beginning of Table 6.5). Note also that we adopt the usual convention of appending an "s" to a variable name if this variable stands for a list, e.g., if t is a variable of type 'term, a variable of type 'term list will be called ts (as in the TupleTerm case above).

[9] The expressions "syntactic category" and "syntactic domain" are used as synonyms here.
[10] The h_s' are possibly, but not necessarily, defined by structural induction themselves.

```
type ('name, 'patt, 'term, 'rule) RULE_TARGET_ALGEBRA =
{ SkipRule     : 'rule,
  UpdateRule   : ('name * 'term) * 'term -> 'rule,
  BlockRule    : 'rule list -> 'rule,
  CondRule     : 'term * 'rule * 'rule -> 'rule,
  CaseRule     : 'term * 'patt * 'rule * 'rule -> 'rule,
  DoForallRule : 'patt * 'term * 'term * 'rule -> 'rule }

fun rule_induction
      (h_name :NAME -> 'name, h_patt :PATT -> 'patt, h_term :TERM -> 'term)
      (A :('name,'patt,'term,'rule) RULE_TARGET_ALGEBRA) (R :RULE) :'rule =
let val h_rule = rule_induction (h_name, h_patt, h_term) A
in case R of
     SkipRule              => (#SkipRule A)
   | UpdateRule ((f,t),t') => (#UpdateRule A) ((h_name f, h_term t), h_term t')
   | BlockRule Rs          => (#BlockRule A) (map h_rule Rs)
   | CondRule (G,R1,R2)    => (#CondRule A) (h_term G, h_rule R1, h_rule R2)
   | CaseRule (t0,p,R1,R2) => (#CaseRule A)
                                (h_term t0, h_patt p, h_rule R1, h_rule R2)
   | DoForallRule (p,U,G,R) => (#DoForallRule A)
                                (h_patt p, h_term U, h_term G, h_rule R)
end
```

Table 6.4: Implementation of Σ-Hom. $h : T(\Sigma_G) \to A$—(Structural Induction)

The higher order function "`term_induction`" implementing $h_{\texttt{TERM}}$, i.e., structural-inductive transformations of terms, is implemented as follows:

```
fun term_induction
      (h_fun_id :FUN_ID -> 'fun_id)
      (A :('fun_id, 'term) TERM_TARGET_ALGEBRA) :TERM -> 'term =
  let val h_term = term_induction h_fun_id A
  in fn t =>
       case t of
         AppTerm (f, t') => (#AppTerm A) (h_fun_id f, h_term t')
       | TupleTerm ts    => (#TupleTerm A) (map h_term ts)
       | CondTerm (G, t1, t2) =>
           (#CondTerm A) (h_term G, h_term t1, h_term t2)
  end
```

The first parameter `h_fun_id` is a function transforming function names into objects of type `'fun_id`, which corresponds to $h_{\texttt{FUN\_ID}}$. The second parameter `A` is a record of the type discussed above.[11] With respect to the ML notation used here, we mention that "`#`*fld rec*" extracts the field labelled *fld* from a record *rec*, while the predefined function "`map`" used in the `TupleTerm` case applies a given function f to all elements of a list, i.e.:

$$\texttt{map}\ f\ [x_1, \ldots, x_n] \quad = \quad [f(x_1), \ldots, f(x_n)].$$

[11] Table 6.4 shows a larger example, namely the complete implementation of structural induction for transition rules (`rule_induction`). Note that `rule_induction` has more parameters and more cases, but conceptually does not differ from the simplified `term_induction` above. In fact, all other "*s*`_induction`" functions are realized along the same line as well.

The `term_induction` function can now be applied in order to produce a function `eval_term` implementing the dynamic semantics (evaluation) of terms:

```
val eval_term =
  term_induction interpretation term_evaluation_algebra
```

It is assumed that a function `interpretation` : `FUN_ID` → (`VALUE` → `VALUE`), which provides the interpretation of function names, is defined.

An alternative notation to instantiate `term_induction`, equivalent to the other one, is the following:

```
fun eval_term t =
  term_induction interpretation term_evaluation_algebra t
```

An Extended Example The minimal example introduced above was deliberately oversimplified, as it had to expose the main concepts of the structural-inductive techniques without distracting the reader with unnecessary details. However, that example is too simple and not realistic. In fact, as already seen in Chapter 5, the value of a term t depends on the *context* in which t is evaluated, specifically a *state* S defining function interpretations and an *environment* ρ containing bindings for the logical variables which possibly occur free in t.

This kind of context-dependency is a common phenomenon, which does not only affect terms, but almost any construct of any programming/modelling language. Chapter 5 already showed how we deal with context-dependency using a well-known technique from denotational semantics. The idea is to define the semantics of a phrase as a function from the evaluation context to the actual intended evaluation result. For instance, in the case of terms, the evaluation context is a pair (*state*, *environment*), while the intended evaluation result is a *value*. For this reason, the semantic mapping for terms has been defined as

$$\mathcal{E} : \textsc{Term} \to \mathit{STATE} \times \mathit{ENV} \to \mathit{VALUE}.$$

in Chapter 5. Here we show how $\mathcal{E}$ can be implemented using the structural induction technique.

First, the subset of terms under consideration is extended to include variables (applied occurrences) and at least one variable-binding construct. As prototype for variable-binding terms we consider here, to keep the example simple, a `let`-term without pattern matching.[12] Thus, the grammar G is extended by two additional productions:

term → *var_id*
| `let` *var_id* `==` *term* `in` *term* `endlet`

Σ_G is extended accordingly with a sort `VAR_ID` (with a fixed $h_{\texttt{VAR\_ID}} = id$ and simply implemented as `string`, see Table 6.3) and two function symbols:

`VarTerm` : `VAR_ID` → `TERM`
`LetTerm` : `VAR_ID`, `TERM`, `TERM` → `TERM`.

[12] Actually, `let`-terms in ASM-SL are a special case of `case`-terms and support pattern matching. See Table 6.5 for the full implementation by structural induction of the dynamic semantics of ASM-SL (the ASM-WB *evaluator*), including pattern matching and `case`-terms.

The target algebra A must then be refined to take into account the fact that the interpretation of function names is state-dependent and the value of terms depends on a (*state*, *environment*) pair. Assuming that *STATE* and *ENV* are defined as in Chapter 5, the following algebra induces a $h_{\texttt{TERM}}$ which coincides to the semantic mapping $\mathcal{E}$, as defined in Table 5.3:

$$\begin{array}{rcl}
A_{\texttt{FUN_ID}} & = & \mathit{STATE} \to \mathit{VALUE} \to \mathit{VALUE} \\
A_{\texttt{TERM}} & = & \mathit{STATE} \times \mathit{ENV} \to \mathit{VALUE} \\
\\
\texttt{VarTerm}^A\,(v)\,(S,\rho) & = & \rho(v) \\
\texttt{AppTerm}^A\,(f,\,t)\,(S,\rho) & = & (f(S))\,(t(S,\rho)) \\
\texttt{TupleTerm}^A\,([\,t_1,\ldots,t_n\,])\,(S,\rho) & = & (t_1(S,\rho),\ldots,t_n(S,\rho)) \\
\texttt{CondTerm}^A\,(G,\,t_1,\,t_2)\,(S,\rho) & = & \begin{cases} t_1(S,\rho) & \text{if } G(S,\rho) = \textbf{true} \\ t_2(S,\rho) & \text{otherwise} \end{cases} \\
\texttt{LetTerm}^A(v,\,t_1,\,t_2)\,(S,\rho) & = & t_2\,(S, \rho \oplus \{v \mapsto t_1(S,\rho)\}).
\end{array}$$

Note that, in the equations above, the notation "$f^A\,(\mathit{args})\,(S,\rho) = \mathit{rhs}$" is used instead of the equivalent "$f^A\,(\mathit{args}) = \lambda(S,\rho).\mathit{rhs}$" only for better readability.

Application I: Evaluation As a first full-sized application of the structural induction mechanism, Table 6.5 shows an *evaluator* for the ASM-SL subset of Tables 6.2 and 6.3, which implements the dynamic semantics presented in Chapter 5 (deterministic case, no `choose`-rules).

The evaluator shown in Table 6.5 is an executable ML program.[13] Note that the types of the functions `eval_patt`, `eval_term` and `eval_rule` correspond to the types of the semantic mappings $\mathcal{M}$, $\mathcal{E}$ and Δ of Chapter 5. However, the special elements FAIL and ERROR used there are replaced, in the implementation, by corresponding *exceptions*.

The program of Table 6.5 does not contain any new concepts besides those already discussed in the simplified examples above. It is presented here just to give an impression of the size and appearance of a realistic program module developed with the help of the generic mechanisms of the ASM Workbench kernel (in this case, structural induction). Thus, no further explanations are needed, except a few ones concerning ML constructs and types/functions used in the evaluator.

The "pattern-matching λ-abstraction" is in ML a syntactic shorthand, expanded as follows:

$$\begin{array}{lcl}
\lambda\, p_1\,.\,t_1 & \equiv & \lambda x.\texttt{case}\ x\ \texttt{of}\ p_1\ \texttt{=>}\ t_1 \\
\texttt{|}\ p_2\,.\,t_2 & & \texttt{|}\ p_2\ \texttt{=>}\ t_2 \\
\quad\vdots & & \quad\vdots \\
\texttt{|}\ p_n\,.\,t_n & & \texttt{|}\ p_n\ \texttt{=>}\ t_n
\end{array}$$

Besides the built-in ML type "α `list`" of lists, the evaluator makes use of two further polymorphic types, "α `set`" for finite sets and "(α,β) `map`" for finite maps, defined in appropriate libraries.

[13] With the exception that the ML λ-term "`fn` p `=>` t" is written "$\lambda p.t$" to save some space.

It is also assumed that the following operations are defined:

- List operations `list_map` (synonym for the `map` function) and `zipWith`:

$$\begin{array}{rcl} \texttt{list\_map}\ f\ [x_1,\ldots,x_n] & = & [f(x_1),\ldots,f(x_n)] \\ \texttt{zipWith}\ g\ ([x_1,\ldots,x_n],[y_1,\ldots,y_n]) & = & [g(x_1,y_1),\ldots,g(x_n,y_n)] \end{array}$$

  (`zipWith`, called with two lists of different length, raises an exception).

- Set operations `set_empty`, `set_add`, `set_singleton`, `set_union`, `set_diff` (with the obvious meaning), `set_Union` (which computes the union of a list of sets, i.e., $\texttt{set\_Union}\ [S_1,\ldots,S_n] = S_1 \cup \ldots \cup S_n$), `set_map` (analogous to `list_map`), and `set_fold`:

$$\begin{array}{rcl} \texttt{set\_fold}\ (F,E)\ \emptyset & = & E \\ \texttt{set\_fold}\ (F,E)\ S & = & F(x_1, \texttt{set\_fold}\ (F,E)\ (S \setminus \{x_1\})) \ \text{ if } x_1 \in S \end{array}$$

  (where $F : A \times B \to B$ should satisfy $F(x,F(y,Z)) = F(y,F(x,Z))$ for all $x, y \in A$, $Z \in B$; E must be an element of B).

- Map operations `map_empty`, `map_singleton`, `map_union`, and `map_Union`, with the same meaning as the corresponding set operations applied on sets of pairs representing the graph of the map(s), except that they raise an exception if the resulting set is not a map, i.e., if it contains two pairs (x,y) and (x,y') with $y \neq y'$.

  An additional map operation $\texttt{lookup} : (\alpha,\beta)\ \texttt{map} \to \alpha \to \beta$, as well as infix operators `|->` and `++`, with:

$$\begin{array}{rcl} \texttt{lookup}\ M\ x_0 = y_0 & \Leftrightarrow & \{(x,y) \in M \mid x = x_0\} = \{(x_0,y_0)\} \\ x\ \texttt{|->}\ y & = & \{(x,y)\} \\ M_1\ \texttt{++}\ M_2 & = & M_1 \oplus M_2 \end{array}$$

Finally, an abstract data type `STATE` representing ASM states is defined in the ASM-WB kernel, together with an operation

$\texttt{interpretation} : \texttt{STATE} \to \texttt{NAME} \to (\texttt{VALUE} \to \texttt{VALUE})$,

such that "`interpretation` S f" yields the interpretation of function name f in the state S. All other entities used in the evaluator are defined in Table 6.5.

Application II: Substitution Another class of applications is the definition of syntactic transformations, such as *substitution*. The goal here is to define two functions `subst_in_term` and `subst_in_rule` which, given a substitution $S = \{v_1 \mapsto t_1, \ldots, v_n \mapsto t_n\}$, transform a term t or a rule R in $t[t_i/v_i]_{i=1}^n$ or $R[t_i/v_i]_{i=1}^n$, respectively. An important requirement is that only free occurrences of variables are to be replaced. Patterns are clearly not affected by substitutions, as all variables occurring in patterns are defining occurrences. The task is actually quite simple, but defining the substitution functions by explicit recursion is tedious. A concise implementation can be obtained by structural induction and is shown in Table 6.6. Following this line, similar modules performing useful symbolic transformations, such as folding/unfolding, partial evaluation, etc., can be implemented quite straightforwardly.

```
datatype VALUE =
  INT of int                (* integers *)
| TUPLE of VALUE list       (* n-tuples, encoded as lists *)
| CELL of string * VALUE    (* cells, to represent values of free types *)
| SET of VALUE set          (* finite sets *)

val TRUE = CELL ("true", TUPLE [])

type ENV = (string, VALUE) map

type LOCATION   = NAME * VALUE
type UPDATE_SET = (LOCATION, VALUE) map
```

```
fun eval_patt (p :PATT) :VALUE -> ENV =
  patt_induction id {
    VarPatt = λ var_id.λ x.(var_id |-> x),
    AppPatt = λ(IntConst i,_).(λ x.if x = INT i then [] else raise FAIL)
              |(FunId f,p').(λ CELL(tag,x).if f = tag then p' x else raise FAIL
                                |_.raise FAIL),
    TuplePatt = λ ps.(λ TUPLE xs.map_Union (zipWith (λ(p,x).p x) (ps,xs))
                      |_.raise ERROR)
  } p

fun eval_cond (G,e1,e2) (S,env) =
  if G(S,env) = TRUE then e1(S,env) else e2(S,env)

fun eval_case (t0,p,e1,e2) (S,env) =
  e1(S,env ++ p(t0(S,env))) handle FAIL => e2(S,env)

fun eval_comprehension (F,E) (e,p,U,G) (S,env) =
  let fun F'(env',rest) = if G(S,env') = TRUE then F(e(S,env'),rest) else rest
  in case U(S,env) of
       SET X => set_fold (λ(x,rest).F'(env ++ p x,rest) handle FAIL => rest, E) X
     | _ => raise ERROR
  end

fun eval_term (t :TERM) :STATE * ENV -> VALUE =
  term_induction (id, eval_patt) {
    VarTerm   = λ var_id.λ(S,env).lookup env var_id,
    AppTerm   = λ(f,t').λ(S,env).(interpretation S f) (t'(S,env)),
    TupleTerm = λ ts.λ(S,env).TUPLE (list_map (λ t.t(S,env)) ts),
    CondTerm  = eval_cond,
    CaseTerm  = eval_case,
    SetCompr  = λ(t,p,U,G).SET ∘ ( eval_comprehension (set_add, set_empty)
                                     (t,p,U,G) )
  } t

fun eval_rule (R :RULE) :STATE * ENV -> UPDATE_SET =
  rule_induction (id, eval_patt, eval_term) {
    SkipRule   = λ(S,env).map_empty,
    UpdateRule = λ((f,t),t').λ(S,env).((f, t(S,env)) |-> t'(S,env)),
    BlockRule  = λ Rs.λ(S,env).map_Union (list_map (λ R.R(S,env)) Rs),
    CondRule   = eval_cond,
    CaseRule   = eval_case,
    DoForallRule = λ(p,U,G,R).( eval_comprehension (map_union, map_empty)
                                  (R,p,U,G) )
  } R
```

Table 6.5: Structural Induction — Application I: Evaluation

```
type SUBST = (string, TERM) map

fun vars_in_patt (p :PATT) :string set =
  patt_induction id
    { VarPatt = set_singleton, AppPatt = λ(_,p').p', TuplePatt = set_Union } p

fun bind (p :PATT) (S :SUBST) :SUBST =
  restrict S (set_diff (domain S, vars_in_patt p))

fun subst_in_term (t :TERM) :SUBST -> TERM =
  term_induction (id, id) {
    VarTerm   = λ var_id.λ S.(lookup S var_id) handle ERROR => VarTerm var_id,
    AppTerm   = λ(f,t').λ S.AppTerm (f,t' S),
    TupleTerm = λ ts.λ S.TupleTerm (list_map (λ t.t S) ts),
    CondTerm  = λ(G,t1,t2).λ S.CondTerm (G S,t1 S,t2 S),
    CaseTerm  = λ(t0,p,t1,t2).λ S.CaseTerm (t0 S,p,t1(bind p S),t2 S),
    SetCompr  = λ(t,p,U,G).λ S.let val S' = bind p S
                               in SetCompr (t S',p,U S,G S') end
  } t

fun subst_in_rule (R :RULE) :SUBST -> RULE =
  rule_induction (id, id, subst_in_term) {
    SkipRule     = λ S.SkipRule,
    UpdateRule   = λ((f,t),t').λ S.UpdateRule ((f, t S),t' S),
    BlockRule    = λ Rs.λ S.BlockRule (list_map (λ R.R S) Rs),
    CondRule     = λ(G,R1,R2).λ S.CondRule (G S,R1 S,R2 S),
    CaseRule     = λ(t0,p,R1,R2).λ S.CaseRule (t0 S,p,R1(bind p S),R2 S),
    DoForallRule = λ(p,U,G,R).λ S.let val S' = bind p S
                                  in DoForallRule (p,U S,G S',R S') end
  } R
```

Table 6.6: Structural Induction — Application II: Substitution

6.4.2 Context-Dependent Transformations

The examples presented above show that the use of structural induction in the way proposed here allows for quite concise definitions of program transformations. However, it can be observed that some unnecessary overhead is still present when the transformations to be defined are context-dependent.

Consider, for instance, the structural-inductive definition of the evaluation of tuple terms:

$$\texttt{TupleTerm}^A\,([\,t_1,\ldots,t_n\,])\,(S,\rho) \quad = \quad (t_1(S,\rho),\ldots,t_n(S,\rho))$$

A consistent part of this definition merely cares for passing around the context (S,ρ) without any changes. The same consideration applies to the substitution example, except that the context passed around there is a substitution instead of a (*state*, *environment*) pair.

Furthermore, it can be observed that only some constructs modify the context, and that the context modification mechanisms are shared by several constructs, such that it would make sense to reuse them. For instance, in the evaluation example, all variable-binding constructs modify the context by extending the environment with new variable bindings. The variable-binding mechanism is the same for `case` terms, comprehension terms, and `do forall` rules.

Therefore, the definition of context-dependent transformations can be simplified by delegating the propagation and modification of the context to the generic transformation scheme. This can be done by changing the structural-inductive transformation scheme h to handle context information. Of course, the new generic transformation scheme will no longer be a Σ-homomorphism, as it will necessarily contain some *ad hoc* context modification mechanisms, which depend on the nature of the language under consideration.

To illustrate this idea more concretely, we first show how a generic transformation scheme for context-dependent transformations (which will be named ϕ) can defined for the subset of terms from the extended example. Then we apply the transformation scheme ϕ to define term evaluation.

First of all, a domain CTX of contexts should be fixed. To simplify the presentation, it is assumed that the context is the same for all syntactic categories. If some components of the context are not needed, they can be ignored. In the evaluation example, for instance, the evaluation context for function names is a state. However, it does no harm to chose the set of (*state*, *environment*) pairs—the context of term evaluation—as global context domain: for the purpose of interpreting function names, the environment can be simply ignored.

Like the Σ_G homomorphism h, the transformation scheme ϕ has a component for each sort s:

$$\phi_s \ : \ T(\Sigma_G)_s \rightarrow CTX \rightarrow RES_s$$

Note that "$CTX \rightarrow RES_s$" in ϕ corresponds to A_s in h. For the syntactic domain of terms, $\phi_{\texttt{TERM}}$ can be defined as follows:

$$\begin{aligned}
&\phi_{\texttt{TERM}}\,(\texttt{VarTerm}\,(v))\,(ctx) = \\
&\quad = \texttt{VarTerm}^{\phi}\,(ctx)\,(\phi_{\texttt{VAR_ID}}(v)) \\
&\phi_{\texttt{TERM}}\,(\texttt{AppTerm}\,(f,\,t))\,(ctx) = \\
&\quad = \texttt{AppTerm}^{\phi}\,(ctx)\,(\phi_{\texttt{FUN_ID}}(f)(ctx),\,\phi_{\texttt{TERM}}(t)(ctx)) \\
&\phi_{\texttt{TERM}}\,(\texttt{TupleTerm}\,([\,t_1,\ldots,t_n\,]))\,(ctx) = \\
&\quad = \texttt{TupleTerm}^{\phi}\,(\phi_{\texttt{TERM}}(t_1)(ctx),\ldots,\phi_{\texttt{TERM}}(t_n)(ctx)) \\
&\phi_{\texttt{TERM}}\,(\texttt{CondTerm}\,(G,\,t_1,\,t_2))\,(ctx) = \\
&\quad = \texttt{CondTerm}^{\phi}\,(\phi_{\texttt{TERM}}(G)(ctx),\,\phi_{\texttt{TERM}}(t_1)(ctx),\,\phi_{\texttt{TERM}}(t_2)(ctx)) \\
&\phi_{\texttt{TERM}}\,(\texttt{LetTerm}(v,\,t_1,\,t_2))\,(ctx) = \\
&\quad = \texttt{LetTerm}^{\phi}\,(\phi_{\texttt{TERM}}(t_2)(bind^{\phi}(ctx)(\phi_{\texttt{VAR_ID}}(v),\phi_{\texttt{TERM}}(t_1)(ctx))))
\end{aligned}$$

where the function

$$bind^{\phi} : CTX \rightarrow RES_{\texttt{VAR\_ID}} \times RES_{\texttt{TERM}} \rightarrow CTX$$

specifies how the context is modified by a variable binding. (The ML implementation of this transformation scheme for the syntactic domain of transition rules is shown in Table 6.7.)

Besides the function *bind*, the specification of a context-dependent term transformation consists of the following functions f^{ϕ} (corresponding to the f^A in the Σ_G-homomorphism):

```
type ('name, 'patt, 'term, 'rule) RULE_TRANSF =
{ SkipRule     :'rule,
  UpdateRule   :('name * 'term) * 'term -> 'rule,
  BlockRule    :'rule list -> 'rule,
  CondRule     :'term * (unit -> 'rule) * (unit -> 'rule) -> 'rule,
  CaseRule     :'term * 'patt * (unit -> 'rule) * (unit -> 'rule) -> 'rule,
  DoForallRule :'patt * 'term * ('term -> 'term) * ('term -> 'rule) -> 'rule }

fun rule_transf ( h_name :NAME -> 'name,
                  h_patt  :PATT -> 'patt,
                  h_term  :TERM -> 'context -> 'term )
                ( bind :'context -> 'patt * 'term -> 'context )
                ( A :('name, 'patt, 'term, 'rule) RULE_TRANSF )
                ( R :RULE ) ( C :'context ) :'rule =
let val h_rule = rule_transf (h_name, h_patt, h_term) bind A
in case R of
     SkipRule => (#SkipRule A)
   | UpdateRule ((f,t),t') => (#UpdateRule A) ((h_name f,h_term t C), h_term t' C)
   | BlockRule Rs => (#BlockRule A) (map (λR.h_rule R C) Rs)
   | CondRule (G,R1,R2) =>
       (#CondRule A) (h_term G C, λ().h_rule R1 C, λ().h_rule R2 C)
   | CaseRule (t0,p,R1,R2) =>
       let val p_res = h_patt p
           val t0_res = h_term t0 C
           val R1_res = λ().h_rule R1 (bind C (p_res,t0_res))
           val R2_res = λ().h_rule R2 C
       in (#CaseRule A) (t0_res, p_res, R1_res, R2_res) end
   | DoForallRule (p,U,G,R) =>
       let val p_res = h_patt p
           val U_res = h_term U C
           val G_res = λx.h_term G (bind C (p_res,x))
           val R_res = λx.h_rule R (bind C (p_res,x))
       in (#DoForallRule A) (p_res, U_res, G_res, R_res) end
end
```

Table 6.7: Implementation of Context-Dependent Transformations

$$
\begin{aligned}
\texttt{VarTerm}^{\phi} &: CTX \to RES_{\texttt{VAR\_ID}} \to RES_{\texttt{TERM}} \\
\texttt{AppTerm}^{\phi} &: CTX \to RES_{\texttt{FUN\_ID}} \times RES_{\texttt{TERM}} \to RES_{\texttt{TERM}} \\
\texttt{TupleTerm}^{\phi} &: LIST(RES_{\texttt{TERM}}) \to RES_{\texttt{TERM}} \\
\texttt{CondTerm}^{\phi} &: RES_{\texttt{TERM}} \times RES_{\texttt{TERM}} \times RES_{\texttt{TERM}} \to RES_{\texttt{TERM}} \\
\texttt{LetTerm}^{\phi} &: RES_{\texttt{VAR\_ID}} \times RES_{\texttt{TERM}} \times RES_{\texttt{TERM}} \to RES_{\texttt{TERM}}
\end{aligned}
$$

Note that the context has to be explicitly handled in $\texttt{VarTerm}^{\phi}$ and $\texttt{AppTerm}^{\phi}$, as the transformation of these constructs may directly depend on the context. For the other constructs, instead, the context is propagated or modified in a standard way, as specified by the equations for $\phi_{\texttt{TERM}}$ above.

To illustrate the technique, consider term evaluation, as defined in the extended example of Sect. 6.4.1. The context consists of a $(state, environment)$ pair:

$$CTX \quad = \quad STATE \times ENV.$$

The result types for variable names, function names and terms are, respectively:

```
fun eval_bind (S,env) (p, x) = (S, env ++ p x)
fun eval_cond (G,e1,e2) = if G = TRUE then e1 () else e2 ()
fun eval_case (t0,p,e1,e2) = (p t0; e1 ()) handle FAIL => e2 ()

fun eval_comprehension (F,E) (e,p,SET range,G) =
  let fun F'(x,rest) = (if G x = TRUE then F(e x,rest) else rest) handle FAIL => rest
  in set_fold (F', E) range end
|   eval_comprehension _ _ = raise ERROR

fun eval_term (t :TERM) (S :STATE, env :ENV) :VALUE =
  term_transf (interpretation S, eval_patt) eval_bind {
    VarTerm   = λ(S,env).lookup env,
    AppTerm   = λ(f,t').f t',
    TupleTerm = TUPLE,
    CondTerm  = eval_cond,
    CaseTerm  = eval_case,
    SetCompr  = SET ∘ (eval_comprehension (set_add, set_empty))
  } t (S,env)

fun eval_rule (R :RULE) (S :STATE, env :ENV) : UPDATE_SET =
  rule_transf (id, eval_patt, eval_term) eval_bind {
    SkipRule   = map_empty,
    UpdateRule = op |->,
    BlockRule  = map_Union,
    CondRule   = eval_cond,
    CaseRule   = eval_case,
    DoForallRule = λ(p,U,G,R).( eval_comprehension (map_union, map_empty)
                                  (R,p,U,G) )
  } R (S,env)
```

Table 6.8: Context-Dependent Transformations — Evaluation

$$
\begin{aligned}
RES_{\texttt{VAR\_ID}} &= \texttt{VAR\_ID} \\
RES_{\texttt{FUN\_ID}} &= VALUE \to VALUE \\
RES_{\texttt{TERM}} &= VALUE.
\end{aligned}
$$

Assuming $\phi_{\texttt{VAR\_ID}}(ctx)(v) = v$ for all $v \in \texttt{VAR\_ID}$ and $\phi_{\texttt{FUN\_ID}}(S,\rho)(f) = \mathbf{f}_S$ for all $f \in \texttt{FUN\_ID}$, the semantics of terms can be specified as a context-dependent transformation as follows:

$$
\begin{aligned}
\texttt{VarTerm}^{\phi}\,(S,\rho)\,(v) &= \rho(v) \\
\texttt{AppTerm}^{\phi}\,(S,\rho)\,(f,\,t) &= f\,(t) \\
\texttt{TupleTerm}^{\phi}\,([\,t_1,\ldots,t_n\,]) &= (t_1,\ldots,t_n) \\
\texttt{CondTerm}^{\phi}\,(G,\,t_1,\,t_2) &= \begin{cases} t_1 & \text{if } G = \mathbf{true} \\ t_2 & \text{otherwise} \end{cases} \\
\texttt{LetTerm}^{\phi}(v,\,t_1,\,t_2) &= t_2
\end{aligned}
$$

where the binding mechanism is defined as

$$
bind^{\phi}\,(S,\rho)\,(v,x) \;=\; (S,\rho \oplus \{v \mapsto x\}).
$$

Table 6.8 shows a ML implementation of the evaluator for ASM-SL terms and rules based on the technique discussed here.

```
datatype ('name, 'patt) PATT' =
  VarPatt' of string
| AppPatt' of 'name * 'patt
| TuplePatt' of 'patt list
```

```
datatype PATT_NN = Patt_NN of (NAME, PATT_NN) PATT' * int

fun patt_nn (p :PATT) :PATT_NN =
  let fun nn (Patt_NN (p',n)) = n
  in patt_induction id {
       VarPatt   = λ var_id.Patt_NN (VarPatt' var_id, 1),
       AppPatt   = λ(f,p').Patt_NN (AppPatt' (f,p'), 1 + nn p'),
       TuplePatt = λ ps.Patt_NN (TuplePatt' ps, sum (map nn ps))
     } p
  end
```

Table 6.9: Polymorphic ASTs with an Example

6.4.3 Polymorphic ASTs

In the previous section we have seen two examples of program transformations of quite different nature, the first one (evaluation) mapping language phrases into objects of the corresponding semantic domains, the second one (substitution) transforming phrases into phrases of the same type. Other useful transformations often involve one of the following:

- annotating the abstract syntax trees with various information (example: a type checker which, in addition to determining whether the program is type-correct or not, annotates each AST node with the type of the corresponding phrase), or
- transforming ASTs into objects of another data type, which extends the abstract syntax in some way (example: a simplifier which replaces any constant term by its value—which is not a term—and leaves everything else as it is).

It is very tedious to define, for each transformation of this kind, the corresponding target data structures, as a large part of these data structures will consist of a repetition of the AST constructors. Therefore, the ASM-WB kernel offers a polymorphic version of the ASTs, which can be instantiated to implement any kind of annotated ASTs or to enrich the AST structure.

This technique is illustrated in Table 6.9. This time, for conciseness, we consider the case of patterns (the other cases are analogous). The upper part of the table shows the data type implementing polymorphic ASTs for patterns. The lower part is a simple example, which shows how to define and compute annotated ASTs, where the annotation of each AST node is the number of nodes in the subtree having that node as its root (an integer). The first line contains the type definition for the annotated AST `PATT_NN` (where `NN` stands for "number of nodes"): note that, by instantiating the type variable `'patt` of `PATT'` to `PATT_NN`, one implicitly defines a pair of mutually recursive data types

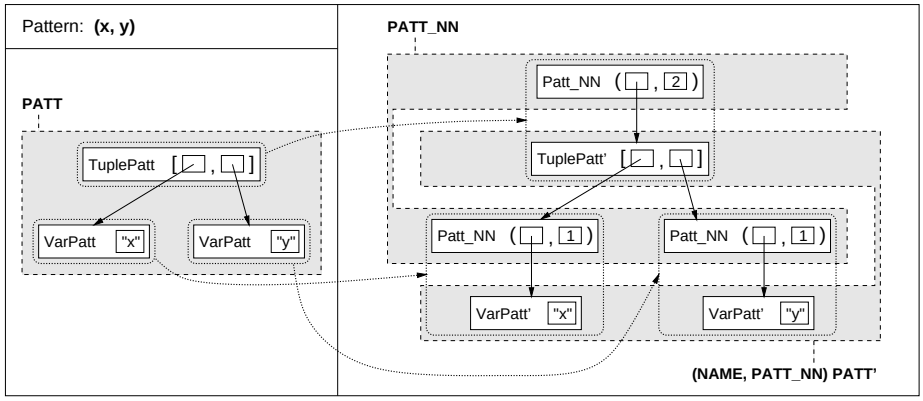


Figure 6.3: Example of Annotated AST (Annotation = Number of Nodes)

(namely, "`PATT_NN`" and "`(NAME, PATT_NN) PATT'`"). After the definition of the type `PATT_NN`, a function `patt_nn` is defined, which takes an ordinary AST of type `PATT` and annotates it appropriately: the result is an AST of type `PATT_NN`, where each node is annotated with the number of nodes in the corresponding subtree. Fig. 6.3 illustrates graphically the structure of annotated ASTs as well as the transformation of an ordinary AST (of type `PATT`) into an annotated AST (of type `PATT_NN`).

Both the structural induction and the context-dependent transformation schemes can be generalized to work with polymorphic ASTs. This generalization is quite straightforward, therefore we do not discuss it here. A non-trivial application of the transformation schemes for polymorphic ASTs will be presented in Chapter 7. As a part of a transformation which makes ASM-SL programs amenable to model-checking, a simplifier which replaces certain constant terms by values or locations is defined there. The basic data structure of the simplifier consists in the ASTs for terms and rules, enriched by extending terms to "partially evaluated terms" (which can be values, locations, or ordinary terms). This data structure and the related transformation are easily implemented by means of an appropriate instance of polymorphic ASTs (see Sect. 7.5.4 for details).

6.5 An Industrial Case Study

In this chapter, we introduced the basic ideas underlying the ASM Workbench and its architecture. The applications discussed in Chapters 2 and 9 show that both the language and the tools are appropriate for modelling quite different systems at quite different levels of abstraction.

Moreover, a first successful application of the ASM Workbench in an industrial context has been developed at the Munich Research Center of Siemens AG. Within the project "FALKO" [37] (dealing with a simulation system for the development and validation of train schedules) a prototype of the system

was developed in ASM-SL and tested using the ASM Workbench.

The size of the ASM-SL specification is quite considerable: the static part includes 323 function definitions, the dynamic part 113 transition rules (in fact, this is the largest known application of the ASM Workbench). The ASM-SL specifications—in conjunction with existing libraries implementing numerical computations—have been simulated (the link to the existing code has been realized by using oracles). A test system to be simulated included 16 stations, 150 signals, 60 switches, and 5 crossings.

Although first simulations performed using the Workbench's interpreter were quite slow[14] (mainly due to the large amount of interprocess communication generated by the oracle queries, needed because of the lack of a C interface to call the functions of the numeric library), a C++ code generator based on the ASM Workbench parser was later developed at Siemens, such that the same ASM-SL specifications could be simulated very efficiently.[15]

This seems to provide some experimental evidence that, despite some limitations of the tools which are currently part of the ASM Workbench, its language ASM-SL and its architecture constitute a good basis for developing ASM tools satisfying the requirements of realistic specification and modelling tasks.

6.6 Related Work

As mentioned in the introduction, while there are essentially no tools supporting static analysis, transformations, or verification of ASMs, several ASM simulators (interpreters or compilers) have been developed during the last years,

An early ASM simulator was developed by Angelica Kappel at Dortmund University [61]. It is based on a specification language called DASL (Dynamic Algebra Specification Language)[16], a typed specification language extending ASMs by a restricted form of multi-sorted equational specifications to define the static parts (conceptually, a similar approach as in ASM-SL). The interpreter is based on a simple abstract machine: DASL specifications are compiled into an intermediate code, which is then interpreted by the abstract machine. Both the compiler and the abstract machine are implemented in Prolog, and the user interface is basically the Prolog environment. Despite its nice features, such as the simple and clean specification language, this tool implements only a small subset of sequential deterministic ASMs, with syntax and semantics partially different from the present ASM definition (in fact, this tool was realized before the official ASM definition [45] appeared).

Another early ASM simulator is the Michigan interpreter [55], developed at the University of Michigan at Ann Arbor. This interpreter, implemented in C, has been updated over the years and (in its most recent version) includes the complete language of ASMs as defined in [45]. The specification language is

[14] About 24 hours simulation time for 4,5 hours train schedule.

[15] About 2-5 minutes for the same train schedule.

[16] At that time ASMs were known as *Dynamic Algebras*. Then the name was changed to *Evolving Algebras* (*EAs*), and finally to Abstract State Machines.

untyped and includes some constructs for the definition of the static part of ASM specifications by combining a rich set of built-in data structures and operations: however, this part of the language is somewhat unsystematic and not very expressive.[17] It is also possible to "customize" the interpreter, implementing additional ASM functions as C functions: however, this is not a practical alternative for the specification of static algebras, not only because of the low abstraction level of the C language, but also because each time a new version of the interpreter has to be built. The (textual) user interface is based on the Tcl shell and allows some degree of user interaction (inspection of the interpreter state and some of the usual features found in debuggers with text-based user interface). Jim Huggins also wrote a partial evaluator based on the Michigan interpreter [48]. Unfortunately, its development was abandoned quite early: the partial evaluator was never released.

Two further ASM simulators are *leanEA* ("A Lean Evolving Algebra Compiler") by Posegga and Beckert [6] and the EA interpreter by Dag Diesen [34]. They are similar in the sense that, in both of them, the implementation of ASM is embedded in the implementation language (Prolog for *leanEA*, Scheme in Diesen's interpreter), on which both tools rely for the definition of the static part of specifications. Both tools support only a subset of sequential deterministic ASMs. The user interface is provided by the Prolog and Scheme interactive environments, respectively.

A runtime system for the execution of ASMs—called ASM Virtual Architecture (ASM/VA)—was developed by Igor Đurđanović [36]. The ASM/VA is a C++ class library providing the basic mechanisms needed to make ASMs executable, to be used in combination with a code generator translating ASM specifications into C++ code containing appropriate ASM/VA calls. It is based on the C++ Standard Template Library (STL) and its primary aim is the efficient execution of ASMs.

A recent ASM simulator is the XAsm compiler developed by Matthias Anlauff (formerly known as *Aslan* [2]). The XAsm language—which is untyped—includes all the ASM transition rules defined in [45] and additional modularization constructs. For defining the static parts of a specification, XAsm offers the possibility of defining constructors. Universes and functions that are not built-in can be defined in C or in XAsm itself. XAsm generates C code, which is then linked to the run-time system and to the user-defined C functions. A special feature of XAsm, useful when using ASMs to define programming language semantics, is the possibility of integrating grammar productions in the ASM specification. The user interaction features are more advanced than in older tools: like in the ASM Workbench, a graphical debugger allows to control the flow of the simulation, inspect the ASM state, and so on.

Finally, the *AsmGofer* system developed by Joachim Schmid [80] provides an advanced ASM interpreter embedded in the well known functional programming environment "Gofer" (which is an implementation of Haskell). Conceptually,

[17] For instance, lists are provided as a built-in data structure, but there is no possibility of defining free types or constructors in general. There are some constructs to define static functions, but recursive definitions are not allowed.

AsmGofer and the ASM Workbench are quite similar, as both of them are based on a combination of Abstract State Machines and functional programming. On the one hand, the language of AsmGofer is more expressive than ASM-SL, as the Gofer type system extends the ML type system by the notion of "type classes" and functional programming can be used without any restriction in AsmGofer for the definition of types and functions (in particular, higher-order functions are allowed). Moreover, Graphical User Interfaces (GUIs) for AsmGofer programs can be implemented very easily with the TkGofer system [83]. On the other hand, AsmGofer can not be used for developing analysis or transformation tools, as the ASM-related parts of AsmGofer are deeply embedded in the Gofer system.

Chapter 7

Analysis Techniques for Finite-State ASMs

In order to support computer-aided verification of ASM-SL specifications by means of model checking, we developed an interface between the ASM Workbench and the symbolic model checker SMV [66], based on a transformation which maps a large subset of ASM into the SMV language. In this chapter we present the basic ideas of the transformation (which could also be used for other model-checkers), as well as its implementation in the Workbench.

The chapter has the following structure. After some motivating considerations (Sect. 7.1), we discuss the problem of model checking ASMs in general (Sect. 7.2). After that, the translation from ASM to SMV is presented in two stages: first we illustrate the translation from a special subset of ASM called ASM_0 to SMV (Sect. 7.3), then we show how an arbitrary finite-state ASM can be reduced to ASM_0, such that the first translation scheme can then be applied (Sect. 7.4). Then we describe the ASM2SMV tool, an extension of the ASM Workbench where all the techniques of this chapter are put together (Sect. 7.5, which also includes a discussion of implementation issues as well as an example). The chapter is concluded by references to related work (Sect. 7.6).

7.1 Motivation

Although the use of formal specification alone can already provide some benefits, especially in the software domain, support for computer-aided verification is crucial to gain even more practical advantages from the use of formal methods [26]. This is particularly true in hardware and embedded systems design, where current formal verification techniques are usually applicable, as such systems are in general much less complex—and often more safety-critical—than traditional software systems (such as applications running on PCs or distributed information systems running on computer networks).

For Abstract State Machines, systematic tool support for computer-aided

verification is still missing: neither verification tools specifically designed for ASMs nor transformation tools mapping ASM specifications into the logic of available verification systems are available. This does not mean that there are no approaches to mechanical verification of ASMs (see, for instance, [79, 86, 90]), but these approaches have not been mechanized: in all the cited works the necessary transformations were applied by hand.

As a means for the analysis and validation of ASM specifications, we consider the use of model checking. This choice is motivated by the fact that model checking, in the field of hardware and embedded systems design, gains more and more acceptance, such that it even begins to be used routinely in industry. Moreover, the transition system semantics on which most model-checkers are based is very close to the computational model of ASMs, such that using model checking for verification of ASMs seems to be very natural. On the other hand, providing model checking support for ASMs supplies existing model checkers, such as the supported SMV system, with a higher-level modelling language (at the price of a possible lost of efficiency).

Obviously, verification of ASMs by means of model checking is not possible in general, since ASM specifications define transition systems with a possibly infinite number of states, while model checkers can only deal with finite-state systems. Therefore, we have to consider a special class of *finite-state ASMs*. To this purpose, we extended ASM-SL by a language feature called *finiteness constraints*, which allows to restrict the state space by constraining the function ranges (i.e., by small local modifications which only affect the function declarations). In this way, finite instances of the system under consideration—to be verified by model checking—can be obtained without additional effort. In particular, there is no need to create two different versions of the system description, the one for specification and the other for verification purposes.

Although the verification of such finite instances does not correspond—in general—to a complete correctness proof, it is a much more effective way of debugging than the simulation of a few isolated test cases: essentially, it represents a systematic form of testing, which is very helpful in the early design phase for the validation of *ground models*, i.e., models representing the first formalization of a system. As a part of the validation process, the counterexamples possibly found by the model-checker can be fed in the ASM simulator and inspected in order to find out the origin of the erroneous behaviour and fix the problem. (See Chapters 8 and 9 for a more detailed methodological discussion.)

The choice of SMV as verification tool was motivated by several reasons: it is freely available, well documented, and continuously maintained and improved. Moreover, the first approach to model checking ASMs [86] was developed using SMV. Note, however, that all the techniques discussed in this chapter, except for the translation step from ASM_0 to SMV explained in Sect. 7.3 (which is derived from [86] and is quite SMV-specific), are independent from the choice of a particular model checker, thus constituting a sort of "front-end". Replacing the SMV "back-end" of Sect. 7.3 should allow to interface other verification systems as well.

7.2 ASMs and Model Checking

In this section, we discuss from a general perspective the problems which have to be solved to make ASMs amenable to model checking. Although we always refer to SMV throughout the discussion, most of the considerations below also apply to other languages and systems for temporal model checking[1], of which SMV is a typical representative.

7.2.1 CTL Model Checking and the SMV Tool

In order to make the presentation of this chapter self-contained, we recall here a few basic notions of CTL model checking and briefly review the main features of the SMV model checker. The presentation given here follows [66] and [25].

Models of System Behaviour

The system models to be checked are represented by so-called *Kripke structures*, a particular kind of transition systems.

Let AP be a set of atomic propositions. A Kripke structure M over AP is a 4-tuple $M = (S, S_0, R, L)$ where:

1. S is a finite set of states.
2. $S_0 \subseteq S$ is the set of initial states.
3. $R \subseteq S \times S$ is a transition relation (which must be total, i.e., for every state $s \in S$ there is a state $s' \in S$ such that $(s, s') \in R$).
4. $L : S \to \mathcal{P}(AP)$ is a function that labels each state with the set of atomic propositions true in that state ("*valuation*").

A *path* in M from a state s is an infinite sequence of states $\pi = s_0 s_1 s_2 \ldots$ such that $s_0 = s$ and $(s_i, s_{i+1}) \in R$ for all $i \geq 0$.

Note that, equivalently, the set of atomic propositions AP can be replaced by a set V of state variables ranging over finite sets. In fact, on the one hand, atomic propositions are a special case of state variables (*boolean* state variables); on the other hand, each non-boolean state variable x with range X can be replaced by atomic propositions $x = x_i$ for each $x_i \in X$. In this way, an arbitrary finite transition system can be easily reduced to a Kripke structure.[2] An example, where $V = \{\text{reset} : \{0, 1\}, \text{ctr} : \{0, 1, 2\}\}$, is shown in Fig. 7.1.

[1] *Temporal model checkers* verify that systems (modelled as finite transition systems) satisfy properties (specified as formulae of a temporal logic). The other class of model checkers is that of *equivalence checkers*, which test whether two systems (specified as automata) have equivalent behaviour. Temporal model checkers are useful to verify that a specification satisfies a set of requirements, while equivalence checkers are useful to verify correctness of implementations.

[2] In doing this, a further technical detail has to be considered: for those states s which have no successor in the transition system (final states), the transition relation R of the Kripke structure must be extended by a self-edge $(s, s) \in R$ in order to make it total.

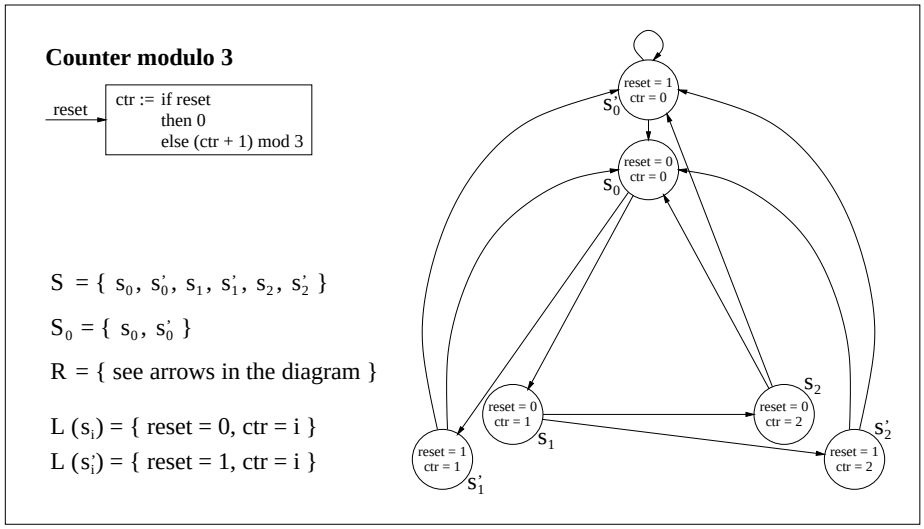


Figure 7.1: Counter modulo 3: Kripke Structure (State Transition Graph)

Specification of Properties: CTL

The properties to be proved about the system are formalized in a temporal logic, in the case of SMV in CTL (Computation Tree Logic).

Conceptually, CTL formulae describe properties of computation trees, hence the name of the logic. A computation tree can be obtained by fixing a given initial state s_0 in a Kripke structure and then unwinding the structure into an infinite tree with s_0 as its root, as shown in Fig. 7.3. The computation tree shows all the possible execution paths starting from s_0. Conversely, if the number of reachable states in a computation tree is finite, the tree can be folded into a Kripke structure by identifying nodes corresponding to the same state. In this sense, computation trees and Kripke structures are equivalent representations for finite transition systems.

The syntax of CTL formulae is as follows.

1. Every atomic proposition is a CTL formula.
2. If ϕ and ψ are CTL formulae, then so are:
   - $\neg\phi$, $(\phi \wedge \psi)$, $(\phi \vee \psi)$,
   - $\mathbf{AX}\,\phi$, $\mathbf{EX}\,\phi$,
   - $\mathbf{AF}\,\phi$, $\mathbf{EF}\,\phi$,
   - $\mathbf{AG}\,\phi$, $\mathbf{EG}\,\phi$,
   - $\mathbf{A}\,(\phi\,\mathbf{U}\,\psi)$, $\mathbf{E}\,(\phi\,\mathbf{U}\,\psi)$.

Note that, in CTL, a *path quantifier* **A** ("for all computation paths") or **E** ("for some computation path") always precedes a *temporal operator* **X**, **F**, **G**, or

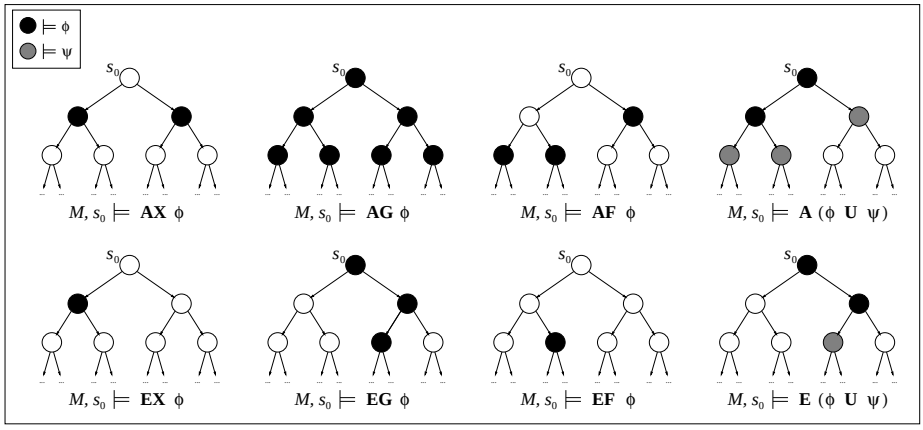


Figure 7.2: CTL Operators

U, describing properties of a path through the computation tree. The temporal operators have the following meaning:

- **X** ("next time") requires that a property holds in the second state of the path;
- **F** ("eventually" or "in the future") requires that a property will hold at some state on the path;
- **G** ("always" or "globally") specifies that a property holds at every state on the path;
- **U** ("until") combines two properties by requiring that there is a state on the path where the second property holds, and, at every preceding state on the path, the first property holds.

The validity of a given CTL formula is always relative to a given model M and state s: thus, one writes $M, s \models \phi$ if a CTL formula holds at state s in the Kripke structure M. Formally, the semantics of CTL is defined as follows.

$M, s_0 \models p$	iff	$p \in L(s_0)$
$M, s_0 \models \neg\phi$	iff	$M, s_0 \not\models \phi$
$M, s_0 \models \phi \vee \psi$	iff	$M, s_0 \models \phi$ or $M, s_0 \models \psi$
$M, s_0 \models \phi \wedge \psi$	iff	$M, s_0 \models \phi$ and $M, s_0 \models \psi$
$M, s_0 \models \mathbf{AX}\,\phi$	iff	for all paths $\pi = s_0 s_1 s_2 \ldots$ from s_0, $M, s_1 \models \phi$
$M, s_0 \models \mathbf{EX}\,\phi$	iff	for some path $\pi = s_0 s_1 s_2 \ldots$ from s_0, $M, s_1 \models \phi$
$M, s_0 \models \mathbf{A}\,(\phi\,\mathbf{U}\,\psi)$	iff	for all paths $\pi = s_0 s_1 s_2 \ldots$ from s_0, there is a $k \geq 0$ such that $M, s_k \models \psi$ and for all j with $0 \leq j < k$, $M, s_j \models \phi$
$M, s_0 \models \mathbf{E}\,(\phi\,\mathbf{U}\,\psi)$	iff	for some path $\pi = s_0 s_1 s_2 \ldots$ from s_0, there is a $k \geq 0$ such that $M, s_k \models \psi$ and for all j with $0 \leq j < k$, $M, s_j \models \phi$.

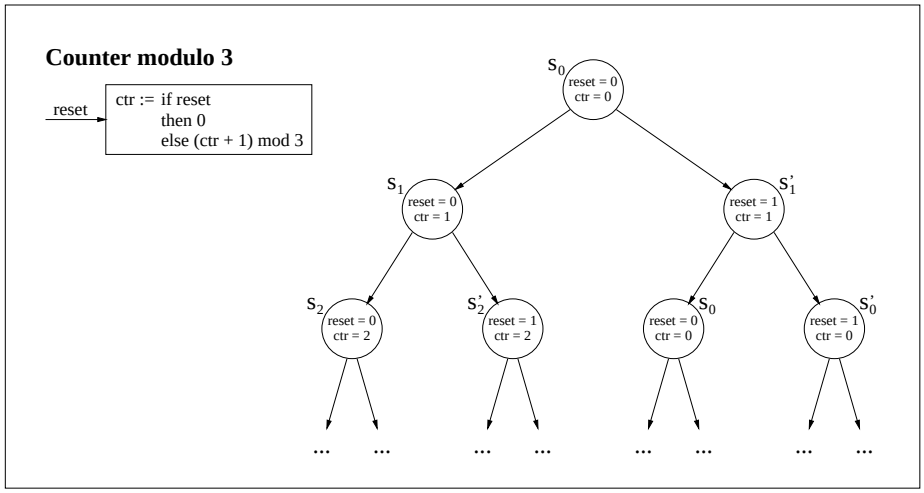


Figure 7.3: Counter modulo 3: Computation Tree

The other operators can be defined in terms of the above, in particular:

$$\begin{array}{lcl} \mathbf{AF}\,\phi & = & \mathbf{A}\,(\mathrm{true}\,\mathbf{U}\,\phi) \\ \mathbf{EF}\,\phi & = & \mathbf{E}\,(\mathrm{true}\,\mathbf{U}\,\phi) \\ \mathbf{AG}\,\phi & = & \neg\,\mathbf{EF}\,(\neg\phi) \;=\; \neg\,\mathbf{E}\,(\mathrm{true}\,\mathbf{U}\,\neg\phi) \\ \mathbf{EG}\,\phi & = & \neg\,\mathbf{AF}\,(\neg\phi) \;=\; \neg\,\mathbf{A}\,(\mathrm{true}\,\mathbf{U}\,\neg\phi) \end{array}$$

An Example

As an example, consider a simple synchronous circuit implementing a counter modulo 3. What this hardware component does is simply to increase the value of its internal register *ctr* (modulo 3) or to reset it to 0 in case the input signal *reset* is 1. The corresponding Kripke structure is shown in Fig. 7.1, and the computation tree resulting from unfolding it starting at state s_0 in Fig. 7.3.

Some properties which one may want to prove about the behaviour of this system, and their corresponding CTL formalizations, are the following.

1. If the *reset* signal is high in a given state, then the value of the counter must be 0 in any state immediately following that state:

   $\mathbf{AG}\,(\mathrm{reset} = 1 \Rightarrow \mathbf{AX}\,(\mathrm{ctr} = 0))$.

2. At some point the value of the counter will become 2:

   $\mathbf{AF}\,(\mathrm{ctr} = 2)$.

3. If no reset is ever effected, then the counter will eventually assume all of its admissible values:

   $(\mathbf{AG}\,(\mathrm{reset}{=}0)) \Rightarrow (\mathbf{AF}\,(\mathrm{ctr}{=}0) \wedge \mathbf{AF}\,(\mathrm{ctr}{=}1) \wedge \mathbf{AF}\,(\mathrm{ctr}{=}2))$.

```
MODULE main
VAR
  reset : { 0, 1 };
  ctr   : { 0, 1, 2 };
ASSIGN
  init(ctr) := 0;
  next(ctr) := case
                 reset : 0;
                 TRUE  : (ctr + 1) mod 3;
               esac;

SPEC  AG (reset = 1 -> AX (ctr = 0))
SPEC  AF (ctr = 2)
SPEC  AG (reset = 0) -> (AF (ctr = 0) & AF (ctr = 1) & AF (ctr = 2))
```

Table 7.1: Counter modulo 3: SMV Input (Model and Specification)

The SMV Tool

The *model checking* problem is the problem of establishing whether, given a Kripke structure $M = (S, S_0, R, L)$ (the *model*) and a CTL formula ϕ (the *specification*), $M, s_0 \models \phi$ holds for all initial states $s_0 \in S_0$. A tool implementing an algorithm to solve this decision problem is called a *model checker*. A nice feature of model checkers is that, if a given property is not satisfied by the model, i.e., $M, s_0 \not\models \phi$ for some initial state s_0, they are able to provide a *counterexample*, i.e., a path $\pi = s_0 s_1 s_2 \ldots$ in M which falsifies ϕ.

The SMV tool is a model checker based on a technique called *symbolic model checking* [66]. The basic idea of this technique is to represent sets of states as well as the transition relation by means of *binary decision diagrams* (BDDs for short), such that it is not necessary to explicitly construct the Kripke structure by enumerating all the states. Efficient BDD-based algorithms can be used instead. Although the model checking problem is still, in general, NP-complete, symbolic model checking performs better than enumerative methods in many applications and allows the to verify larger systems [66, 26, 24].

The *input* to SMV consists of a model description (sometimes called simply "model" in SMV parlance) and a set of CTL formulae to be proved about this model (the "specification"). The model description is to be written in an SMV-specific language (called SMV language or simply SMV) and is quite close, at least in the style, to an hardware description at the register-transfer level.[3] The *output* produced by SMV is either a response of the form "specification ϕ is true" or a counterexample, i.e., a path that falsifies the specification.

As an example, Table 7.1 shows the SMV input for the Counter modulo 3, consisting of the model description followed by the CTL properties to be proved. The example illustrates the typical structure of an SMV program, which consists of a set of *modules* (in this example only the main module).

[3]Note that model checkers used in industry are based on the same principles, but accept standard hardware description languages such as VHDL or Verilog as input languages.

Each module contains:

- *Declarations* of its state variables with the corresponding ranges, introduced by the keyword `VAR`. The allowed ranges are:
  - `boolean`;
  - finite sets of integers, including intervals;
  - finite sets of symbolic names (*atoms*), i.e., enumerated types.
- *Assignments* of the state variables, introduced by the keyword `ASSIGN`. Assignments are of two types:
  - `init`-assignments, specifying the value of a given state variable in the initial states;
  - `next`-assignments, specifying the value that a given state variable assumes in the next state in terms of the values of the state variables in the current state.

  Note that, taken as a whole, `init`-assignments define the set of initial states of the model, while `next`-assignments define the transition relation. The right-hand side of an assignment can be an arbitrary SMV *expression*, constructed from state variables, constants, and the built-in SMV operators, which include logical connectives, integer arithmetics, and the conditional `case` expression (equivalent to "`if` ... `elseif` ..." in ASMs).

  Note also that, if no assignments are specified for a given state variable (as in the case of *reset* in our example), that state variable may assume all values in its declared range. This is a way of introducing non-determinism in the models, which corresponds to the use of external functions in ASMs.
- The *specification*, i.e., the CTL properties to be verified about the module behaviour, introduced by the keyword `SPEC`.

The SMV language also provides some other important features, such as the possibility of defining *fairness constraints*. However, we are not going to discuss them, as we are concerned here only with the constructs involved in the ASM transformation presented in this chapter.[4]

Finally, to give the reader an impression of what SMV actually does, we show in Table 7.2 the output produced by SMV for the input of Table 7.1. The following can be observed:

- the first and third property could be proved true;
- the second property, $\mathbf{AF}\,(\mathrm{ctr} = 2)$, is proved false by the execution path $s'_0 s'_0 s'_0 \ldots$ (counterexample). Intuitively, this can be interpreted as follows: the value of the counter can not become 2, if a *reset* continues to occur before the counter can reach 2. The property would be trivially true, if we had a path quantifier **E** instead of **A**: *there are* paths, e.g. $s_0 s_1 s_2 \ldots$, where the counter reaches 2, but this does happen in *all* possible runs.

[4]The interested reader can consult [66] for details about SMV.

```
-- specification AG (reset = 1 -> AX ctr = 0) is true

-- specification AF ctr = 2 is false
-- as demonstrated by the following execution sequence
-- loop starts here --
state 1.1:
reset = 1
ctr = 0

state 1.2:

-- specification AG reset = 0 -> AF ctr = 0 & AF ctr = 1 ... is true

resources used:
user time: 0.01 s, system time: 0 s
BDD nodes allocated: 324
Bytes allocated: 1245184
BDD nodes representing transition relation: 11 + 4
```

Table 7.2: Counter modulo 3: SMV Output

7.2.2 ASM versus SMV

ASM specifications, like model descriptions in SMV, are specifications of transition systems (see, for instance, Table 7.3). Thus, an ASM specification corresponds—from a conceptual point of view—to an SMV model. However, ASMs differ from SMV in several important respects.

1. **Infinite state space**. Abstract State Machines define, in general, systems with a possibly infinite number of states, as a consequence of the fact that both the number of locations and the location ranges may be infinite. Model checkers, instead, can only verify transition systems with finitely many states, where a state usually consists of a finite number of state variables, each of which can assume values from a (fixed) finite set.

2. **Specification of transitions**. The way of specifying transitions in ASMs and SMV is different. In SMV, transitions are specified by means of **next**-assignments, which unambiguously define, for a given state variable, the value assumed by this variable in the next state. In ASMs, instead, there may be many updates of a given dynamic function f scattered throughout the program. This leads to the possibility that a location of f is updated more than once, possibly resulting in a conflict, or that it is not updated at all, which results in leaving its value unchanged (by the ASM semantics). Compare, for instance, the way the counter *ctr* of our example is updated in the SMV version of Table 7.1 and in the ASM version of Table 7.3.

3. **Functions instead of state variables**. The ASM notions of *dynamic function* and *external function* generalize the notion of *state variable* of basic transition systems, as found in SMV and other verification tools. State variables correspond to *nullary* external or dynamic functions (such as *reset* and *ctr* in the example), while n-ary dynamic/external functions

```
external function reset : INT
with reset in { 0, 1 }

dynamic function ctr : INT
with ctr in { 0, 1, 2 }
initially 0

transition Program ==
  if reset
  then ctr := 0
  else ctr := (ctr + 1) mod 3 endif
```

Table 7.3: Counter modulo 3: ASM-SL Specification

with $n > 0$ give rise to a (potentially infinite) set of *locations*.[5] For example, in the instruction set model of Sect. 2.2, a unary dynamic function `mem : INT -> INT` is used to represent the memory contents, such that for each x there is a distinct location $\texttt{mem}(x)$, which can be individually read/written like a state variable.

4. **Modelling of non-determinism.** The ASM language provides external functions as a means of specifying environmental influences on the system behaviour, which lead to non-deterministic runs. The SMV counterpart of external functions are state variables with no assignments ("unrestricted" state variables, such as *reset* in the Counter modulo 3 example). However, there is no SMV construct corresponding directly to the ASM `choose` rule for non-deterministic choice.

The above discrepancies must be overcome in order to use SMV for the verification of properties of ASMs. The first issue (infinite state space) is solved by introducing *finiteness constraints*, as explained in Sect. 7.2.3 below. The second and the third issues (specification of transitions and n-ary functions) are addressed by the transformation schemes of Sect. 7.3 and 7.4, respectively.

With respect to the fourth issue (modelling of non-determinism), `choose` rules are not directly supported by our approach. However, `choose` rules can always be replaced by external functions for arbitrary choice of a value, by means of a simple transformation resembling skolemization of first-order formulae. For example, let A_i be terms of type $SET(T_i)$, $i = 1, 2, 3$, and $f_x : T_1$, $f_z : T_2 \to T_3$ external functions with $f_x \in A_1$ and $f_z(y) \in A_3$ for each $y \in A_2$, respectively. Then the following two rules are equivalent:

$$\begin{array}{l}\texttt{choose } x \texttt{ in } A_1 \\ \quad \texttt{do forall } y \texttt{ in } A_2 \\ \quad\quad \texttt{choose } z \texttt{ in } A_3 \\ \quad\quad\quad a(x, y, z) := x + y + z\end{array} \quad \cong \quad \begin{array}{l}\texttt{do forall } y \texttt{ in } A_2 \\ \quad a(f_x, y, f_z(y)) := f_x + y + f_z(y)\end{array}$$

[5] The notion of *location* is an ASM peculiarity (see Sect. 1.1.3). Conceptually, a location corresponds to a state variable. Note that nullary external/dynamic functions contain exactly one location. Thus, they are equivalent to state variables.

(Note that the requirements $f_x \in A_1$ and $f_z(y) \in A_3$ above, which are necessary to ensure the equivalence of the two rules, are examples of finiteness constraints.)

7.2.3 Finiteness Constraints

In order to ensure that the ASM programs to be translated into SMV define systems with a finite state space, the user has to specify, for each dynamic or external function $f : \tau_1 * \ldots * \tau_n \rightarrow \tau$, a finiteness constraint of the form $f(v_1, \ldots, v_n) \in t[v_1, \ldots, v_n]$, where $t : SET(\tau)$ is a term denoting a finite set, possibly depending on the arguments of f (see Fig. 7.5 for some examples).

The concrete ASM-SL syntax for external and dynamic function definitions is extended to include finiteness constraints as follows:

`external function` f `:` [τ_1 `*` ... `*` τ_n `->`] τ
[`with` f [`(` v_1 `,` ... `,` v_n `)`] `in` t]

`dynamic function` f [`:` [τ_1 `*` ... `*` τ_n `->`] τ]
[`with` f [`(` v_1 `,` ... `,` v_n `)`] `in` t]
`initially` *fexpr*

Note that finiteness constraints are optional. In fact, they are needed in order to successfully translate ASM into SMV, but not (for instance) to type-check or execute an ASM.

For *external functions*, finiteness constraints correspond to *assumptions* about the *environment behaviour*. Such assumptions are expressed, in the resulting SMV model, by the range of the state variables corresponding to the single locations of the given external function. Those state variables can assume, in every moment, exactly the values specified by their range. For instance, in Table 7.3, the environment assumption expressed by the finiteness constraint on *reset* is simply that the value of *reset* can be either 0 or 1.[6]

For *dynamic functions*, finiteness constraints correspond to *requirements* on the *system behaviour*. In every state, the values of the locations of a given dynamic function, resulting from its initialization and the subsequent updates, are required to stay within the bounds specified by its finiteness constraints. This means that it must be proved that the constraints are not violated by any run. Thus, the finiteness constraints on f result in the SMV model in proof obligations, which we call *range conditions* and are of the form

$$\mathbf{AG}\,((f_l = x_1) \vee (f_l = x_2) \vee \ldots \vee (f_l = x_n)),$$

where f_l is a state variable corresponding to a location l of f and $\{x_1, \ldots, x_n\}$ are the admissible values for f_l, according to the finiteness constraints on f. In many cases the range conditions are trivially true and can be discarded by a simple static analysis of the rules, such that their time-consuming proof by model-checking can be avoided.

[6] In some situations, one may want to formulate more complex assumptions about the environment, e.g., that under certain conditions a given input does not change. In some verification systems this can be done by specifying constraints as formulae of linear temporal logic restricting the possible environment behaviours (seen as state sequences). However, as SMV does not support this kind of constraints, we did not consider including them in ASM-SL.

7.3 The Basic Translation Scheme

When comparing the SMV and ASM versions of our example (Tables 7.1 and 7.3, respectively), one can observe that they are quite similar. The reason is that all the external and dynamic functions occurring in the ASM specification are nullary and, as such, correspond one-to-one to SMV state variables. Moreover, all data types and static functions used in the ASM specification (integers with operations `+` and `mod`) are available in SMV as primitive operators, such that the ASM terms built out of them correspond immediately to SMV expressions. The only observable difference regards the way how the initial states and the transition relation are specified (see also point 2. of Sect. 7.2.2, "Specification of transitions").

This discrepancy can be overcome by means of a simple translation scheme, which was first introduced in [86] and can be applied to transform into SMV a subset of ASM restricted to:

- *basic transition rules*, i.e., skip, update, block, and conditional rules (but not `do-forall` rules);
- *nullary* external and dynamic functions;
- *integers*, *booleans* and *enumerated types* as data structures;
- static functions corresponding to *primitive SMV operators* (boolean connectives, simple integer arithmetics, and conditionals).

We call this subset ASM_0, as it only admits functions of arity 0 (except for a few static functions). The content of Table 7.3 is an example of ASM_0 program.

For ASM_0, the translation into SMV is very close. Dynamic and external functions, which are nullary, correspond one-to-one to SMV state variables. Thus, the declaration part (`VAR`) in the SMV model simply consists of variable declarations, with ranges given by the finiteness constraints in the ASM-SL specification. In our example:

```
reset : { 0, 1 };
ctr   : { 0, 1, 2 };
```

The translation of an ASM term t into an SMV expression $\mathcal{C}[\![t]\!]$ is trivial:

1. Every occurrence of a dynamic or external function is translated into the corresponding state variable.
2. ASM data values are mapped one-to-one to SMV constants (boolean constants, integer constants, or atoms).
3. Applications of static functions are translated into applications of the corresponding primitive operators of SMV.

Finally, the assignment part (`ASSIGN`) has to be defined. For each dynamic function f, an `init`-assignment and a `next`-assignment are generated. The

$$
\begin{array}{l}
[\![\texttt{skip}]\!] = \textit{(empty block)} \\
[\![f := t]\!] = \texttt{if } \textit{true} \texttt{ then } f := t \\
[\![R_1 \ldots R_n]\!] = [\![R_1]\!] \ldots [\![R_n]\!] \\
[\![R_T]\!] = \begin{cases} \texttt{if } G_T^1 \texttt{ then } R_T^1 \\ \ldots \\ \texttt{if } G_T^n \texttt{ then } R_T^n \end{cases} \\
[\![R_F]\!] = \begin{cases} \texttt{if } G_F^1 \texttt{ then } R_F^1 \\ \ldots \\ \texttt{if } G_F^m \texttt{ then } R_F^m \end{cases}
\end{array}
\Rightarrow [\![\texttt{if } G \texttt{ then } R_T \texttt{ else } R_F]\!] = \begin{cases} \texttt{if } G \wedge G_T^1 \texttt{ then } R_T^1 \\ \ldots \\ \texttt{if } G \wedge G_T^n \texttt{ then } R_T^n \\ \texttt{if } \neg G \wedge G_F^1 \texttt{ then } R_F^1 \\ \ldots \\ \texttt{if } \neg G \wedge G_F^m \texttt{ then } R_F^m \end{cases}
$$

Table 7.4: Rule-flattening Transformation

`init`-assignment is of the form

```
init( f ) := C[[ t0 ]]
```

where t_0 is the term following the `initially` clause in the ASM-SL definition of f. The generation of the `next`-assignments is slightly more involved and is performed in two steps.

1. The ASM program P is transformed into an equivalent ASM program P' consisting only of a block of *guarded updates* (rules of the form "`if` G `then` $f := t$") through the "flattening" transformation of Table 7.4.[7]

2. For each dynamic function f, all guarded updates of f (i.e., all guarded updates of the form "`if` G_i `then` $f := t_i$") are collected from P'. In this way a pair $(f, \{(G_1, t_1), \ldots, (G_n, t_n)\})$, which associates f to a set of pairs (*guard*, *update-rhs*), is obtained for each f.

The pair $(f, \{(G_1, t_1), \ldots, (G_n, t_n)\})$ obtained for each dynamic function f after step 2 is then translated into a `next`-assignment of the form

```
next( f ) :=
   case
    C[[ G1 ]]   :   C[[ t1 ]]
          ...
    C[[ Gn ]]   :   C[[ tn ]]
      TRUE      :   f
   esac;
```

Note that a default case is added at the end, in order to specify that the value of f remains unchanged if none of the guards is satisfied. Moreover, it must be observed that the above translation is correct only if the ASM never produces conflicting updates, i.e., it never updates the same location to different values in the same computation step (this issue is discussed in detail below, under "No-conflict conditions").

[7] It could be proved by a trivial structural induction that the result of the flattening transformation: *(i)* is a block of guarded updates, and *(ii)* is equivalent to the original rule.

In our example, the `ASSIGN` section of the SMV model resulting from translating the ASM of Table 7.3, will be as follows. The initialization of `ctr` is obviously given by:

```
init(ctr) := 0;
```

The computation of the `next`-assignments will go through the following steps:

1. *Flattening*: according to Table 7.4, the ASM program `Program` is transformed into the block of guarded updates

   `if` $reset \wedge true$ `then` ctr `:=` 0 `endif`
   `if` $\neg reset \wedge true$ `then` $ctr := (ctr + 1) \bmod 3$ `endif`.

2. *Collecting guarded updates* by function: both guarded updates above are updates of ctr, this step yields therefore one pair

   $(ctr, \{ (reset, 0), (\neg reset, (ctr + 1) \bmod 3) \})$

   (after simplification of the guards[8]).

Thus, the resulting `next`-assignment will be:[9]

```
next(ctr) :=
  case
    reset  : 0;
    !reset : (ctr + 1) mod 3;
    TRUE   : ctr;
  esac;
```

Note that the generated assignment is slightly redundant, but equivalent to the one of Table 7.1. The problem of conflicting updates mentioned above does not arise in this case, as the guards `reset` and `!reset` are mutually exclusive.

No-conflict conditions In general, it is not always the case that guards of updates of the same function are disjoint. To prevent the mentioned problem of conflicting updates, appropriate proof obligations must be generated. We call them *no-conflict conditions*. For example, consider the following ASM program:

```
if a = 0 then a := 10 endif
if a >= 0 then a := a - 1 endif
```

As this program has already the form of a block of guarded updates, the `next`-assignment resulting from its translation will be:

```
next(a) :=
  case
    a = 0  : 10;
    a >= 0 : a - 1;
    TRUE   : a;
  esac;
```

[8] In fact, in the actual implementation of the translation schemes, some rewriting rules—such as $\alpha \wedge true \rightsquigarrow \alpha$—are applied in order to simplify the transformation results.

[9] The `!` operator has the meaning of negation in SMV.

However, this translation is not correct. In a state S with $S \models a = 0$ (such a state is reachable, unless $a < 0$ in all initial states), the original ASM program will produce conflicting updates $a := 10$ and $a := -1$ and abort. The SMV model, instead, will simply update a to 10 and continue (due to semantics of the SMV `case`-expression, which corresponds to "`if...elseif...`").

As already mentioned, the translation scheme introduced above is correct only under the condition that the source ASM is free of such conflicts. Instead of modifying the translation scheme by encoding conflicts into the SMV model (e.g., by adding error variables), which would introduce unnecessary overhead, we simply consider an ASM leading to conflicts as a "wrong" specification and reject it. This can be done by generating, for each dynamic function f, a special *no-conflict condition*

$$\mathbf{AG}\,(\bigwedge_{i \neq j}(G_i \wedge G_j \Rightarrow t_i = t_j)),$$

where $i, j \in \{1, \ldots, n\}$ and $(f, \{(G_1, t_1), \ldots, (G_n, t_n)\})$ is the result of collecting the guarded updates of f. Such proof obligations are appended to the SMV specification, in addition to the range conditions discussed in Sect. 7.2.3 and to the CTL system properties specified by the user, and must be verified by the model checker.[10] If they are verified, the translation is guaranteed to be correct. Otherwise, the counterexamples show how conflicts arise; the user should then correct the original ASM specification to eliminate the conflicts.

For example, the no-conflict condition for the state variable a in the program above is $\mathbf{AG}\,(a = 0 \wedge a \geq 0 \Rightarrow 10 = a - 1)$, which is obviously equivalent to $\mathbf{AG}\,(a \neq 0)$. This formula is not satisfied by the model for any initial state with $a \geq 0$. The user should modify the ASM specification, e.g., by changing the guard of the second conditional to $a > 0$, in order to eliminate the conflict.

7.4 The Extended Translation Scheme

The ASM_0 subset treated in the previous section is clearly very restrictive. In fact, only very few of the existing ASM specifications are covered by ASM_0, as this subset excludes—among other things—the most typical feature of ASMs, the possibility of updating a function interpretation at a given point by an update rule of the form $f(t_1, \ldots, t_n) := t$.

In this section we introduce another translation scheme[11], by which almost any (finite-state) ASM can be reduced to ASM_0. The resulting ASM_0 programs can then be translated into SMV by means of the basic translation scheme of the previous section. The extended translation scheme supports:

- basic transition rules and `do-forall` rules;

[10] In the actual implementation of the translation scheme, some steps are taken to avoid the unnecessary generation of no-conflict conditions, which are quite expensive, as their size is quadratic in the number of guards. For instance, if updates of the same function f are found in different branches of a conditional rule (such as the updates of *ctr* in the Counter modulo 3 example), they are never conflicting and thus no proof obligation needs to be generated.

[11] This translation scheme has been published in [33].

- external and dynamic functions of arbitrary arity;
- all data structures available in ASM-SL, including lists, finite sets, finite maps, and user-definable free types;
- arbitrary static functions.

Not supported are `choose` rules, which can however be replaced manually by external choice functions before applying the transformation (as explained in Sect. 7.2.2), and derived functions, which can be eliminated by macro-expansion, unless their definition is recursive (see "Preprocessing" in Sect. 7.5.1 below).

7.4.1 Motivation

The main problem in reducing arbitrary (finite-state) ASMs to basic transition systems, as described for instance by ASM_0, is the following.

Let f be an external or dynamic function. If f is nullary, then f contains one location and corresponds to a state variable (essentially, it *is* a state variable). On the contrary, a function $f : T_1 \dots T_n \to T$, with $n > 0$, consists of a set of locations $\{(f, \overline{x}) \mid \overline{x} \in \mathcal{T}_1 \times \dots \times \mathcal{T}_n\}$, each of which carries its own value and can be individually read (and, if f is dynamic, also written) like a state variable. Thus, it is natural to map each location to a state variable in the SMV model.

However, a term $t \equiv f(t_1, \dots, t_n)$—occurring, for instance, on the left-hand side of an update rule—does not correspond, in general, to a fixed location (unless $n = 0$, of course). The location denoted by t may differ from state to state, if some t_i contains non-static function names.[12] Therefore, a straightforward syntax-directed translation of terms as in the basic translation scheme is not applicable. The source ASM has to be transformed in order to expose the locations that are read or written. The locations so obtained can then be identified with nullary external/dynamic functions (i.e., state variables), such that—essentially—the ASM is reduced to a basic transition system.

A similar issue arises when dealing with ASM-SL data structures. In order to represent complex data structures—such as lists or free types—in ASM_0/SMV, their values must be mapped to nullary constructors (elements of a corresponding enumerated type), which can then be mapped further to SMV symbolic constants (atoms). However, a term $t \equiv f(t_1, \dots, t_n)$, where f is a static n-ary function with $n > 0$, does not uniquely identify a value, unless all terms t_i are static. Thus, a trasformation is needed in order to expose the single values.

Another difficulty is constituted by those ASM constructs which introduce and bind logical variables[13], like `do-forall` rules, `case` rules with pattern matching, and quantifiers. To translate them into ASM_0, all constructions involving logical variables must be eliminated and replaced by equivalent constructions containing only ground terms and ground rules.

[12] However, if all terms t_i are static (i.e., contain only static function names), they can be evaluated statically to values x_i, and the term $f(t_1, \dots, t_n)$ denotes uniquely a location $(f, \overline{x})$.

[13] To avoid confusions, we distinguish here between state variables, which make up the system state, and logical variables, bound by particular constructs within a given scope.

```
static function n == 4

freetype CMD == { reset, incr, select : INT }

external function cmd : CMD
with cmd in { reset, incr } union { select (i) | i in { 0..n-1 } }

dynamic function addr : INT
with addr in { 0..n-1 }
initially 0

dynamic function ctr : INT -> INT
with ctr (i) in { 0, 1, 2 }
initially MAP_TO_FUN { i -> 0 | i in { 0..n-1 } }

transition Program ==
  case cmd of
    reset      : do forall i in { 0..n-1 }
                    ctr (i) := 0
                 enddo ;
    select (i) : addr := i ;
    incr       : ctr (addr) := (ctr (addr) + 1) mod 3
  endcase
```

Table 7.5: Multi-Counter: ASM-SL Specification

7.4.2 An Example

As an example, we introduce in Table 7.5 a generalization of the Counter modulo 3. Our hardware module now consists of n counters (represented by the unary dynamic function `ctr`), individually accessible through an address i, $0 \leq i < n$. At each moment, only one of the counters is *active* and can be incremented (its address is given by the nullary dynamic function `addr`). The input of the module is a *command* (external function `cmd`). The command is either *reset*, which resets all counters to 0, *select*(i), which selects counter i as the active counter, or *incr*, which increments the active counter.

The problem discussed above about n-ary functions with $n > 0$ is illustrated, for instance, by the update rule

```
ctr (addr) := (ctr (addr) + 1) mod 3,
```

which is executed when the command is *incr*. Which counter is actually incremented depends on the value of the dynamic function `addr`, which lies between 0 and $n - 1$ and can be changed at any time by a *select* command. Clearly, in ASM_0 or SMV, each of the counters (which are the *locations* of `ctr`) must be represented by a different state variable, e.g., by a state variable named `ctr_`i, $0 \leq i < n$. However, the term `ctr (addr)` can not be translated into any of these state variables, as it does not always denote the same location.

Note also that, in the `case` rule, while the cases `reset` and `incr` could easily be replaced by conditional rules (`if` *cmd* = *reset* ..., `if` *cmd* = *incr* ...), there is no trivial translation for the case `select(i)`, which involves pattern matching.

It can be observed that problems are due to applications of external and dynamic functions "deep" inside rules and terms. In fact, as already noticed:

- An arbitrary static term can be evaluated off-line[14] to a *value* $x = S(t)$, where S is an arbitrary state (the static part of a state is always the same in any state).
- A term of the form $f(t_1, \ldots, t_n)$, where f is a dynamic or external function name and $t_1, \ldots, t_n$ are static terms, can be evaluated off-line to a *location* $l = (f, (x_1, \ldots, x_n)) = (f, (S(t_1), \ldots, S(t_n))$, where S is an arbitrary state. We call such a term a *locational term*.

Static and locational terms are in a certain sense "state-independent", as they can be evaluated off-line to values and locations, which can then be immediately mapped to constants and state variables. On the contrary, if a term contains locational terms as subterms (e.g., `ctr(addr)` in our example), it is state-dependent and can not be directly translated to ASM_0/SMV, unless it is either an application of a primitive operator or a conditional term (corresponding to a `case`-expression in SMV).

Thus, an approach to translate ASM into ASM_0 is to iterate on the ASM program an "unfolding" transformation that, by introducing explicit case distinctions, gradually eliminates locational terms from the program (except those occurring on the left-hand side of update rules, which are needed). After each unfolding step, a simplification is applied, in order to reduce the size of the transformed program and the run time of the transformation.

How the transformation works is best understood by an example. Consider the ASM program of Table 7.5 (transition rule `Program`). It can be observed that the program contains two locational terms which are not on the left-hand side of an update, namely `cmd` (after the `case` keyword) and `addr` (in the *incr* case). We first unfold the program with respect to `cmd`, whose range (according the finiteness constraint for `cmd`) is $\{incr, reset, select(0), \ldots, select(3)\}$. By applying the unfolding transformation, as defined in Table 7.9 below, we obtain:

```
transition Program' ==
  if cmd = reset then 𝓔(⟦Program [reset / cmd]⟧)
  elseif cmd = incr then 𝓔(⟦Program [incr / cmd]⟧)
  elseif cmd = select(0) then 𝓔(⟦Program [select(0) / cmd]⟧)
  elseif cmd = select(1) then 𝓔(⟦Program [select(1) / cmd]⟧)
  elseif cmd = select(2) then 𝓔(⟦Program [select(2) / cmd]⟧)
  elseif cmd = select(3) then 𝓔(⟦Program [select(3) / cmd]⟧)
  endif
```

where $\mathcal{E}$ ("expand") is the unfolding transformation itself, $[\![\cdot]\!]$ is the simplifying transformation, and the notation $R[x \,/\, l]$ denotes the rule R with l replaced by x (the meaning of "replace" is plain syntactic substitution, except that locational terms occurring as left-hand side of update rules are not substituted).

[14] Here, *off-line* means "independently of the computation state" or, in the terminology of compiler constructors, "at compile-time".

Note that, after an unfolding step, each branch of the generated conditional must be simplified and further unfolded (the unfolding process terminates when there are no more locational terms to be expanded). We leave the unfolding apart, for the moment, and focus on the simplication. Consider, for instance, the branch corresponding to the case `cmd = select(1)`. The rule `Program` [`select(1)` / `cmd`] is as follows:

```
case select (1) of
  reset      : ... ;
  select (i) : addr := i ;
  incr       : ...
endcase
```

By means of the simplifying transformation $[\![\cdot]\!]$, which is a simple form of partial evaluation, the above rule is reduced to `addr := 1`. (See Tables 7.7 and 7.8 below for the definition of $[\![\cdot]\!]$). Once all the branches are simplified, the transformed program `Program'` is as follows (note the simplification of the `do-forall` rule to a block rule):

```
transition Program' ==
  if cmd = reset then 𝓔 ( ctr (0) := 0
                          ctr (1) := 0
                          ctr (2) := 0
                          ctr (3) := 0 )
  elseif cmd = incr then 𝓔( ctr (addr) := (ctr (addr) + 1) mod 3 )
  elseif cmd = select (0) then 𝓔( addr := 0 )
  elseif cmd = select (1) then 𝓔( addr := 1 )
  elseif cmd = select (2) then 𝓔( addr := 2 )
  elseif cmd = select (3) then 𝓔( addr := 3 )
  endif
```

The rules for the cases `reset` and `select(`i`)` do not need to be expanded further, as locational terms occur only as left-hand sides of updates there. The rule for `incr`, instead, still contains an unresolved locational term `addr` and must therefore be unfolded with respect to `addr`, whose range is $\{0, 1, 2, 3\}$:

```
𝓔( ctr (addr) := (ctr (addr) + 1) mod 3 ) =
  if addr = 0 then ctr (0) := (ctr (0) + 1) mod 3
  elseif addr = 1 then ctr (1) := (ctr (1) + 1) mod 3
  elseif addr = 2 then ctr (2) := (ctr (2) + 1) mod 3
  elseif addr = 3 then ctr (3) := (ctr (3) + 1) mod 3
  endif
```

The rule above, and consequently the whole program, is now essentially in ASM_0 form. Formally, it does still contain unary functions `select` and `ctr`. However, if one considers `ctr(0)`, `select(0)`, etc., as names of nullary functions (and not as applications of unary functions to a constant term), the transformed program can be considered as an ASM_0 program.

By replacing `ctr(0)`, `select(0)`, ..., through appropriate nullary function names and adding the corresponding declarations, we obtain a well-formed ASM_0 specification, shown in Table 7.6. This specification can be translated into SMV by means of the basic translation scheme of the previous section.

```
freetype CMD == { reset, incr, select_0, select_1, select_2, select_3 }

external function cmd : CMD
with cmd in { reset, incr, select_0, select_1, select_2, select_3 }

dynamic function addr : INT  with addr in { 0, 1, 2, 3 }  initially 0

dynamic function ctr_0 : INT  with ctr_0 in { 0, 1, 2 }  initially 0
dynamic function ctr_1 : INT  with ctr_1 in { 0, 1, 2 }  initially 0
dynamic function ctr_2 : INT  with ctr_2 in { 0, 1, 2 }  initially 0
dynamic function ctr_3 : INT  with ctr_3 in { 0, 1, 2 }  initially 0

transition Program ==
  if (cmd = reset)
  then ctr_0 := 0
       ctr_1 := 0
       ctr_2 := 0
       ctr_3 := 0
  elseif (cmd = incr)
  then if (addr = 0) then ctr_0 := (ctr_0 + 1) mod 3
       elseif (addr = 1) then ctr_1 := (ctr_1 + 1) mod 3
       elseif (addr = 2) then ctr_2 := (ctr_2 + 1) mod 3
       elseif (addr = 3) then ctr_3 := (ctr_3 + 1) mod 3
       endif
  elseif (cmd = select_0) then addr := 0
  elseif (cmd = select_1) then addr := 1
  elseif (cmd = select_2) then addr := 2
  elseif (cmd = select_3) then addr := 3
  endif
```

Table 7.6: Multi-Counter: Generated ASM_0 Specification

7.4.3 The Transformation

To formally define the transformation, it is convenient to extend the syntactic category of terms to "partially evaluated terms" (simply called "terms" in the sequel) by adding values and locations:

$$t \ ::= \ x \ \mid \ l \ \mid \ v \ \mid \ (t_1, \ldots, t_n) \ \mid \ f(t) \ \mid \ \ldots$$

(We adopt the convention that x stands for a value and l for a location).[15]

In this way, checking whether a given subterm is static or whether it is locational can be avoided. Instead, the simplifying transformation will reduce "on-the-fly" static terms to values and locational terms to locations.

The semantics of terms is extended to value and location terms as follows: $S_\rho(x) = x$, if x is a value; $S_\rho(l) = S((f,x)) = \mathbf{f}_S(x)$, if $l = (f,x)$ is a location. The semantics of rules is extended to cover the case of locations occurring as left-hand sides of update rules: $\Delta[\![\, l := t \,]\!](S,\rho) = \{\, (\, l,\, S_\rho(t)\,)\,\}$, if l is a location.

[15] Note that we switched to the ASM-SL convention that all functions (and consequently also all locations) are formally unary and that there are special terms and values for tuples, including the 0-tuple `()`.

Values and Locations:

$[\![x]\!]_\rho = x \qquad [\![\, l \,]\!]_\rho = l$

Variables:

$$[\![\, v \,]\!]_\rho = \begin{cases} x = \rho(v) & \text{if } v \in \mathrm{dom}(\rho) \\ v & \text{otherwise} \end{cases}$$

Tuple Terms:

$$[\![(t_1, \ldots, t_n)]\!]_\rho = \begin{cases} x = (x_1, \ldots, x_n) & \text{if } [\![t_i]\!]_\rho = x_i \text{ for all } i \in \{1, \ldots, n\} \\ & \quad \text{(i.e., each } [\![t_i]\!]_\rho \text{ is a value)} \\ t = ([\![t_1]\!]_\rho, \ldots, [\![t_n]\!]_\rho) & \text{otherwise} \end{cases}$$

Note: in the first case the result x is a value (a tuple),
in the second case the result t is an incompletely evaluated tuple term.

Function Applications:

If $[\![t]\!]_\rho = x'$ (i.e., $[\![t]\!]_\rho$ is a value):

$$[\![f(t)]\!]_\rho = \begin{cases} x = \mathbf{f}(x') & \text{if } f \text{ is a static function name} \\ l = (f, x') & \text{if } f \text{ is a dynamic or external function name} \end{cases}$$

Otherwise:

$[\![f(t)]\!]_\rho = f([\![t]\!]_\rho)$

Conditional and Case Terms:

See conditional and case rules (Table 7.8).

Finite Quantifications and Comprehensions:

If $[\![A]\!]_\rho = X$ (i.e., $[\![A]\!]_\rho$ is a value, in particular a finite set):

$[\![(\texttt{forall}\ p\ \texttt{in}\ A : G)]\!]_\rho = \bigwedge_{x \in X'} ([\![G]\!]_{\rho \oplus \mathcal{M}[\![p]\!](x)})$

$[\![(\texttt{exists}\ p\ \texttt{in}\ A : G)]\!]_\rho = \bigvee_{x \in X'} ([\![G]\!]_{\rho \oplus \mathcal{M}[\![p]\!](x)})$

$[\![\{\ t\ \texttt{|}\ p\ \texttt{in}\ A\ \texttt{with}\ G\ \}]\!]_\rho = \bigcup_{x \in X'} ([\![\texttt{if}\ G\ \texttt{then}\ \{t\}\ \texttt{else}\ \{\}]\!]_{\rho \oplus \mathcal{M}[\![p]\!](x)})$

where $X' = \{\, x \in X \mid \mathcal{M}[\![p]\!](x) \neq \mathrm{FAIL} \,\}$

Otherwise:

$[\![(Q\ p\ \texttt{in}\ A : G)]\!]_\rho = (Q\ p\ \texttt{in}\ [\![A]\!]_\rho : [\![G]\!]_{\rho \setminus \mathrm{vars}(p)})$

$[\![\{\ t\ \texttt{|}\ p\ \texttt{in}\ A\ \texttt{with}\ G\ \}]\!]_\rho = \{\ [\![t]\!]_{\rho \setminus \mathrm{vars}(p)}\ \texttt{|}\ p\ \texttt{in}\ [\![A]\!]_\rho\ \texttt{with}\ [\![G]\!]_{\rho \setminus \mathrm{vars}(p)}\ \}$

Table 7.7: Term Simplification

Basic Transition Rules:

$[\![\texttt{skip}]\!]_\rho$ = *(empty block)*

$[\![t_L := t_R]\!]_\rho = [\![t_L]\!]_\rho := [\![t_R]\!]_\rho$

$[\![R_1 \ldots R_n]\!]_\rho = [\![R_1]\!]_\rho \ldots [\![R_n]\!]_\rho$

$$[\![\texttt{if } G \texttt{ then } R_T \texttt{ else } R_F]\!]_\rho = \begin{cases} [\![R_T]\!]_\rho & \text{if } [\![G]\!]_\rho = \textbf{true} \\ [\![R_F]\!]_\rho & \text{if } [\![G]\!]_\rho = \textbf{false} \\ \texttt{if } [\![G]\!]_\rho \texttt{ then } [\![R_T]\!]_\rho \texttt{ else } [\![R_F]\!]_\rho & \text{otherwise} \end{cases}$$

Case Rules:

$$[\![\texttt{case } t_0 \texttt{ of } p : R_1 \texttt{ ; otherwise } R_2]\!]_\rho = \\ = \begin{cases} [\![R_1]\!]_{\rho \oplus \mathcal{M}[\![p]\!](x)} & \text{if } [\![t_0]\!]_\rho = x \text{ and } \mathcal{M}[\![p]\!](x) \neq \text{FAIL} \\ [\![R_2]\!]_\rho & \text{if } [\![t_0]\!]_\rho = x \text{ and } \mathcal{M}[\![p]\!](x) = \text{FAIL} \\ \texttt{case } [\![t_0]\!]_\rho \texttt{ of } p : [\![R_1]\!]_{\rho \setminus \text{vars}(p)} \texttt{; otherwise } [\![R_2]\!]_\rho & \text{if } [\![t_0]\!]_\rho \text{ is not a value} \end{cases}$$

Do-Forall Rules:

If $[\![A]\!]_\rho = X$ (i.e., $[\![A]\!]_\rho$ is a value, in particular a finite set):

$$[\![\texttt{ do forall } p \texttt{ in } A \texttt{ with } G \;\; R']\!]_\rho = \\ = \;\; [\![\texttt{if } G \texttt{ then } R']\!]_{\rho \oplus \mathcal{M}[\![p]\!](x_1)} \\ \ldots \\ [\![\texttt{if } G \texttt{ then } R']\!]_{\rho \oplus \mathcal{M}[\![p]\!](x_n)}$$

where $\{ x_1, \ldots, x_n \} = X' = \{ x \in X \mid \mathcal{M}[\![p]\!](x) \neq \text{FAIL} \}$

Otherwise:

$$[\![\texttt{ do forall } p \texttt{ in } A \texttt{ with } G \;\; R']\!]_\rho = \\ = \;\; \texttt{do forall } p \texttt{ in } [\![A]\!]_\rho \texttt{ with } [\![G]\!]_{\rho \setminus \text{vars}(p)} \;\; [\![R']\!]_{\rho \setminus \text{vars}(p)}$$

Table 7.8: Rule Simplification

As already seen in the example above, besides the *unfolding* transformation $\mathcal{E}$ we also need an auxiliary transformation, the *simplification* $[\![\cdot]\!]_\rho$, which is defined for both terms and transition rules. The subscript ρ denotes the environment in which the simplification is perfomed (recall that an environment ρ binds variables to the corresponding values, see Sect. 1.2). Term simplification is defined in Table 7.7, rule simplification in Table 7.8.

The following properties hold for term simplication, provided that ρ contains bindings for all free variables occurring in t:

- If t is a static term, then $[\![t]\!]_\rho$ is a value x. In every state S, $[\![t]\!]_\rho$ coincides with the interpretation of t, i.e., $[\![t]\!]_\rho = S_\rho(t) = x$.

- If $t \equiv f(t')$ is a locational term (i.e., f is a dynamic or external function name and t' a static term), then $[\![t]\!]_\rho$ is a location $l = (f, x)$. In every state S, $[\![t]\!]_\rho = (f, [\![t']\!]_\rho) = (f, S_\rho(t')) = (f, x)$.

As a special case we have that, in the empty environment, every closed static term simplifies to a value and every closed locational term to a location. Thus,

If R contains no locations, except as left-hand side of update rules:
$\mathcal{E}(R) = R$
Otherwise:
$\mathcal{E}(R) =$ `if` $l = x_1$ `then` $\mathcal{E}([\![R[x_1/l]]\!])$
`else if` $l = x_2$ `then` $\mathcal{E}([\![R[x_2/l]]\!])$
...
`else if` $l = x_n$ `then` $\mathcal{E}([\![R[x_n/l]]\!])$
where l is the first location occurring in R (but not as lhs of an update) and $\{x_1, \ldots, x_n\}$ is the range of location l.

Table 7.9: Rule Unfolding

in the sequel of this section, we write $[\![t]\!]$ as a shorthand for $[\![t]\!]_\emptyset$ whenever t is a closed term (and the same convention will hold for closed rules).

The unfolding transformation $\mathcal{E}(R)$, which operates on closed rules (like the ASM program P), is defined in Table 7.9. It works as follows:

- If locations occur in R only as left-hand sides of update rules, $\mathcal{E}$ terminates yielding R as result (as there is nothing left to unfold).
- Otherwise, it looks for the first location l in R that is not left-hand side of some update rule and unfolds R with respect to l. Then, the unfolding is recursively applied to the subrules $[\![R[x_i/l]]\!]$ obtained by substituting the values x_i for l in R and by simplifying the resulting rule.

Applying $\mathcal{E}$ to the (simplified) ASM program $[\![P]\!]$ yields a program $P' = \mathcal{E}([\![P]\!])$ which is almost an ASM_0 program. As already explained in relation to the example, locations must then be replaced by nullary dynamic/external function names (state variables) and values by nullary constructors (constants) in order to obtain a proper ASM_0 program.

Fig. 7.4 illustrates graphically how the transformation works, when applied to the multi-counter example introduced in the previous section. Another example is shown in Fig. 7.5, where the program is simpler, but the finiteness constraints are somewhat fancier (in particular, note that the finiteness constraint for f depends on the argument of f). In the picture, the root of the tree—enclosed in the dashed box—is the (simplified) ASM program $[\![P]\!]$ to be transformed. The successors of each node in the tree are obtained as result of unfolding one location (under the given finiteness constraints): for instance, the successors of the root node are the rules $[\![[\![P]\!][1/a]]\!]$, $[\![[\![P]\!][2/a]]\!]$, and $[\![[\![P]\!][3/a]]\!]$, respectively. Locations are emphasized by enclosing them in boxes: note that, at the leaves, locations occur only as left-hand side of updates, thus they cause no further unfolding. The dashed box on the right contains the ASM_0 program produced by the transformation: note that the locations actually affected by the ASM program—which are revealed by the unfolding—are mapped to nullary functions ("state variables"), whose ranges are derived from the finiteness constraints (see box at the top right corner).

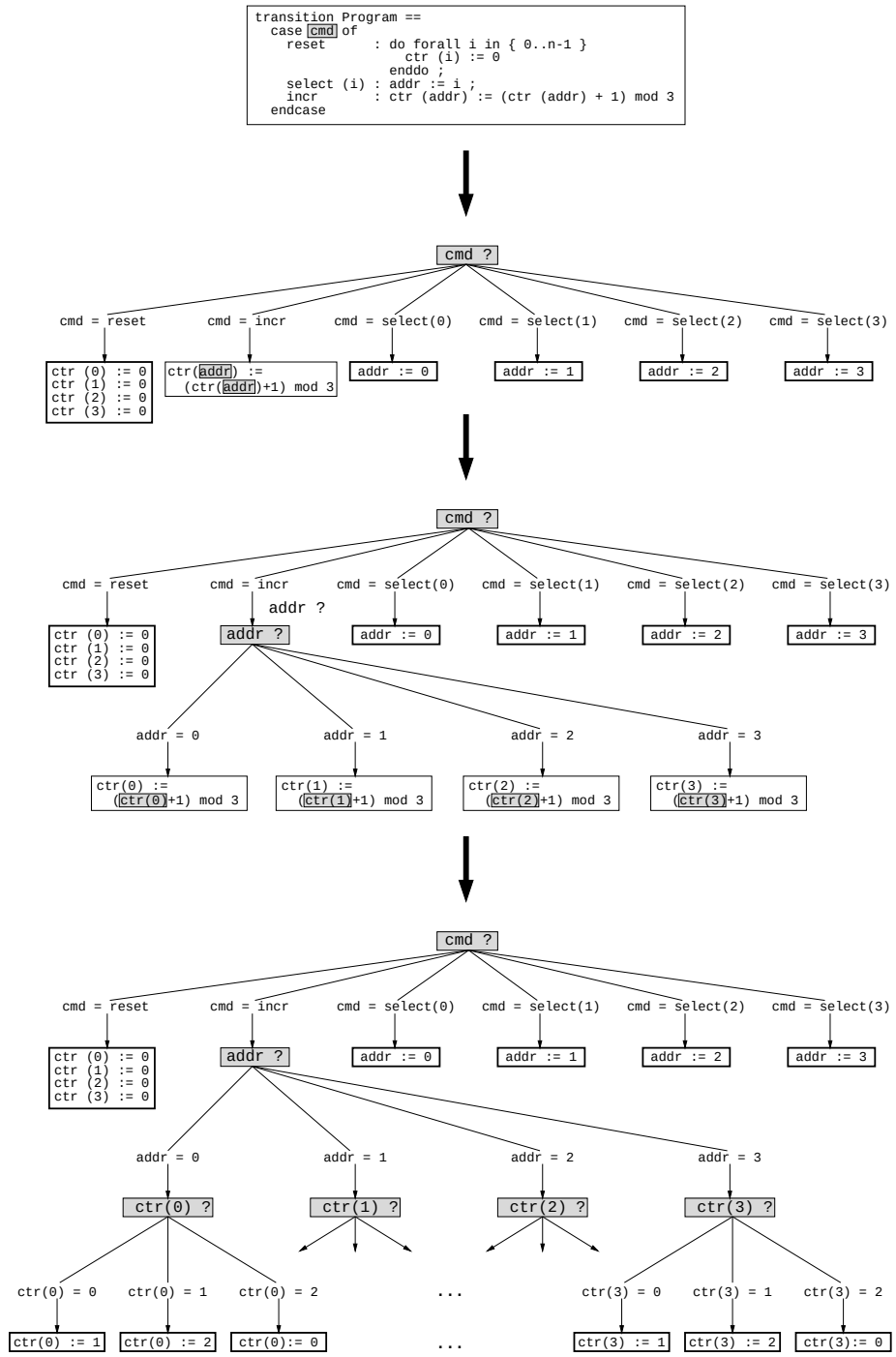


Figure 7.4: Multi-Counter: Transformation ASM $\mapsto$ ASM$_0$

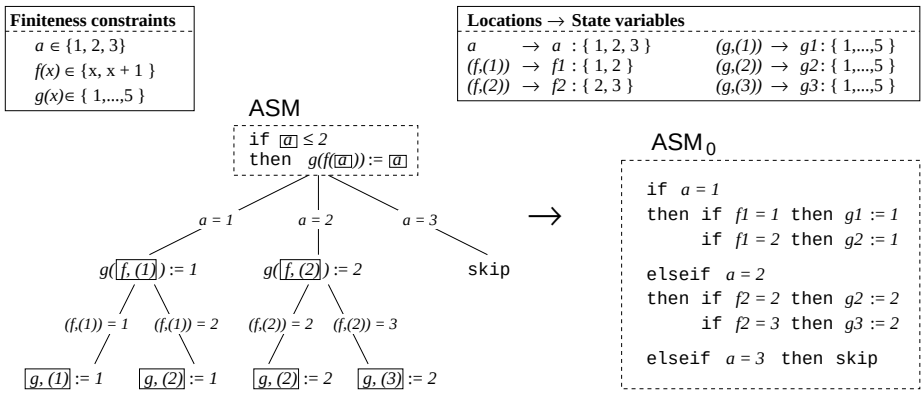


Figure 7.5: Rule Transformation Example

7.4.4 Correctness and Completeness

The translation scheme which transforms ASM to ASM_0 is correct and complete, in the sense that:

1. The generated ASM_0 programs are semantically equivalent to the ASM programs from which they are obtained (correctness).
2. For every ASM program, the transformation algorithm terminates yielding an ASM_0 program (completeness).

This is made precise in the following theorem.

Theorem 7.1 For every closed rule R, not containing `choose`-rules, derived functions, or `FUN_TO_MAP`/`REL_TO_SET` operators:

1. $\mathcal{E}(R)$ is defined, i.e., $\mathcal{E}$ applied to R terminates.
2. $\overline{R} = \mathcal{E}(R)$ is an ASM_0 rule.
3. In all states S, $\Delta_S(\overline{R}) = \Delta_S(R)$, i.e., $\overline{R}$ is semantically equivalent to R.

Proof See Appendix D.

7.5 The ASM2SMV Tool

In this section we describe the ASM2SMV tool, which supports model-checking of ASM-SL specifications by means of SMV. The tool is built on the top of the ASM Workbench kernel, making intensive use of the generic mechanisms described in Sect. 6.4. First, we give an overview of the functionalities and architecture of ASM2SMV (Sect. 7.5.1). Then, we present a simple example to give the reader a more concrete impression of how the techniques and the tool

presented in this chapter are applied (Sect. 7.5.2). After some remarks on the efficiency of the translation (Sect. 7.5.3), the section is concluded by a discussion of some implementation issues (Sect. 7.5.4).

7.5.1 Overview

The ASM2SMV tool allows to translate any finite-state ASM into an SMV model (i.e., into a finite transition system specified in the SMV language). Supported is the whole ASM-SL language, with the exception of `choose` rules and recursive derived functions. The properties to be checked have to be specified separately in CTL: together with the automatically generated SMV model, they constitute the input needed by SMV to perform the verification task.

To carry out the translation, the ASM2SMV tool employs all the techniques discussed in the previous sections. More precisely, the translation from ASM-SL to SMV takes place in three phases.

1. **Preprocessing**. In this phase, applications of named transition rules and of non-recursive derived functions occurring in the program P are expanded as if they were macros. This macro-expansion process is iterated until there is nothing left to expand (note that it always terminates, because no recursive definitions are involved). As a result a new program P', equivalent to P, is obtained, which completely defines (in one "monolithic" rule) the system behaviour and contains only terms consisting of static, dynamic, and external function names (no derived functions).

2. **Unfolding**. In this phase, the extended translation scheme discussed in Sect. 7.4 is applied to the program P' resulting from the preprocessing phase. The result is an ASM_0 program P'' equivalent to P'.

3. **Code generation**. In this phase, the ASM_0 program P'' resulting from the unfolding phase is translated into SMV code according to the basic translation scheme of Sect. 7.3.[16]

If the original ASM contains `choose` rules, they have to be eliminated by hand as explained in Sect. 7.2.2.[17]

7.5.2 An Example

As an example to illustrate the elaborations performed by ASM2SMV,we consider an ASM version of the well-known dining philosophers problem [35]. The corresponding ASM-SL specification is shown in Fig. 7.10. After the constant `n`

[16] This is the only SMV-specific part of ASM2SMV. Other model checkers could be supported by replacing the code generation module, while keeping the first two phases unchanged.

[17] In principle this transformation step could be performed automatically, by letting the tool generate appropriate external function names and the corresponding declarations. However, it is preferable to let the user introduce explicit choice functions and give them meaningful names: then, it becomes easier to recognize in the counterexamples (possibly found by SMV if some property is not satisfied) the sequence of choices which leads to the wrong behaviour.

corresponding to the number of philosophers, declarations of types which represent philosopher identifiers (`PHIL`) and states (`PHIL_STATE`) as well as fork identifiers (`FORK`) and states (`FORK_STATE`) follow. The static functions `Phil`, `PhilState`, `Fork` and `ForkState` are finite subsets of the corresponding types and include only the elements actually needed within the specification: although these declarations may look slightly redundant, they are important for the definition of appropriate finiteness constraints. The static functions `left`, `right : PHIL -> FORK` have obvious meaning and are only introduced for better readability. The dynamic functions `phil_state` and `fork_state` constitute the internal system state, while the external function `self` determines, in each state, which philosopher makes a move. The rule definitions which follow constitute the behavioural specification and are quite self-explaining. The ASM program is the rule called `Program`. The property to be checked is the "progress property" of Table 7.11: if it holds in all reachable states, then no deadlock will happen.

In the preprocessing phase, the applications of named rules `pick_up_forks` and `release_forks` occurring in `Program` are replaced by the corresponding definition bodies with the formal parameter `ph` replaced by the `self`. (We do not reproduce the result of preprocessing here, as it is quite close to the original.)

The result of applying the unfolding/simplifying transformation to the (preprocessed) program is shown in Table 7.12.[18] It is easy to recognize the typical structure of ASM_0 programs obtained by the unfolding transformation of Table 7.9, which are essentially decision trees whose leaves are blocks of updates of the form *location* := *value* (as it clearly appears in Fig. 7.5, where this structure is represented in graphical form for a simpler rule).

In the last phase (code generation), the ASM_0 program of Table 7.12—after being transformed into a block of guarded updates—is translated to SMV by: *(i)* rearranging it in such a way that all the updates of the same location are collected in a single transition rule (which becomes a **next**-assignment in SMV), and *(ii)* mapping each value and each location occurring in the ASM_0 program to a corresponding SMV symbolic constant or state variable, respectively. Moreover, state variable declarations and—for those state variables which result from the unfolding of dynamic functions—corresponding initializations are generated.[19] Assuming that the mapping of values and locations is as indicated in Table 7.13, the SMV model resulting from the code generation will be as shown in Table 7.14. Note that the SMV model does not contain an `init` and a `next`-assignment for `_self`, as this state variable is obtained from an external function of the ASM specification (thus, it is "unrestricted").

Finally, Table 7.15 shows the CTL formula to be checked by SMV, which is obtained by applying the same transformations as for the ASM program to the progress property of Table 7.11 (which is an ASM-SL term) and then prefixing the resulting formula by the temporal operator AG (meaning that the property has to hold in all reachable states).

[18] The rule shown in the table is the actual ASM_0 code generated by ASM2SMV, except that, for obvious reasons, some cases are omitted (`...`). The `endif` delimiters are also omitted.

[19] In general, the generated SMV code also includes proof obligations corresponding to range conditions and no-conflict conditions, but in this example they could be discarded statically.

```
static function n == 5

freetype PHIL       == { phil : INT }
freetype PHIL_STATE == { thinking, eating }

freetype FORK       == { fork : INT }
freetype FORK_STATE == { free, used_by : PHIL }

static function Phil == { phil(i) | i in {0..n-1} }
static function PhilState == { thinking, eating }

static function Fork == { fork(i) | i in {0..n-1} }
static function ForkState == { free } union { used_by(ph) | ph in Phil }

static function left (phil(i))  == fork(i)
static function right (phil(i)) == fork((i+1) mod n)

dynamic function phil_state : PHIL -> PHIL_STATE
with phil_state (ph) in PhilState
initially MAP_TO_FUN { ph -> thinking | ph in Phil }

dynamic function fork_state : FORK -> FORK_STATE
with fork_state (fo) in ForkState
initially MAP_TO_FUN { fo -> free | fo in Fork }

external function self : PHIL
with self in Phil

transition pick_up_forks (ph) ==
  if fork_state (left (ph)) = free and fork_state (right (ph)) = free
  then fork_state (left (ph))  := used_by (ph)
       fork_state (right (ph)) := used_by (ph)
       phil_state (ph) := eating
  endif

transition release_forks (ph) ==
  block
    fork_state (left (ph))  := free
    fork_state (right (ph)) := free
    phil_state (ph) := thinking
  endblock

transition Program ==
  case phil_state (self) of
    thinking : pick_up_forks (self) ;
    eating   : release_forks (self)
  endcase
```

Table 7.10: Dining Philosophers: ASM-SL Specification

Progress Function:

```
derived function progress (ph) ==
  ( phil_state(ph) = thinking and
    fork_state (left(ph)) = free and fork_state (right(ph)) = free )
  or ( phil_state(ph) = eating )
```

Progress Property:

```
(exists ph in Phil : progress (ph))
```

Table 7.11: Dining Philosophers: Progress Property (in ASM-SL)

```
if (self = phil (0))
then if (phil_state (phil (0)) = eating)
     then fork_state (fork (0)) := free
          fork_state (fork (1)) := free
          phil_state (phil (0)) := thinking
     elseif (phil_state (phil (0)) = thinking)
     then if (fork_state (fork (0)) = free)
          then if (fork_state (fork (1)) = free)
               then fork_state (fork (0)) := used_by (phil (0))
                    fork_state (fork (1)) := used_by (phil (0))
                    phil_state (phil (0)) := eating
elseif (self = phil (1))
then ...
elseif (self = phil (2))
then ...
elseif (self = phil (3))
then ...
elseif (self = phil (4))
then if (phil_state (phil (4)) = eating)
     then fork_state (fork (4)) := free
          fork_state (fork (0)) := free
          phil_state (phil (4)) := thinking
     elseif (phil_state (phil (4)) = thinking)
     then if (fork_state (fork (0)) = free)
          then if (fork_state (fork (4)) = free)
               then fork_state (fork (4)) := used_by (phil (4))
                    fork_state (fork (0)) := used_by (phil (4))
                    phil_state (phil (4)) := eating
```

Table 7.12: Dining Philosophers: Program after Unfolding

ASM Values		**SMV Symbolic Constants**
phil(0), phil(1),...	↦	phil_0, phil_1,...
thinking, eating	↦	thinking, eating
fork(0), fork(1),...	↦	fork_0, fork_1,...
free, used_by(phil(0)),...	↦	free, used_by_phil_0,...
ASM Locations		**SMV State Variables**
phil_state(phil(0)),...	↦	phil_state_0,...
fork_state(fork(0)),...	↦	fork_state_0,...
self	↦	_self

Table 7.13: Dining Philosophers: Mapping of Values and Locations

```
MODULE main

VAR
  _self : {phil_0, phil_1, phil_2, phil_3, phil_4};
  fork_state_0 : {free, used_by_phil_0, ..., used_by_phil_4};
  ...
  fork_state_4 : {free, used_by_phil_0, ..., used_by_phil_4};
  phil_state_0 : {eating, thinking};
  ...
  phil_state_4 : {eating, thinking};

ASSIGN
  init (fork_state_0) := free;
  next (fork_state_0) :=
   case
     (_self = phil_0) & (phil_state_0 = eating) : free;
     (_self = phil_0) & (phil_state_0 = thinking)
       & (fork_state_0 = free) & (fork_state_1 = free) : used_by_phil_0;
     (_self = phil_4) & (phil_state_4 = eating) : free;
     (_self = phil_4) & (phil_state_4 = thinking)
       & (fork_state_4 = free) & (fork_state_0 = free) : used_by_phil_4;
     TRUE : fork_state_0;
   esac;

  ...

  init (fork_state_4) := free;
  next (fork_state_4) :=
   case
     (_self = phil_3) & (phil_state_3 = eating) : free;
     (_self = phil_3) & (phil_state_3 = thinking)
       & (fork_state_3 = free) & (fork_state_4 = free) : used_by_phil_3;
     (_self = phil_4) & (phil_state_4 = eating) : free;
     (_self = phil_4) & (phil_state_4 = thinking)
       & (fork_state_4 = free) & (fork_state_0 = free) : used_by_phil_4;
     TRUE : fork_state_4;
   esac;

  init (phil_state_0) := thinking;
  next (phil_state_0) :=
   case
     (_self = phil_0) & (phil_state_0 = eating) : thinking;
     (_self = phil_0) & (phil_state_0 = thinking)
       & (fork_state_0 = free) & (fork_state_1 = free) : eating;
     TRUE : phil_state_0;
   esac;

  ...

  init (phil_state_4) := thinking;
  next (phil_state_4) :=
   case
     (_self = phil_4) & (phil_state_4 = eating) : thinking;
     (_self = phil_4) & (phil_state_4 = thinking)
       & (fork_state_4 = free) & (fork_state_0 = free) : eating;
     TRUE : phil_state_4;
   esac;
```

Table 7.14: Dining Philosophers: Generated SMV Model

```
SPEC
  AG ( (phil_state_0 = thinking & fork_state_0 = free & fork_state_1 = free)
       | (phil_state_0 = eating) )
    | ( (phil_state_1 = thinking & fork_state_1 = free & fork_state_2 = free)
       | (phil_state_1 = eating) )
    | ( (phil_state_2 = thinking & fork_state_2 = free & fork_state_3 = free)
       | (phil_state_2 = eating) )
    | ( (phil_state_3 = thinking & fork_state_3 = free & fork_state_4 = free)
       | (phil_state_3 = eating) )
    | ( (phil_state_4 = thinking & fork_state_4 = free & fork_state_0 = free)
       | (phil_state_4 = eating) )
```

Table 7.15: Dining Philosophers: SMV Specification

7.5.3 Performance Considerations

By looking superficially at the unfolding transformation of Table 7.9 and at the generated SMV code for the dining philosophers in Table 7.14, one may have the impression that the translation from ASM to SMV leads to an explosion in the size of the models, which can have disastrous consequences on the efficiency of verification by model checking. However, a more careful analysis reveals that this is not the case.

Actually, it is true that the unfolding transformation often results in very large decision trees (`case`-structures in SMV), but this fact by itself does not have a negative influence on the verification efficiency. The reason is that the verification costs depend on the size of the binary decision diagrams (BDDs) representing the transition relation, and not on the size of the SMV input containing the transition system description.[20] As BDDs are a canonical representation of boolean functions (once a variable ordering is fixed), the same transition relation will always have the same internal representation in SMV, no matter whether it is specified by a concise expression or by an exhaustive case distinction.

Table 7.16 shows some experimental results related to the costs of transforming the dining philosophers example from ASM to SMV and of the subsequent verification of the progress property by SMV (the size of the original ASM-SL specification is 53 lines, 1375 bytes).[21] It can be observed that, despite the growth of the translation time and of the size of the SMV models generated by ASM2SMV, the time required for the transformation is negligible compared to the time needed for the actual verification task (at least for larger values of the parameter n). This is not surprising, as the unfolding performed by ASM2SMV only concerns the transition relation, while for the verification task the whole set of reachable states must be checked (even if by symbolic techniques).[22]

On the other hand, although the size of the generated SMV model has by itself no influence on the verification performance, the translation process can

[20] The time needed for parsing the input and constructing the BDD representation of the transition relation out of it—compared to the time needed for the actual verification task, whose complexity is in general exponential—is negligible.

[21] The experiments were performed on a Sparc Ultra workstation with 128 MB main memory.

[22] The complexity of model checking this problem by means of the algorithms implemented in SMV is actually exponential in the parameter n (see discussion in Chapter 9 of [66]).

No. of agents	ASM2SMV		SMV			
	(a)	(b)	(c)	(d)	(e)	(f)
$n = 4$	0.66/0.41	434 / 14,156	0.11/0.01	4,100	1,216 k	$474 + 14$
$n = 5$	0.82/0.46	556 / 18,524	0.17/0.01	6,976	1,216 k	$674 + 15$
$n = 6$	0.79/0.40	686 / 23,280	0.22/0.05	10,296	1,344 k	$1077 + 22$
$n = 7$	1.00/0.31	824 / 28,424	0.36/0.03	11,150	1,344 k	$1279 + 5$
$n = 8$	1.10/0.43	970 / 33,956	0.77/0.04	11,043	1,408 k	$2536 + 34$
$n = 9$	1.40/0.36	1,124 / 39,876	1.20/0.04	13,155	1,472 k	$2942 + 33$
$n = 10$	2.05/0.44	1,286 / 46,184	2.89/0.03	25,027	1,664 k	$3975 + 45$
$n = 11$	2.17/0.47	1,456 / 52,968	4.57/0.02	28,427	1,728 k	$3855 + 28$
$n = 12$	2.63/0.45	1,634 / 60,150	6.83/0.05	50,295	2,048 k	$5640 + 52$
$n = 13$	2.87/0.47	1,820 / 67,730	16.53/0.04	72,548	2,368 k	$5978 + 45$
$n = 14$	3.01/0.44	2,014 / 75,708	25.08/0.12	111,856	3,008 k	$7647 + 61$
$n = 15$	2.07/0.40	2,216 / 84,084	70.50/0.08	211,057	4,544 k	$6995 + 6$
$n = 16$	3.93/0.50	2,426 / 92,858	83.68/0.08	403,953	7,552 k	$12504 + 82$

(a) time for the ASM to SMV translation (user/system time, in seconds)
(b) size of the generated SMV model (in lines of code / bytes)
(c) time consumed by SMV for verification (user/system time, in seconds)
(d) BDD nodes allocated
(e) bytes allocated
(f) BDD nodes representing transition relation

Table 7.16: Dining Philosophers: Costs for Transformation/Verification

be a bottleneck if dynamic/external functions of the original ASM specification assume values from large integer ranges or complex data types (such as lists or finite sets), as this may give rise to a very large amount of unfolding. Moreover, the careless use of the high-level constructs and complex data structures of ASM-SL leads easily to specifications which look simple, but describe transition systems which are too complex to be model-checked.

Experiences with applications of ASM2SMV, such as the switch controller presented in Chapter 9 or the FLASH cache coherence protocol discussed in [33], showed that the high-level features are very helpful for modelling (in particular, in the mentioned applications, lists were used to model message queues), but that verification is in practice feasible only for small system instances (in particular, the length of queues had to be limited to two or three messages, respectively). However, an advantage of the approach presented here is that the dimension of the state space can be easily adjusted by changing only the finiteness constraints (until a point is reached where verification becomes feasible).

7.5.4 Implementation Issues

The implementation of ASM2SMV includes modules for carrying out the three phases mentioned in Sect. 7.5.1 above, namely: preprocessing, unfolding, and code generation. The preprocessing consists of simple syntactic expansions, while the modules implementing the unfolding and code generation phases are

essentially a transliteration in ML of the transformation schemes presented in Sect. 7.4 and 7.3, respectively. All modules take advantage of the resources provided by the ASM Workbench kernel to support ASM-SL (such as ASTs, structural induction, pretty-printer). The code generator must additionally provide SMV-related resources (essentially, ASTs for the SMV language and the corresponding pretty-printer).

The module implementing the unfolding phase is the most conspicuous one, as it has to deal with the complete ASM-SL language (while the code generator operates on ASM_0, which is a quite small ASM subset). A relevant part of this module is constituted by the *simplifier*, which implements the transformations defined in Table 7.7 and 7.8 for terms and rules, respectively. The term simplifier will be described in some detail here, as it is—among other things—an interesting application of the polymorphic ASTs introduced in Sect. 6.4.3.[23]

The core data structure around which the term simplifier is built is constituted by the *partially evaluated terms* introduced in Sect. 7.4, which enrich the structure of terms by adding *values* and *locations*. Partially evaluated terms (*pe-terms* for short) can be easily implemented by appropriately instantiating the type `TERM'` of polymorphic ASTs for terms:[24]

```
datatype PE_TERM =
  Val of VALUE
| Loc of LOCATION
| Term of (NAME, PATT, PE_TERM) TERM'
```

According to this definition, a pe-term can be either a value, or a location, or an "ordinary" (i.e., unevaluated) pe-term. Note that, in this way, mutually recursive types `PE_TERM` and `(NAME, PATT, PE_TERM) TERM'` are implicitly defined, as already explained in Sect. 6.4.3.

Before an ASM-SL term (of type `TERM`) can be simplified, it has to be converted into a `PE_TERM`. This is done by means of a function[25]

`pe_term` : `TERM` → `PE_TERM`,

which wraps each node of the original AST with the `Term` constructor of `PE_TERM`, thus indicating that the term is still completely unevaluated. For instance:

```
pe_term (TupleTerm (VarTerm ("x"), VarTerm ("y"))) =
  Term (TupleTerm' (Term (VarTerm' ("x"), Term (VarTerm' ("y")))))
```

(See also Fig. 6.3 in Chapter 6, which shows an intentionally similar example).

The implementation of the term simplifier—for the subset of ASM-SL identified in Chapter 6, see Tables 6.2 and 6.3—is shown in Table 7.17. The actual term simplification is performed by the function

[23] A discussion of the rule simplifier is omitted, as it follows the same lines.

[24] In this way, the term constructors (`VarTerm'`, `AppTerm'`, `TupleTerm'`, etc.) from `TERM'` can be reused. Otherwise, all term constructors should be included in the definition of `PE_TERM`, in addition to the constructors `Val` for values and `Loc` for locations.

[25] Not shown here, as quite trivial and similar in structure to the `patt_nn` function defined in Chapter 6, shown in Table 6.9.

`simplify_term` : `PE_TERM` → `ENV` → `PE_TERM`,

where "`simplify_term` t ρ" corresponds to "$[\![t]\!]_\rho$" of Table 7.7.

In conformance to the mutually recursive type structure, the simplifier is built on two mutually recursive functions, `simplify_term` and `simplify_term'`. The latter is implemented by structural induction over polymorphic ASTs. Despite the ML syntax, it should be possible to recognize the transformation defined in Table 7.7 in the `simplify_term` function and in the auxiliary functions handling the different kinds of terms (such as `simplify_var_term`, `simplify_tuple_term`, and so on).

7.6 Related Work

Extending tool environments for high-level specification languages with an interface to a model checker is an upcoming topic. One can find approaches that are quite similar to ours but work on a different language: for instance, [29, 69] deal with model checking of Statecharts, in [64] Controller Specification (CSL) models are transformed and model checked by the SVE tool, [68] equips the multi-language environment SYNCHRONIE with an interface to the VIS model checker (just to mention a few examples).

Closer to our approach from the point of view of language, [82] also investigates automatic verification of ASM: there ASMs are represented, independently of their possible input, by means of a logic for computation graphs (called CGL*). The resulting formula is combined with a CTL*-like formula which specifies properties and then checked by means of deciding its finite validity. While this approach addresses the problem of checking systems with infinitely many inputs, it is only applicable to ASMs with only 0-ary dynamic functions (i.e., ASM_0 programs).

```
fun simplify_term (t :PE_TERM) (env :ENV) :PE_TERM =
  case t of
    Val x  => Val x
  | Loc l  => Loc l
  | Term t => simplify_term' t env

and simplify_term' (t :(NAME, PATT, PE_TERM) TERM') :ENV -> PE_TERM =
  term'_induction (id, id) simplify_term {
    VarTerm'   = simplify_var_term,
    AppTerm'   = simplify_app_term,
    TupleTerm' = simplify_tuple_term,
    CondTerm'  = simplify_cond_term,
    CaseTerm'  = simplify_case_term,
    SetCompr'  = simplify_set_compr
  } t

and simplify_var_term var_id env =
  (Val (lookup env var_id))
  handle _ => Term (VarTerm' var_id)

and simplify_tuple_term ts env =
  let val ts' = map (fn t => t env) ts
  in if (forall_in_list ts' (fn Val x => true | _ => false))
     then Val (TUPLE (map (fn Val x => x) ts'))
     else Term (TupleTerm' ts')
  end

and simplify_app_term (f, t) env =
  case (function_kind f, t env) of
    (Static, Val x) => Val (static_interpretation f x)
  | (_,      Val x) => Loc (f, x)
  | (_,      t)     => Term (AppTerm' (f, t))

and simplify_cond_term (G, t1, t2) env =
  case G env of
    Val (CELL ("true", _))  => t1 env
  | Val (CELL ("false", _)) => t2 env
  | G_ => Term (CondTerm' (G_, t1 env, t2 env))

and simplify_case_term (t0, p, t1, t2) env =
  case t0 env of
    Val x => (t1 (env ++ eval_patt p x) handle FAIL => t2 env)
  | t0_ => Term (CaseTerm' (t0_, p, t1 (env \\ vars_in_patt p), t2 env))

and simplify_compr ast_cons combine element (t, p, A, G) env =
  case A env of
    Val (SET range) => simplify_term (combine (map element range)) env
  | A_ => let val env' = env \\ vars_in_patt p
          in Term (ast_cons (t env', p, A_, G env')) end

and simplify_set_compr (t, p, A, G) env =
  simplify_compr SetCompr' UNION
    (fn x => let val env' = env ++ eval_patt p x
             in simplify_term (Term (CondTerm' (G env', term_singleton (t env'),
                                                term_emptyset))) env'
             end handle FAIL => term_emptyset)
    (t, p, A, G) env
```

Table 7.17: Implementation of the Term Simplification Function

Part III

Applications

Chapter 8

Heterogeneous Modelling and Meta-Modelling

As an introduction to the case study presented in the next chapter, we discuss here from a general methodological perspective the subject of heterogeneous system modelling and the integration of heterogeneous models by means of a meta-model (which is, in our case, an ASM-based model).[1]

This chapter is structured as follows. After a few words about motivation (Sect. 8.1), we begin with a methodological discussion about the problem of the initial formalization of an informally specified or already existing system, solved by constructing a so-called ground model (Sect. 8.2). In this context, we introduce our approach to heterogeneous modelling and the related integration problem (Sect. 8.3). Finally, we focus on a particular aspect, namely the integration of controller and device models, which is developed in detail in the case study of the next chapter (Sect. 8.4).

8.1 Motivation

Complex technical systems in virtually all application areas increasingly rely on *embedded* hardware/software components performing various control and supervision tasks. Typical examples are automotive control and industrial automation. In general, a strong tendency towards distributed solutions running on heterogeneous system platforms can be observed. Complex embedded systems are frequently realized as loosely-coupled aggregations of disparate components performing specialized tasks rather than monolithic architectures with a regular structure [88, 89].

As a result, engineering of complex embedded systems often involves several domain-specific description techniques in order to deal with distinct system facets at various abstraction levels. Heterogeneous modelling approaches, by

[1] Part of the material of the current and of the next chapter was presented in [31].

allowing the abstractions which are appropriate for the different system aspects, usually result in more natural and understandable descriptions. On the other hand, any systematic analysis of heterogeneously described systems requires a coherent and consistent view of the underlying partial models, such that the relevant (global) system properties can be investigated.

Here we concentrate on discrete behavioural models of distributed embedded systems with the objective of supporting the validation process in an early design phase by the analysis of dynamic system properties.

8.2 Reliable Ground Models

Formal specification and verification methods and tools offer a variety of mathematical modelling and proof techniques supporting the entire system design process from abstract requirements specifications down to concrete realizations. At any given abstraction level, the correctness of a model M_i with respect to a more abstract model M_{i-1}, from which M_i is obtained as result of some refinement step, can (at least in principle) be proved.

There is however no way of mathematically proving correctness of the *initial* formalization step, i.e., the relation between the first formal (mathematical) model M_0 and the part of the real (physical) world this model is intended to describe. Consequently, one never gains absolute evidence on whether such a formal model faithfully represents the user's (or customer's) intuitions about the expected behaviour of a system to be designed (or the actual behaviour of an existing system), nor whether the implications resulting from the stated requirements are completely and correctly understood so that any unexpected and undesirable behaviour is a priori excluded. However, this is a crucial point, because even the most formal design process will fail to provide correct designs and implementations if the initial formalization M_0 already contains flaws.

Now, the question is: "*How can one establish that a model is faithful to reality?*". The approach documented here relies on the assumption that this can be done by constructing a mathematical model which "reflects the given system so closely that the correctness can be established by observation and experimentation" ([45], p. 9). Such a model is called a *ground model*.

In order to gain confidence in the ground model, which is the basis for the entire system design process, but can not be proved "correct" in any mathematically well-defined sense by any merely logico-mathematical means, we have to somehow *justify* its *appropriateness* (or, in other words, to *validate* it). According to [14] (Sect. 2.2), this justification process has three basic dimensions:

- **Conceptual justification**, aiming at a direct comparison of the ground model with a given part of the real world (e.g., as described by an informal specification).

- **Experimental justification**, which can be provided for the ground model by accompanying it with clearly stated system test and verifica-

tion conditions (*system acceptance plan*), i.e., falsifiability criteria which lay the ground for objectively analyzable and repeatable experiments.

- **Mathematical justification**, aiming at establishing the internal consistency of the ground model (essentially a problem of high-level reasoning and—possibly machine assisted—proof checking).

This kind of validation provides a *pragmatic foundation* for the reliability of the ground model. Taking into account possible consequences of specification errors that are not discovered in early design phases, it is certainly one of the most important steps (though often underestimated) in the system design process.

The quality of ground models considerably depends on the underlying abstractions for dealing with real-world phenomena, i.e., on the way in which basic objects and operations of a ground model are related to basic entities and actions as observed in the real world. This consideration clearly influences the choice of the formalism used to formulate the ground model: some formalisms are more appropriate than others to represent particular kinds of systems in an intuitively transparent way, as their underlying abstractions are closer to the nature of the system under consideration.

In fact, in view of the conceptual justification, the relation between the system and its formalization should be made as evident as possible: the model should be readily understood by persons who are experts of the application domain (but not necessarily of the formalism), as their judgements are essential to establish whether the model is faithful to reality. In this respect, graphical formalisms with a precise semantics can be very helpful: they are on the one hand intuitively understandable, on the other hand amenable to formal analysis.[2]

For the experimental justification, it is an advantage if the formalisms of choice are executable (as this enables the use of *simulation* as a means of experimentation), while, for the mathematical justification, it is useful to have tool support for automatic analysis/verification of particular consistency properties. Note that model-checking of temporal properties, the way it is employed in the case study of Chapter 9, is primarily a means of experimental justification (as it constitutes a systematic and highly efficient way of conducting experiments, where the temporal formulae essentially formalize the system acceptance plan), and secondarily a means of mathematical justification (as it also allows to check certain forms of internal consistency, such as the absence of conflicting updates).

8.3 Heterogeneous System Modelling

As already mentioned at the beginning of this chapter, complex technical systems are by nature extremely heterogeneous, not only because the embedded HW/SW components dedicated to their control and supervision (IT-components) may be heterogeneous, but also because they additionally include

[2]See, for instance, the case study presented in Chapter 9, where two graphical formalisms (namely, high-level Petri nets and SDL) are used to define the ground model.

a variety of mechanical and electronic devices, sensors and actuators, measuring devices, communication media, and sometimes may even involve human beings (e.g., operators).

In order to deal with all these heterogeneous entities, several modelling paradigms and languages have been developed over the years. By now, well established languages, methods and tools are available and successfully employed in industrial practice for the design of almost every single kind of system or system aspect. For instance, even if we restrict attention to the IT-related parts of technical systems, we find numerous complementary modelling paradigms (such as synchronous, asynchronous, control-flow, data-flow, with or without real-time constraints) and corresponding formalisms (Statecharts [50], synchronous languages [49], SDL [58], VHDL [57], and so on). All these techniques are (more or less) well understood and equipped with formal semantics, a design methodology and extensive tool support (including simulation, analysis, synthesis, verification, code generation). The same can not be said for their combination: not solved is, in general, the problem of how to *integrate* heterogeneous descriptions into one consistent and coherent system model which can undergo a well-defined validation and analysis process.

When modelling individual system components, one can resort to an *open system view*, where the interaction between a component and its environment is simply represented by well-defined *interfaces*.[3] From the point of view of heterogeneous modelling, an open system view has the advantage that it allows to model individual components or subsystems in isolation, without the need to model details of the surrounding environment, such that—as a consequence—different components or subsystems can be described using different (and most appropriate) formalisms. The problem is that, whenever one wants to validate the system model or verify particular properties of this model, one needs to consider the behaviour of the *overall* system, resulting from the interaction of its parts, and not the behaviour of single components or subsystems.

Thus, *composition mechanisms* are needed which allow to build a unified model of the whole system out of heterogeneous component/subsystem specifications. If one considers general composition mechanisms (such as sharing of state variables, message passing, or connection of ports), there is nothing in the composition mechanisms themselves that hinders them from working in an heterogeneous setting. However, to precisely formulate and formally analyse the behaviour of the composed system, one needs a common semantic framework (i.e., a *meta-model*) capturing and presenting in a uniform way the behaviour of both the system components and the composition mechanisms.

Thus, we are not going to introduce here new ways of composing systems nor do we discuss issues of compositionality from a theoretical point of view. Instead, we use ASMs as a meta-model for the integration of heterogeneous behavioural specifications. The approach will be exemplified in Chapter 9 by a case study, where we first identify existing description languages and composi-

[3] The open system view has been introduced, in the context of ASMs, in Sect. 1.1: however, it virtually applies to any other description technique as well.

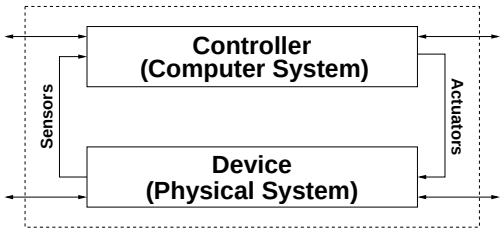


Figure 8.1: Controller-Device Interaction in Embedded Control Systems

tion mechanisms which seem appropriate for the problem at hand, then show how to express the overall system behaviour in terms of the meta-model, and finally validate the resulting system model.

In particular, the integration aspect on which we focus is the composition of a physical model of a controlled (mechanical) device with a specification of its control software, with the objective of analysing and validating the resulting behaviour against an abstract requirements specification.

8.4 Integration of Controller and Device Models

Design of embedded control systems requires to deal with two kinds of conceptually different units, the control units (*controllers*) and the controlled physical systems (*devices*), which are interfaced with each other by means of *sensors* and *actuators* (see Fig. 8.1). In addition to the connections corresponding to sensors and actuators, the picture also shows further connections to the external world. In fact, a complex technical system may consist of a network of such controller/device units (forming a *distributed* system, such as the material flow system—MFS for short—from the case study of Chapter 9).

A very important aspect in the specification of embedded control systems—especially in view of their validation and verification—is a clear separation between controller and device model, and the precise modelling of the device behaviour. Especially computer scientists often tend to overlook this aspect and only specify the controller behaviour (possibly very thoroughly and formally), but leave everything else (the "*environment*") out of the formal model.[4]

Note that the ultimate aim of the validation process is not to prove properties of the *control program* under some assumptions on the environment, but to show that the *controlled device* achieves the intended goals while ensuring some safety requirements (for instance, transporting something from one place to another, while avoiding collisions which can damage the system). Hence, we need formal models of both the controller and the device (where the device model, by itself, reflects the behaviour of the *uncontrolled* device): only on the

[4]See, for instance, [51], where a production cell controller is formally specified using SDL, but no formal model of the device is given.

basis of the combined controller/device model it is possible to precisely formulate and possibly prove properties of interest, which follow from the interaction of the control program with the physical system behaviour.

In general, the physical device is much more complex than the control program, as it involves a large number of parameters (including continuous ones). Due to this complexity, the development of a completely faithful and detailed device model is often not a feasible task. However, many details are not strictly necessary in order to formulate and prove properties of interest. Under appropriate *abstractions*, the task of modelling the physical device becomes affordable.

A typical example is *discretization*, where continuous values in the physical system are replaced by a finite set of "critical" values. Discretization is the basic abstraction tecnique employed in our MFS case study. For example, consider the "switch" device of the MFS, whose central part consists of a rotating plate: in a detailed physical model, the state of the switch plate is given by its rotation angle (a continuous value), but the essential information is whether it is in one of its stable states (left or right end position) or moving between them (clockwise or counterclockwise). A complete discretized model of the physical switch can be represented by means of a high-level Petri net, as shown in Sect. 9.3 (Fig. 9.3).

The methodological approach sketched above is very close to the one of [28], which deals with modelling and verification of the well-known case study "Production Cell" [63] by means of Statecharts. That paper also emphasizes the importance of a formal behavioural model of the physical device, as well as the need for clearly identified abstractions. With respect to [28], where Statecharts are used for modelling both the physical device and the controller, the originality of our approach is in the heterogeneous modelling.

The ability to employ different languages/formalisms within the system's ground model leads in many situations to descriptions which are more intuitive and thus easier to relate to the system being modelled (which is an essential requirement for a reliable ground model, especially in view of the conceptual justification, as discussed in Sect. 8.2). Although this kind of improvement is clearly subjective and can not be quantified, we argue that, for instance, our Petri net model of the physical switch reflects the switch device much more directly than the Statechart model in [28] does for the production cell, where much encoding overhead is present.[5]

8.5 Related Work

The topic of heterogeneous system modelling and the related integration problem are receiving increasing attention in the literature. Here we only mention the works which are most closely related to ours.

[5] On the other hand, Statecharts are a very convenient formalism to specify controllers based on finite state machines. In fact, in [28], the Statechart specification of the production cell controller is simpler and more intuitive than the corresponding model of the physical production cell. Actually, it seems that the use of Statecharts for the physical device model was mainly motivated by the verification tools requiring Statecharts as input.

In the area of hardware/software codesign, the *Ptolemy* system [22] has been developed to allow the simulation of systems whose subsystems are defined according to different models of computation, e.g., data-flow, discrete-event systems, and finite state machines (so called "co-simulation"). As a theoretical justification of the Ptolemy approach, Lee and Sangiovanni-Vincentelli define in [62] the *tagged signal model*, a denotational framework (meta-model) for comparing models of computation. In the tagged signal model, notions like "signal" and "process" are formalized in set-theoretic terms. The structure of the tag system associated to signals identifies the corresponding model of computation. For example, if the set of tags is totally ordered, the model of computation is timed: timed models can be further distinguished into continuous, discrete-event, and so on.

In the area of software specification, mainly motivated by the problem of providing a consistent formal semantics to the different notations of UML [12], Große-Rhode proposes an abstract semantic meta-model for the integration of software specification languages, called *reference model* [42]. The reference model of [42] is conceptually very close to Abstract State Machines, as it is based on transition systems with states labeled by algebras. A difference consists in the different treatment of control flow, which is strictly separated from the data. A more significant difference is that [42] only defines the meta-model in which the semantics of the object languages—e.g., the UML languages—has to be interpreted, but does not fix a (meta-)language to define such semantics, neither for specifying the control flow nor for data specification. Thus, on the one hand, the model is very general, on the other hand, it can not be used for computer-aided analysis or verification, which is in fact not an intended goal.

Different with respect to the choice of the meta-model, but closer to our approach from a methodological point of view, is the *PEP* project.[6] There, as opposite to [62] and [42], the primary objective of the integration is the computer-aided analysis of the heterogeneous system model. As meta-model for the integration a variant of high-level Petri nets—so called M-nets [9]—is chosen. After mapping the heterogeneous system model to an uniform Petri net model, according to the semantics of the respective object languages, Petri net based analysis techniques can be applied. Moreover, like the ASM Workbench, the PEP tool [40] provides a model checking interface: in addition to SMV, it also supports the SPIN model checker [53]. The overall approach is exemplified in [41] by a simple protocol, heterogeneously modelled by a finite state machine, a parallel program, and an SDL process diagram. Compared to PEP, in our ASM-based integration approach heterogeneous system models are directly mapped to a transition system model, such that the intermediate step of encoding them into Petri nets is skipped. Thus, one may expect an efficiency gain in model checking. The question whether resp. under which conditions the ASM-based approach leads to more efficient model checking is still to be investigated.

[6] PEP stands for "Programming Environment based on Petri Nets".

Chapter 9

A Case Study from Automated Manufacturing

In this chapter, in order to illustrate the methodological approach discussed in the previous chapter, we present an industrial case study from the application domain of automated manufacturing, the distributed control for a material flow system (MFS). First, we give an overview of the case study and the related problems (Sect. 9.1), then we concentrate on a subsystem of the MFS, namely the switch module (Sect. 9.2), and use it as a running example to discuss the proposed modelling and analysis techniques (Sects. 9.3-9.6).

9.1 Overview

The subject of our case study is the distributed control for a modular *material flow system*. The case study has been provided by the Computer Integrated Manufacturing (CIM) group of our institute, with which we cooperate in the research project *ISILEIT*.[1]

The MFS is based on a modern modular material flow technology that allows to build complex transportation topologies by composing standard modules (essentially: straight and curved *tracks*, *switches*, and *halting points*, see Fig. 9.1). Special vehicles called *shuttles* are employed to transport pieces over the railway between halting points (corresponding, for instance, to machines or stores). While operating, tracks are permanently supplied with current, such that shuttles keep moving over the railway in a fixed direction, unless they are arrested by means of *stopping cams* (placed at certain points along the tracks). Finally, each shuttle has a built-in device that avoids collisions by enforcing a minimum distance from the foregoing shuttle.

Traditionally, the control of a MFS is centralized: a central control unit drives the physical MFS according to predefined transport plans and routes,

[1] *ISILEIT* is a project funded by the DFG (the German Research Foundation) within the program "Integration of Software Specification Techniques for Engineering Applications".

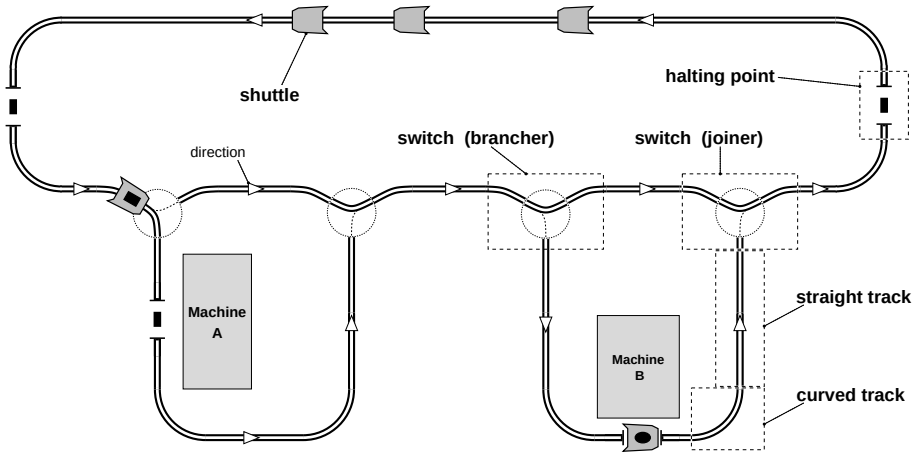


Figure 9.1: Example of Modular MFS

which implement the material flow for a given manufacturing process. However, the modular structure of the MFS sketched above suggests an alternative solution, based on *distributed control*. Instead of being controlled by a central unit, each of the modules of the MFS (e.g., shuttle, switch, or halting point) is controlled by a corresponding *local* control unit (or *control node*), cooperating with other control nodes in order to achieve the global goals of the MFS, i.e., to execute the given transportation tasks.[2] Thus, the control of the MFS is implemented by a network of concurrently operating and asynchronously communicating controllers. The only non-distributed part of the system is an host PC connected to the network for the purpose of user interaction (such as definition, transmission, and visualization of the transportation tasks), which however plays no role in the actual MFS control.

Such a distributed and modular architecture has obvious advantages over a centralized one in terms of scalability, reconfigurability, and fault tolerance, but its software is more difficult to design, implement, and validate. In fact, the implementation does not immediately reflect the system behaviour to be realized. Instead, the overall system behaviour results from the interplay of quite a large number of processes, each one executing its own protocol, having only quite limited knowledge of the system's state. How the proposed methodology can help in the design and validation of this kind of systems is what we are going to illustrate in the sequel of this chapter. In order to do this, we consider a subsystem of the MFS, namely the *switch* module.

[2]The control nodes are not necessarily dedicated processing units *physically bound* to the devices being controlled. Instead, each control node is a process *logically associated* to the controlled device. In our MFS, for instance, the control nodes for the switches are processing units physically located by the switch, while the control nodes for the shuttles—for technical reasons, such as cabling—are not located on the shuttles, but implemented on computers placed outside the physical MFS.

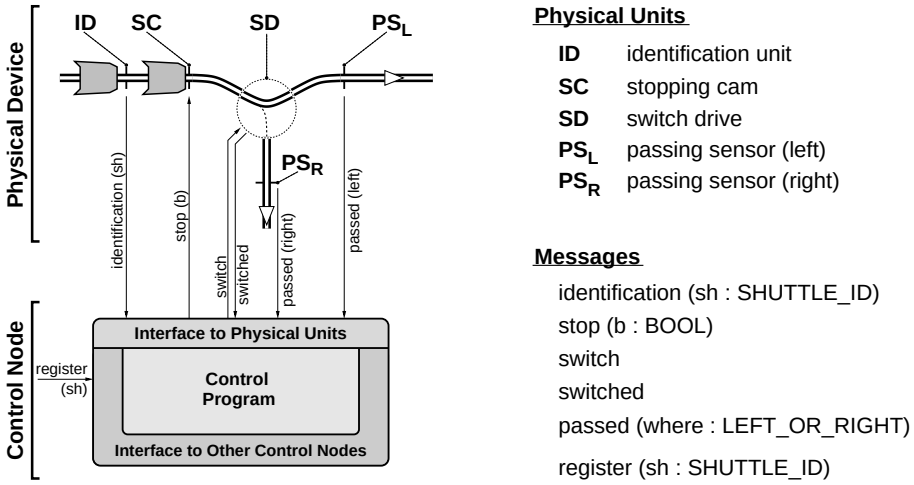


Figure 9.2: The Switch Module

9.2 The Switch Module

As shown in Fig. 9.1, there are two kinds of switch modules, called *brancher* and *joiner*, respectively. As the names suggest, a brancher directs shuttles to the one or the other destination, whereas a joiner reunites two paths into one. The brancher and the joiner differ slightly, both in their physical composition and in their control. The following discussion refers to the brancher, which is slightly more complex and interesting than the joiner.[3]

The switch module, depicted in Fig. 9.2, is built around a *switch drive* (*SD*), which can modify the state of connection between MFS tracks by inducing a rotation of the switch plate around its center. A few additional components are needed to ensure the correct and safe operation of the switch, namely:

- an *identification unit* (*ID*), which detects the passage of a shuttle and, at the same time, ascertains its identity (given by a conventional *shuttle id*);
- a *stopping cam* (*SC*), which may stop a shuttle or let it pass (recall that shuttles keep moving, if not hindered from doing so);
- *passing sensors* (*PS*), which detect the passage of a shuttle (without identifying it).

Note that the length of the track segment between ID and SC is such that at most one shuttle can occupy this segment. This is ensured by the hardware enforcement of the mininum distance, mentioned in Sect. 9.1 (see Fig. 9.2).

[3] In particular, the brancher needs to distinguish between the individual shuttles in order to direct them to the correct destination, while the joiner must simply let any incoming shuttle pass.

The interface between the physical switch and the corresponding control node is represented by an appropriate set of messages, as shown in the picture. The communication takes place over a bidirectional channel. Messages received by the controller are notifications from the sensors, messages sent by the controller are commands for the actuators. In particular, we distinguish the following message types:

- *identification*(*sh*) is sent by the identification unit whenever the shuttle with id *sh* passes over it;
- *stop*(b) controls the stopping cam: if $b = true$ it switches to the stop position, if $b = false$ it is released;
- *switch* is sent to the switch drive in order to let it start its rotation: when the rotation is completed, the switch notifies that the end position is reached by sending back the acknowledgement message *switched*;
- *passed*(*left*) and *passed*(*right*) are sent by the left and right passing sensors, respectively, whenever a shuttle passes over them.

We call the protocol regulating the communication between a control node and the corresponding physical device (which involves, in the case of the switch, the messages listed above) *low-level protocol*. A low-level protocol is responsible for ensuring local properties of the MFS modules, e.g., safety properties such as "*a shuttle does not move onto the switch plate while the plate is moving*".

The *high-level protocols*, instead, are run between neighbouring control nodes in order to direct the overall shuttle traffic in the proper way and thus cooperatively achieve the high-level goals of the MFS (i.e., executing the requested transportation tasks). In the case of a brancher, the high-level protocol is fairly simple, as its task is just to send the incoming shuttles in the "correct" direction. The information about the correct direction is provided by the halting point where a shuttle stopped before coming to the switch under consideration. As soon as the shuttle leaves the halting point, the control node of the latter "registers" the shuttle by the next (brancher) switch, to inform it about the direction that the shuttle has to take.[4]

9.3 Physical Switch: a Petri Net Model

The physical switch can be formally described by means of a high-level Petri net, shown in Fig. 9.3. This model reflects both the topology and the behaviour of the physical device. In particular, different track segments within the switch module are represented by places **S1**, **S2**, ..., each of which contains a list of

[4] Actually, as shown in Fig. 9.2, the *register* message does not mention the direction. In fact, our model follows an existing implementation that, in order to reduce message traffic, distinguishes between a "standard" and a "non-standard" direction: shuttles going in the non-standard direction are registered, while non-registered shuttles are supposed to go in the standard direction (the one expected to be taken most often).

shuttle ids as its only token.[5] The movement of shuttles between these places is represented by corresponding transitions t1, t2, ..., which update the lists appropriately (the condition on t1 reflects the assumption about the length of the track segment **S2** between ID and SC, stated informally in Sect. 9.2). The other places represent other aspects of the physical switch state, namely:

- the position of the stopping cam (which can be either in stop position or released, as reflected by the place *passing_enabled*), and

- the state of the switch plate (which can either stay in one of the end positions *left* or *right*, or be rotating from left to right or from right to left, as represented by the places *moving_lr* and *moving_rl*, respectively).

A peculiarity of the net model shown in Fig. 9.3 is that some transitions are annotated by an input or by an output symbol (whose graphical representation is borrowed from SDL). In this way, we model the interaction of the physical switch with the environment according to an open system view. Intuitively, the meaning of an input annotation is that the given transition is fired depending on a given input signal (i.e., the transition is triggered by an external event). Similarly, an output annotation means that a given output signal is emitted when the corresponding transition occurs. Note that the occurrence of such a transition is "spontaneous", in the sense that the transition happens as a possible consequence of the internal workings of the system, and not as a necessary reaction to an external stimulus.[6]

The semantics of the input and output annotations will be made precise in Sect. 9.5, where it will also be shown how the net model of the physical switch can be composed with the SDL model of its controller. By going from an open system view over to a closed system view, meaningful statements about the system behaviour can be formulated.

9.4 Switch Controller: an SDL Model

The switch controller is specified by means of SDL *process diagrams*, which represent a kind of extended finite state machines. The process diagrams for its three control states (*stable*, *waiting*, and *switching*) are shown in Fig. 9.4.

Within the process diagrams, the following (local) state variables are used:

[5] We use this representation in order to reflect the sequence of shuttles present on a track segment in a given state. This is important, as shuttles leave a track segment in the same order as they entered it, i.e., they can not overtake each other.

[6] For instance, as shuttles always keep moving (unless explicitly stopped), a given shuttle will eventually leave the track segment it occupies and enter the next one. This event is reflected by the occurrence of a transition in our net model. These transitions are typical examples of "spontaneous" or "internal" transitions. Optionally, such a transition can emit an output to notify the event to the environment, making the event visible to an external observer (in Fig. 9.3, *t1*, *t4l* and *t4r* are of this kind).

- *direction* : { *left*, *right* }, initialized with the standard direction[7] *std*, corresponds to the controller's knowledge about the switch position;
- *registered* : *SHUTTLE_ID-set*, initialized with the empty set, keeps track of the registered shuttles;
- *passing* : *INT*, initialized with 0, is a counter keeping track of the number of shuttles occupying the critical area of the switch (i.e., the switch plate and its immediate surroundings): this information is important because it is safe to activate the switch drive only when this area is free, i.e., when $passing = 0$.

Note that the state variables of the controller must be consistent with the physical system state: as this must hold, in particular, in the initial state, it implies that, when the MFS is started, its (physical) state must correspond to the controller state, e.g., all switches must be oriented in the standard direction.

We do not go into further details here, as Fig. 9.4 already contains the complete specification of the switch controller. However, we try to explain the basic idea of how the controller works in a typical case. When a shuttle arrives to the switch, it passes over the identification unit, which informs the controller. The controller, depending on the identity of the incoming shuttle, checks if the switch is already in the right position. If so, it lets the shuttle pass; otherwise it puts the stopping cam in the stop position and: *(a)* if the critical area is free, it activates the switch drive and goes into "switching" mode, where it waits for the acknowledgement message *switched* from the switch drive; *(b)* if the critical area is not free, it goes into "waiting" mode, where it waits for the critical area to become free before it can begin switching.

While executing these steps, care must be taken to keep the physical switch state and the controller state consistent. If inconsistencies arise, the controller may end in the "ERROR!" state (for which no behaviour is defined), or may behave unpredictably. Note however that, if the protocol is correct, this should only happen in case of hardware failures (e.g., if the identification unit detects passage of shuttles when there is none). Thus, a result to be expected from the analysis of the integrated controller/device model is that the controller never reaches the "ERROR!" state, as the device model (the Petri net) reflects only the failure-free behaviour of the physical switch.

9.5 Integrating Device and Controller Models

As a prerequisite for discussing details of the integration technique, we have to make precise the communication model and the semantics of the input and output annotations on net transitions in Fig. 9.3. We follow SDL in assuming an asynchronous communication model with buffered channels. Consequently, the meaning of input and output symbols in the net model can be formalized by

[7] We make use of a constant *std* standing for the standard switch direction (either *left* or *right*, depending on the MFS configuration).

adding to the net, for each (input or output) message queue, a place containing a list of messages and, for each input or output symbol on a transition, the corresponding edges to check and update the corresponding message queue, as shown in Fig. 9.5.

An aspect which is not explicitly specified, neither in the net model nor in the SDL process graphs, is the association between individual input/output commands (or individual messages) and the channels (message queues) over which they are to be sent/received. This association can in general be described by means of SDL *block diagrams*. Here, we omit the block diagram describing the connections between device and controller, as these connections are very simple: we assume that the communication between controller and device takes place over a bidirectional channel, such that two message queues (one for each channel direction) are needed. We call these message queues *sensQ* and *actQ*, standing for "sensor queue" and "actuator queue", respectively.[8]

The integration of the two switch models, the device model and the controller model, into a uniform behavioural model amenable to computer-aided analysis (simulation, model-checking) is then achieved as follows:

1. both the net model of the physical device and the SDL model of the controller are mapped to equivalent ASM models;

2. interaction between the resulting models of the two subsystems takes place by *sharing* message queues;[9]

3. an additional ASM rule specifies the coordination between activities of concurrent subsystems of the switch module and thereby complements their behavioural specifications.

The mapping from high-level Petri nets to ASMs is realised by representing each place by a corresponding dynamic function and by expressing the behaviour of each transition of the net by means of a boolean function and a transition rule in the ASM model. The boolean function corresponds to the *enabling condition* of the transition, while the transition rule specifies the *state change* produced when the transition occurs. We do not go into formal details of the transformation, which is quite straightforward, and illustrate it by the example of the switch's transition *t1* instead, which is shown in Fig. 9.6.[10]

The mapping from SDL process graphs to ASMs is based on the SDL semantics defined in [39, 11]. However, the corresponding ASM rules are simplified with respect to those which can be obtained from the SDL semantics of [39].

[8] Related to the high-level protocol, there is an additional message queue *registrQ* corresponding to the input channel on which the preceding halting point(s) send the *register* message to the switch, in order to communicate the arrival of a given shuttle. However, there will be no further mention of this in the rest of this chapter, as we focus on the device/controller integration.

[9] Note, in fact, that the inputs for the controller are the outputs of the device (and vice versa). The unifying ASM model of the controller-device system makes the interaction explicit by showing how the controller and the device actually read/write those messages from/into the same message queue.

[10] See also [38] which deals with the integration of Pr/T nets and ASMs.

In particular, we assume that an SDL transition is executed in one ASM step, whereas the single actions of a transition should actually be executed in more steps, sequentially. This simplication is important for the subsequent analysis of the model (especially by model-checking) in order to reduce the size of the state space. Of course, care should be taken in order to preserve the SDL transition semantics. Fig. 9.7 shows the ASM rule corresponding to the process graph for the switching state.

The mechanism which allows the interaction of the controller and device models consists in sharing the message queues *actQ* and *sensQ*, which are declared in the ASM model as

```
dynamic function actQ  :LIST(MESSAGE)  initially [ ]
dynamic function sensQ :LIST(MESSAGE)  initially [ ].
```

Note, for instance, that the transition t1 of the device model in Fig. 9.6 writes to `sensQ`, while the process graph for the switching state of the controller model in Fig. 9.7 reads `sensQ`, triggering an SDL transition if an appropriate message is present at the head of `sensQ` (which is the input queue for the SDL process).

Finally, to complete the model integration, *coordination* rules have to be specified to define a discipline by which all concurrently operating parts of the system (represented, in our model, by the single net transitions and the control process) cooperate to achieve the overall system goals.[11] We adopt a simple interleaving model of concurrency and define a main coordination rule (the ASM *program*) which non-deterministically chooses whether the controller or the physical switch makes a move (`phys_switch_step` and `controller_step` are the subrules corresponding to a move of the physical switch or of the controller, respectively):

```
external function choose_round
with choose_round in { phys_switch, controller }

transition Main ==
  case choose_round of
    phys_switch : phys_switch_step ;
    controller  : controller_step
  endcase
```

Moreover, we have to model the implicit assumption that, while the transitions corresponding to shuttles passing track segment boundaries ("shuttle transitions") happen after some time, the transitions corresponding to *reactions* of both the controller and the physical switch to incoming messages ("reaction transitions", triggered by sensors/actors) happen "immediately".[12] A simple

[11] Coordination rules are either derived from the actual physical realization of the system (thus stating assumptions on the environment) or express requirements regarding the control software (thus stating guidelines for its implementation).

[12] "Immediately" does not mean here that these actions take no time, but that the time taken is negligible compared to the time needed for other actions, in particular for a shuttle to traverse a track segment (in fact, the order of magnitude is of seconds for the latter, of milliseconds for the former). Note that we want to avoid explicit mention of time constraints here, as this would be a kind of over-specification at this level of abstraction.

way to do this is to prioritise the transitions of the net model, such that shuttle transitions always have lower priority than reaction transitions. This can be achieved by an appropriate definition of `phys_switch_step`, the coordination rule for the physical switch (net model).

9.6 Analysis and Validation

Analysis and validation are essentially performed by means of simulation and model-checking. In particular, the combination of both is very effective in "debugging" the high-level system specification. The model-checker can be used to find counterexamples, i.e., runs which contradict the expected behaviour (specified by a set of properties expressed in temporal logic). Each counterexample can then be fed into the simulator in order to find out the origin of the wrong behaviour. Debugging features, which are very helpful in this regard, include possibility of executing single steps forward and backward, a sequence of steps until a given condition is satisfied, inspection of the system state, etc. After the specification is fixed, the whole process is reiterated, until all properties are satisfied.

The properties to be checked are specified in CTL (Computation Tree Logic), the temporal logic supported by the model checker SMV [66], which we employ for the verification task. We consider both *safety* and *liveness* properties, which altogether build an abstract *requirements specification* for the switch module:[13]

1. Safety (controller). "*The controller never reaches the* ERROR! *state*" (in fact, the ERROR! state indicates that some hardware failure happened):

   $$\neg\mathsf{EF}\ (\text{control\_state} = \text{ERROR!})$$

2. Safety (device). "*A shuttle does not move onto the switch plate while the plate is moving*". This is ensured by requiring that the critical area of the switch, consisting of the switch plate and its immediate surrounding (places S3, S4L, S4R in the model), is always free whenever the switch plate is moving:

   $$\mathsf{AG}\ (\text{moving\_lr} \vee \text{moving\_rl}\ \Rightarrow\ \text{S3} = [\,] \wedge \text{S4L} = [\,] \wedge \text{S4R} = [\,])$$

3. Liveness. "*All shuttles entering the switch will eventually leave it: unregistered shuttles leave it in the standard direction, registered ones in the non-standard direction.*": for each shuttle id "sh"

   $$\mathsf{AG}\ (\mathit{contains}(\text{S1}, \text{sh}) \wedge \neg(\text{sh} \in \mathit{registered}) \Rightarrow \mathsf{AF}\ \mathit{contains}(\text{S5L}, \text{sh}))$$
   $$\mathsf{AG}\ (\mathit{contains}(\text{S1}, \text{sh}) \wedge \text{sh} \in \mathit{registered} \Rightarrow \mathsf{AF}\ \mathit{contains}(\text{S5R}, \text{sh}))$$

   where *contains* is a static predicate which tests if a given list contains a given element (for simplicity, we show the formulae for the special case *std* = *left*; for the case *std* = *right*, simply exchange S5L and S5R).

[13] Intuitively, the *safety* properties correspond to requirements of the kind "*something bad never happens*", the *liveness* properties to requirements of the kind "*something good eventually happens*" (the system should not only operate safely, but also accomplish its task).

Note that, thanks to our integration approach, we can freely mix state variables of the controller model (which are actually *program variables*) and state variables of the device model (which are an abstraction of the *physical state* of the switch) within our abstract requirements specification. The best example of this is given by the two properties in (3.), where variables of both models occur within the same formula, in order to formalize the causal relation between the controller state at a given moment and the resulting physical system behaviour.

We could verify the properties above for instances of the problem with one, two, and three incoming shuttles. During the validation process (carried out according to the methodology sketched above, based on iteration of the model-checking/counterexample-simulation cycle), we detected a bug in the controller specification (in the control state *switching*, we forgot to increase the *passing* counter on releasing the stopping cam), and found out that the assumptions about immediate reaction of some transitions—mentioned at the end of Sect. 9.5—are essential for the correct working of the system.

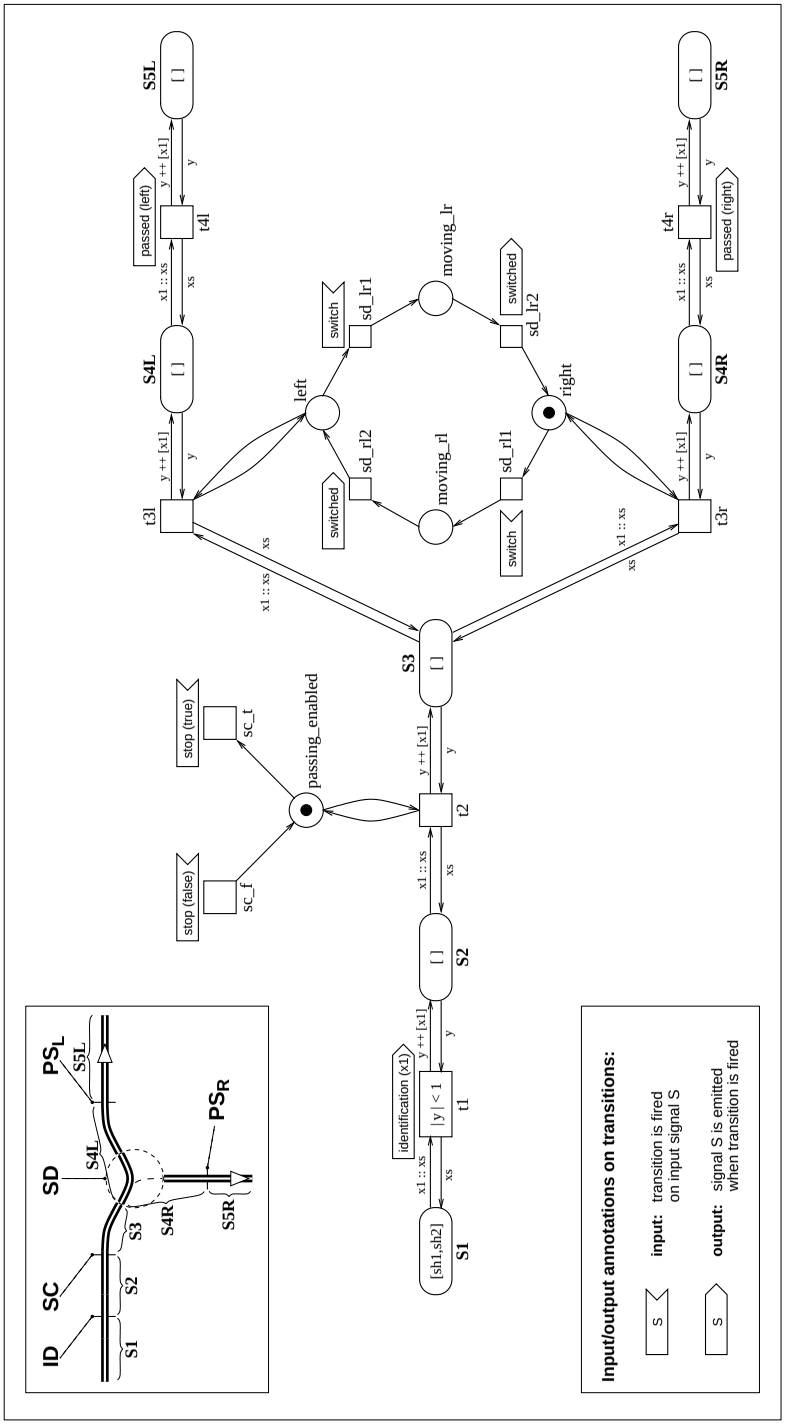


Figure 9.3: Physical Model of the Switch

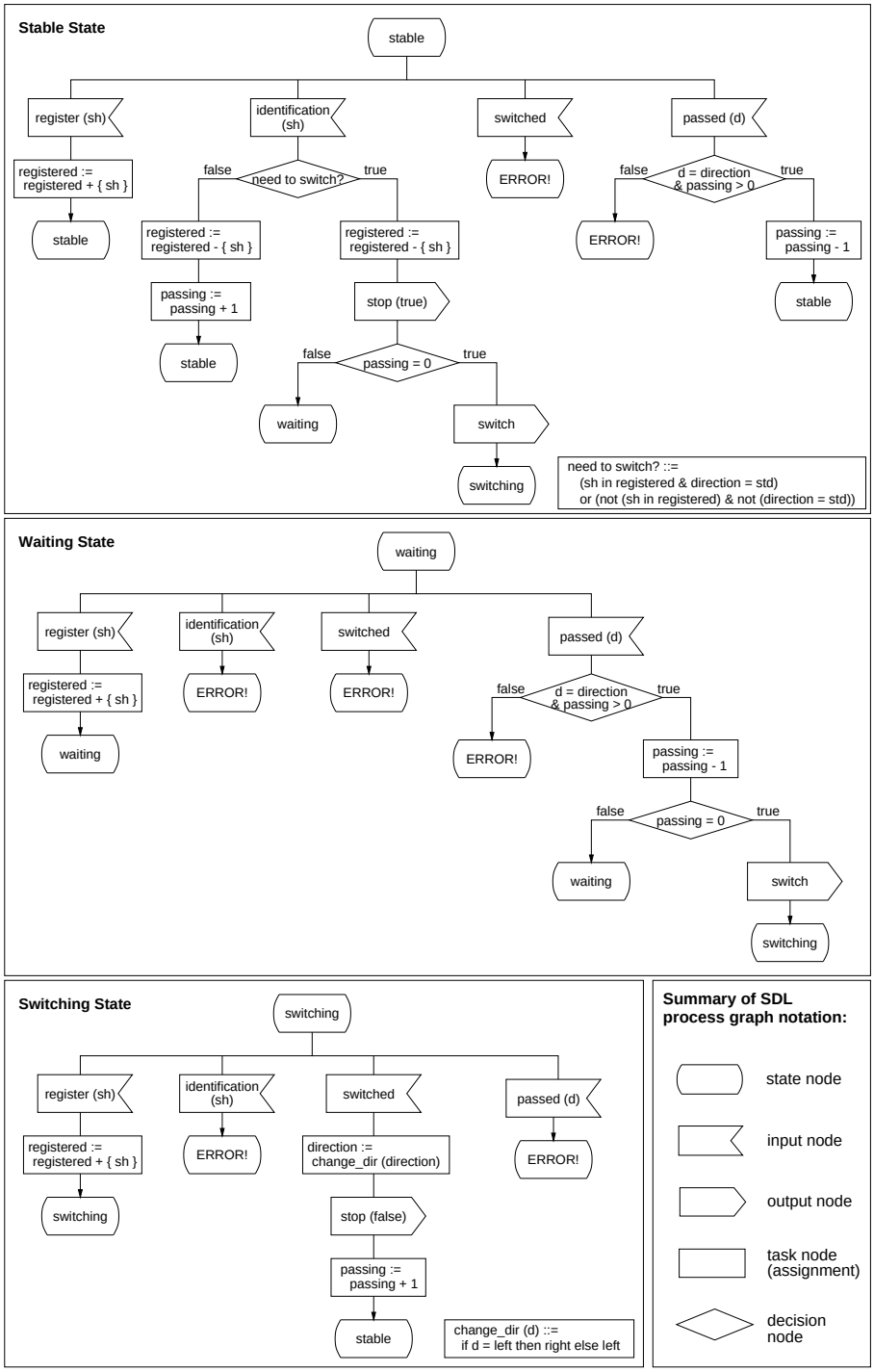


Figure 9.4: Switch Controller

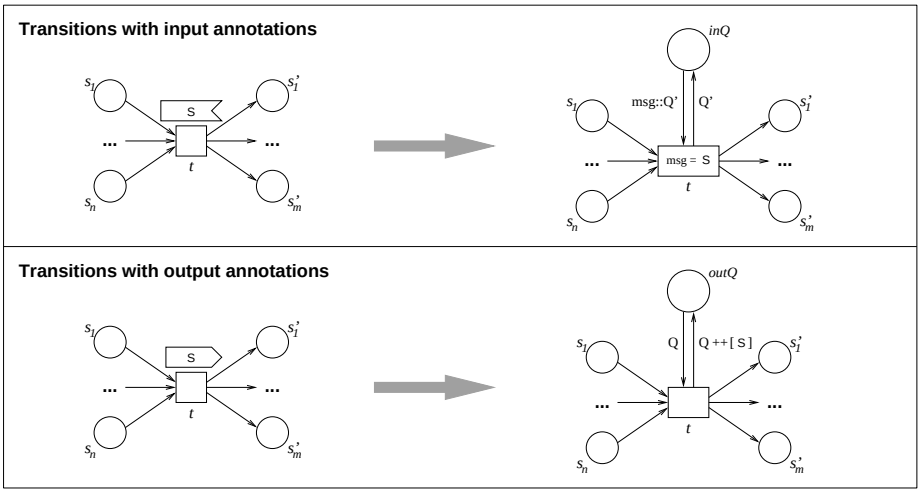
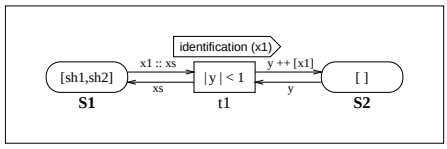


Figure 9.5: Input/Output Annotations on Net Transitions




```
derived function t1_enabled ==
case (S1, S2, sensQ) of
  (x1::xs, y, Q) : length (y) < 1;
  otherwise false
endcase
```

```
transition t1_action ==
case (S1, S2, sensQ) of
  (x1 :: xs, y, Q) :
    if length (y) < 1
    then S1 := xs
         S2 := y @ [ x1 ]
         sensQ := Q @
           [ identification (x1) ]
    endif
endcase
```

Figure 9.6: Mapping from High-Level Petri Nets to ASMs (transition: t1)

```
transition Switching ==
  case sensQ of
    msg :: restQ :
      case msg of
        identification (sh) : control_state := ERROR ;
        passed (d)          : control_state := ERROR ;
        switched            : direction  := change_dir (direction)
                              actQ := actQ @ [ stop (false) ]
                              passing := passing + 1
                              control_state := stable
      endcase
  endcase
```

Figure 9.7: Mapping from SDL Process Graphs to ASMs (state: switching)

Conclusions

In this thesis a specification language based on Gurevich's Abstract State Machines, called *ASM-SL*, has been defined. Related analysis techniques and supporting tools, collectively referred to as *ASM Workbench*, have been introduced. The appropriateness of the language and tools to concrete engineering tasks has been demonstrated by case studies, the most relevant being the *FALKO* project mentioned in Chapter 6 and the *ISILEIT* project discussed in Chapters 8 and 9. The experiences collected can be summarized as follows.

The *specification language* seems appropriate for a wide range of different applications. In particular, the choice of a model-based approach for data specification, although less general than a purely axiomatic approach such as algebraic specification, is flexible enough to deal with a variety of different applications in the fields of software engineering and design of embedded HW/SW systems, especially controllers. It also has the advantage of being constructive and quite intuitive. However, for particular application domains one may need data structures that can not be easily or efficiently encoded into those provided by ASM-SL, for example, bit vectors of arbitrary length with the corresponding arithmetic as needed for the exact modelling of hardware.

The *type system* based on ML-like parametric polymorphism fits well the specification language. The static type checking, which includes type inference, helps to detect trivial mistakes very early, before undertaking any simulation or verification, while the notational overhead due to type annotations is almost completely avoided. This is a considerable advantage with respect to the original untyped definition of ASMs. However, this type system is not always convenient. Problems may arise when using the Workbench to mechanize existing ASM applications, where a much more general notion of type is used, namely predicates as types. For instance, subtyping—which naturally arises from the notion of inheritance in specifications of object-oriented designs—can be easily accommodated in the "predicates as types" framework, but can not be conveniently represented in the type system of ASM-SL.

The problem of *specification in the large* was not considered in this thesis. No mechanisms for structuring large specifications are currently supported by ASM-SL. This was not a problem in the mentioned case studies. In FALKO the application under study was modelled at such a high level of abstraction—by using very rich data structures and complex functions—that no further operational decomposition was needed. In ISILEIT the single system components,

such as the switch module, were modelled and verified in isolation, according to the open system view: each system component was considered as an elementary unit with appropriate interfaces and did not need to be further decomposed. However, for modelling complex systems as resulting from the composition of simpler subsystems, appropriate mechanisms have to be provided. For instance, an appropriate notion of composition will need to be defined in order to derive the global behaviour of the ISILEIT material flow system from the interaction of the local behaviours of its components.

Regarding the *analysis techniques*, in particular via model checking, the transformation schemes presented in Chapter 7 provide a first useful approach to automatic verification of (finite-state) ASM models. In particular, the unfolding transformation introduced here is the first technique for model checking a consistent subset of the ASM language. Its practical applicability has been demonstrated, in Chapter 9, in the context of the ISILEIT case study. Clearly, there is still much place for improvements, in particular to increase the size of models feasible for model checking or allow the verification of infinite models through abstraction techniques.

The *tool infrastructure* of the ASM Workbench, described in Chapter 6, provides a number of basic functionalities, such as parsing, abstract syntax trees, type checking, pretty printing, and evaluation of terms and rules, which can be reused for the development of further ASM tools. Tool developers can also take advantage of the generic mechanisms based on the structural induction principle, by which the effort needed for implementing syntax-directed transformations can be reduced to a minimum. A typical application of the mentioned infrastructure is the ASM2SMV tool, which translates ASM-SL into the input language of the SMV model checker.

From the considerations above the following conclusions, which also suggest directions for further work, can be drawn.

The ASM Workbench tool environment and its source language ASM-SL provide, in general, useful support for computer-aided specification, modelling, analysis, and validation, based on the method of Abstract State Machines. Both the language and the tools are quite general-purpose and result from a compromise between different needs arising in different application domains.

As a consequence, the ASM Workbench is applicable to a wide range of applications, but not optimally suited to any of them. Specialized variants of the language and of the tools may be more useful in particular application domains. For instance, a type system closer to those found in object-oriented languages would be preferable for applications in software engineering, while bit-vector arithmetic would be desirable for hardware modelling. Moreover, structuring and modularization constructs will be needed to deal with large applications, while the current analysis techniques are subject to further improvement. In this context, the tool infrastructure provided by the ASM Workbench should provide a good starting point for designing ASM-based languages tailored to particular application domains, implementing language extensions such as modularization constructs, improving the existing transformation and analysis techniques or defining new ones, and, in general, experimenting with Abstract State Machines.

Part IV

Appendices

Appendix A

ASM-SL Lexical Structure

In this appendix we define the structure of the lexical elements (tokens) occurring in the grammar which defines the concrete syntax of ASM-SL (introduced in Chapter 3 and summarized in Appendix C).

Notational conventions The lexical structure of the tokens is defined using standard EBNF notation. Literals are are set in `small typewriter` font and so are they distinguished from other terminal and non-terminal symbols (set in *small italic* font), and from the EBNF metasymbols "{", "}", "|", "[", "]" (set in the regular text font).[1] In the rest of this appendix we will use the following auxiliary symbols:

digit → `0` | `1` | ... | `9`
letter → `A` | `B` | ... | `Z` | `a` | `b` | ... | `z`

A.1 Keywords

The words in the following lists are *reserved words* (*keywords*), which have a special meaning in ASM-SL and should not be used as identifiers:

`->`	`..`	`:=`	`==`	`_`
`FUN_TO_MAP`	`MAP_TO_FUN`	`REL_TO_SET`	`SET_TO_REL`	`block`
`case`	`choose`	`datatype`	`datatypes`	`derived`
`do`	`dynamic`	`else`	`elseif`	`end`
`endblock`	`endcase`	`endchoose`	`enddo`	`endif`
`endlet`	`endvar`	`exists`	`external`	`fn`
`forall`	`freetype`	`freetypes`	`function`	`functions`
`if`	`in`	`initially`	`let`	`of`
`op`	`op_l`	`op_r`	`otherwise`	`relation`
`skip`	`static`	`then`	`tn`	`transition`
`typealias`	`var`	`with`		

[1] Note that, within productions, EBNF metasymbols are slightly taller than the same characters appearing as literals.

A.2 Type Variables

typevar → ' *letter* (*letter* | *digit* | _)*

A.3 Special Constants

Special constants denote integers (terminal *int_const*), floating point numbers (terminal *float_const*) and strings (terminal *string_const*).

Integer Constants

int_const → *digit* (*digit*)*

Floating Point Constants

float_const → *digit* (*digit*)* . *digit* (*digit*)*

String Constants

printable_char → *letter* | *digit* | ␣ | ! | # | $ | % | & | ' | (|) | * | + | , | - | . | / | : | ; | < | = | > | ? | @ | [|] | ^ | _ | ` | { | | | } | ~

quoted_char → \\ | \" | \t | \n

ascii_code → \ *digit digit digit*

string_const → " (*printable_char* | *quoted_char* | *ascii_code*)* "

A.4 Identifiers

sym_id_char → ! | % | & | $ | # | + | - | / | : | < | = | > | ? | | | \ | ~ | ` | ^ | | | *

id → *letter* (*letter* | *digit* | _)*
| (*sym_id_char*)*

The second form of identifier ("symbolic identifier") is usually, but not necessarily, used to define infix operators. To improve the readability of the productions, in Chapter 3 and Appendix C it is distinguished between different kinds of identifiers (e.g., *var_id* stands for variables, *fun_id* for function names, and so on—see Sect. 3.1 for details).

A.5 Infix Operators

As it is possible to use infix notation in terms, some binary function names may be labelled as *infix operators*: infix operators can be left- or right-associative

and are given a priority from 0 to 9, where 9 indicates the strongest and 0 the weakest associativity.[2]

Some infix operators (e.g., `+` and `*`) are predefined. The user can define infix operators by prefixing the (binary) function name with the keywords `op_l` (for left-associative operators) or `op_r` (for right-associative operators), optionally followed by the operator priority (the default priority is 0). Example:

```
static function op_l 2 ++ (x, y) == x + y
```

defines a left-associative operator `++` with priority 2. (See Sect. 3.8.2 for the syntax of function definitions).

In the grammar productions of Chapter 3 and Appendix C, infix operators are denoted by the symbol *infix_op* (note that infix operators have exactly the same lexical structure as any other identifier, as defined above). The associativies and priorities of built-in ASM-SL operators are as follows.

Name	Type	Assoc.	Priority
`*`	$\texttt{INT} \times \texttt{INT} \to \texttt{INT}$	left	7
`div`	$\texttt{INT} \times \texttt{INT} \to \texttt{INT}$	left	7
`mod`	$\texttt{INT} \times \texttt{INT} \to \texttt{INT}$	left	7
`+`	$\texttt{INT} \times \texttt{INT} \to \texttt{INT}$	left	6
`-`	$\texttt{INT} \times \texttt{INT} \to \texttt{INT}$	left	6
`intersect`	$\texttt{SET}(\alpha) \times \texttt{SET}(\alpha) \to \texttt{SET}(\alpha)$	left	6
`\`	$\texttt{SET}(\alpha) \times \texttt{SET}(\alpha) \to \texttt{SET}(\alpha)$	left	5
`union`	$\texttt{SET}(\alpha) \times \texttt{SET}(\alpha) \to \texttt{SET}(\alpha)$	left	4
`=`	$\alpha \times \alpha \to \texttt{BOOL}$	left	4
`!=`	$\alpha \times \alpha \to \texttt{BOOL}$	left	4
`<`	$\alpha \times \alpha \to \texttt{BOOL}$	left	4
`<=`	$\alpha \times \alpha \to \texttt{BOOL}$	left	4
`>`	$\alpha \times \alpha \to \texttt{BOOL}$	left	4
`>=`	$\alpha \times \alpha \to \texttt{BOOL}$	left	4
`@`	$\texttt{LIST}(\alpha) \times \texttt{LIST}(\alpha) \to \texttt{LIST}(\alpha)$	right	5
`::`	$\alpha \times \texttt{LIST}(\alpha) \to \texttt{LIST}(\alpha)$	right	1
`##`	$\texttt{STRING} \times \texttt{STRING} \to \texttt{STRING}$	left	1
`and`	$\texttt{BOOL} \times \texttt{BOOL} \to \texttt{BOOL}$	left	1
`or`	$\texttt{BOOL} \times \texttt{BOOL} \to \texttt{BOOL}$	left	0

A.6 Comments

Within an ASM-SL specification it is possible to introduce both single-line and multi-line comments. A single-line comment is introduced by `//` and extends until the end of the line. A multi-line comment is enclosed between `(*` and `*)`. Multi-line comments can be nested. In general, comments are simply skipped by the lexical analyzer and thus ignored.

[2] We adopt the convention that, if an ambiguity between a left- and a right-associative operator of the same priority arises, the left-associative has priority.

Appendix B

ASM-SL Primitive Types and Functions

In this appendix the primitive types and function of ASM-SL are listed. The primitive types and functions are part of the *initial context*, as defined in Sect. 3.9, and can be used in any ASM-SL specification.

B.1 Primitive Types

Primitive types are:

- The elementary data types `BOOL`, `INT`, `FLOAT`, and `STRING`, which stand for booleans, integers, floating-point numbers, and strings, respectively. (Strings are sequences of ASCII-encoded characters of arbitrary length.)
- The polymorphic types `LIST`, `SET`, and `MAP`. The type constructor `LIST` is unary and denotes the free type of lists, which would be defined as

  ```
  freetype LIST('a) == {
    op_r 1 :: : 'a * LIST('a) -> 'a,
    nil       : 'a
  }
  ```

  if it was not predefined. The type constructor `SET` is unary. `SET`(τ) denotes the type of finite sets with elements of type τ. The type constructor `MAP` is binary. `MAP`(τ_1, τ_2) denotes the type of finite maps with range τ_1 and domain τ_2. (See also Table 5.1.)

B.2 Primitive Functions

Primitive functions are listed below. For each primitive function we specify the signature and give a short explanation of the meaning, when the meaning is

not standard or obvious. All primitive functions are *static*. Some primitive functions are *constructors*, precisely: the boolean constants `true` and `false`, the `undef` constant, and the list constructors `::` and `nil`.

B.2.1 Boolean Constants and Operations

```
true  : BOOL
false : BOOL

op_l 1 and : BOOL * BOOL -> BOOL
op_l 0 or  : BOOL * BOOL -> BOOL
not : BOOL -> BOOL
```

B.2.2 The Undef Constant

```
undef : 'u'a
```

B.2.3 Comparison Operators

Comparison operators are defined with polymorphic type. They have different interpretations depending on the type of their arguments.

```
op_l 4 =  : 'a * 'a -> BOOL
op_l 4 != : 'a * 'a -> BOOL
op_l 4 <  : 'a * 'a -> BOOL
op_l 4 <= : 'a * 'a -> BOOL
op_l 4 >  : 'a * 'a -> BOOL
op_l 4 >= : 'a * 'a -> BOOL
```

For integer and floating-point numbers, the order relation is the usual one. Strings are ordered lexicographically, according to the ASCII codes of the characters of which they are composed. For finite sets, `<=` is interpreted as the subset relation (and, obviously, `<` stands for "proper subset", and so on). The same holds for finite maps, with maps viewed as the corresponding sets of pairs. For all other types, comparison operators are interpreted as constantly false, with the exception of `=` and `!=`, which always denote the equality and its negation.

B.2.4 Integer Arithmetic

The usual integer arithmetic is available through the following operators.

```
op_l 6 + : INT * INT -> INT
op_l 6 - : INT * INT -> INT
op_l 7 * : INT * INT -> INT
op_l 7 div : INT * INT -> INT
op_l 7 mod : INT * INT -> INT
~ : INT -> INT

abs : INT -> INT
```

Moreover, the following functions perform bitwise logical and shift operations (bitwise `and`, `or`, `xor`, `not`, shift left, and shift right, respectively).

```
andb : INT * INT -> INT
orb  : INT * INT -> INT
xorb : INT * INT -> INT
notb : INT * INT -> INT
lsh  : INT * INT -> INT
rsh  : INT * INT -> INT
```

B.2.5 Floating-Point Arithmetic

The usual floating-point arithmetic is available through the following functions.

```
fadd : FLOAT * FLOAT -> FLOAT
fsub : FLOAT * FLOAT -> FLOAT
fmul : FLOAT * FLOAT -> FLOAT
fdiv : FLOAT * FLOAT -> FLOAT
fneg : FLOAT -> FLOAT

sqrt : FLOAT -> FLOAT
exp : FLOAT -> FLOAT
ln : FLOAT -> FLOAT
sin : FLOAT -> FLOAT
cos : FLOAT -> FLOAT
arctan : FLOAT -> FLOAT
```

B.2.6 String Operations

The `ord` function converts the first character of a string into its ASCII code, while `chr` converts an ASCII code into the corresponding character (a string of length 1):

```
ord  : STRING -> INT
chr  : INT -> STRING
```

String concatenation operator:

```
op_l 1 ## : STRING * STRING -> STRING
```

B.2.7 List Operations

List constructors:

```
nil : LIST('a)
op_r 1 ::  : 'a * LIST('a) -> LIST('a)
```

Head and tail of a list:

```
hd : LIST('a) -> 'a
tl : LIST('a) -> LIST('a)
```

The `list_interval` functions constructs lists representing intervals:

```
list_interval : INT * INT * INT -> LIST(INT)
```

Its semantics is:

$$\texttt{list\_interval}(n_1, n_2, s) = \\ = \begin{cases} [n_1, n_1 + s, \ldots, n_1 + ks] & \text{if } (n_2 > n_1 \text{ and } s > 0) \text{ or } (n_2 < n_1 \text{ and } s < 0) \\ [\,n_1\,] & \text{if } n_1 = n_2 \\ [\,] & \text{if } (n_2 > n_1 \text{ and } s < 0) \text{ or } (n_2 < n_1 \text{ and } s > 0) \end{cases}$$

where $k = \lfloor (n_2 - n_1)/s \rfloor$.

The `length` function yields the length of a list:

```
length : LIST('a) -> INT
```

With respect to list concatenation, the `append` function concatenates two lists, the operator `@` is a synonym of `append`, and `concat` computes the concatenation of a list of lists:

```
append : LIST('a) * LIST('a) -> LIST('a)
op_r 5 @ : LIST('a) * LIST('a) -> LIST('a)

concat : LIST(LIST('a)) -> LIST('a)
```

B.2.8 Set Operations

On finite sets the usual set operations $\emptyset$, $\in$, $\times$, $\cap$, $\setminus$, and $\cup$, are defined, with the following signature:

```
emptyset : SET('a)
member : 'a * SET('a) -> BOOL
prod : SET('a) * SET('b) -> SET('a * 'b)

op_l 6 intersect : SET('a) * SET('a) -> SET('a)
op_l 5 \         : SET('a) * SET('a) -> SET('a)
op_l 4 union     : SET('a) * SET('a) -> SET('a)
```

The `set_interval` functions constructs set representing intervals:

```
set_interval : INT * INT * INT -> INT
```

Its semantics is:

$$\texttt{set\_interval}(n_1, n_2, s) = \texttt{list\_to\_set}(\texttt{list\_interval}(n_1, n_2, s))$$

The `card` function yields the cardinality of a finite set:

```
card : SET('a) -> INT
```

The `element_of` function extracts the element of a singleton set:

```
element_of : SET('a) -> 'a
```

Its semantics is: `element_of`$(\{x\}) = x$, `element_of`$(X) =$ **undef** if $\mid X \mid \neq 1$.

Finally, there are two function `Union` and `Intersect` to compute the union and the intersection of a set of sets:

```
Union     : SET(SET('a)) -> 'a
Intersect : SET(SET('a)) -> 'a
```

B.2.9 Map Operations

On finite maps the following operations are defined:

```
emptymap    : MAP('a,'b)
apply       : MAP('a,'u'b) * 'a -> 'u'b
map_union   : MAP('a,'b) * MAP('a,'b) -> MAP('a,'b)
override    : MAP('a,'b) * MAP('a,'b) -> MAP('a,'b)
domain      : MAP('a,'b) -> SET('a)
range       : MAP('a,'b) -> SET('b)
map_card    : MAP('a,'b) -> INT
```

The `emptymap` constant denotes an empty map. The `apply` function is the functional application of a map M to an argument x, which yields **undef** if M is not defined on x, i.e., $x \notin \text{dom } M$ (thus, the range of the map must be an u-type). The `map_union` and `override` functions correspond to the operators $\cup$ and $\oplus$ defined in *Notational Conventions* (p. 3). Finally, `domain`, `range` and `map_card` yield domain, range and cardinality of the given finite map, respectively.

B.2.10 Conversions

The `floor` and `round` functions convert a floating-point number into an integer by truncating or rounding it, respectively. The `int_to_float` function, instead, converts an integer into a floating-point number.

```
floor : FLOAT -> INT
round : FLOAT -> INT
int_to_float : INT -> FLOAT
```

Finally, the following functions are provided for converting between lists, finite sets, and finite maps.

```
list_to_set : LIST('a) -> SET('a)
set_to_list : SET('a) -> LIST('a)
map_to_set  : MAP('a,'b) -> SET('a * 'b)
set_to_map  : SET('a * 'b) -> MAP('a,'b)
```

Note that:

- `set_to_list`(X) yields a list containing all elements of X in some arbitrary order. However, the order is the same in every state, as the function is static.
- `set_to_map`(X) yields **undef** if the set X contains two elements (x, y) and (x, y') with $y \neq y'$.

Appendix C

ASM-SL Concrete Syntax

In this appendix we summarize the concrete syntax of ASM-SL, as defined in Chapter 3. For each syntactic category, grammar productions for basic and derived forms are collected, followed by a table summarizing the rewriting rules for reducing derived forms to equivalent basic forms. Sometimes short forms of grammar symbols are used, e.g., t for *term* (see Table 3.1 for the complete list).

C.1 Types

Basic Forms

type	→	*typevar*	type variable
	|	`(` `)`	empty tuple type
	|	*type* `*` *type* (`*` *type*)*	tuple type (n-ary, $n > 1$)
	|	*type_id*	type construction (nullary)
	|	*type_id* `(` *type* (`,` *type*)* `)`	type construction (n-ary, $n > 0$)
func_type	→	$type_1$ `->` $type_2$	function type

Derived Forms

type	→	`(` *type* `)`	parenthesized type
	|	`(` *type* `,` *type* (`,` *type*)* `)`	tuple type
	|	`[` *type* `]`	list type
	|	`{` *type* `}`	(finite) set type
	|	`{` *type* `->` *type* `}`	(finite) map type

Derived form	Equivalent basic form
`(` *type* `)`	*type*
`(` $type_1$ `,` $type_2$ (`,` $type_i$)* `)`	$type_1$ `*` $type_2$ (`*` $type_i$)*
`[` *type* `]`	`LIST (` *type* `)`
`{` *type* `}`	`SET (` *type* `)`
`{` $type_1$ `->` $type_2$ `}`	`MAP (` $type_1$ `,` $type_2$ `)`

C.2 Patterns

Basic Forms

patt	→	*int_const*	integer constant
	|	*float_const*	floating-point constant
	|	*string_const*	string constant
	|	*var_id*	variable (binding occurrence)
	|	`(` *var_id* `:` *type* `)`	idem, with type constraint
	|	`_`	placeholder
	|	`(` [*patt* (`,` *patt*)*] `)`	tuple pattern
	|	*fun_id* `(` *patt* `)`	constructor application

Derived Forms

patt	→	*fun_id*	constructor application (nullary)
	|	*fun_id* `(` *patt* `,` *patt* (`,` *patt*)* `)`	constructor application (n-ary, $n > 1$)
	|	*patt infix_op patt*	constructor application (infixed)
	|	`[ ]`	list pattern (empty)
	|	`[` *patt* (`,` *patt*)* `]`	list pattern (non-empty)

Derived form	**Equivalent basic form**
fun_id	*fun_id* `( )`
fun_id `(` $patt_1$ `,` $patt_2$ (`,` $patt_i$)* `)`	*fun_id* `(` `(` $patt_1$ `,` $patt_2$ (`,` $patt_i$)* `)` `)`
$patt_1$ *infix_op* $patt_2$	`op` *infix_op* `(` $patt_1$ `,` $patt_2$ `)`
`[ ]`	`nil`
`[` p_1 (`,` p_i)* `]`	p_1 (`::` p_i)* `::` `nil`

C.3 Terms

Basic Forms

term	→	*int_const*	integer constant
	|	*float_const*	floating-point constant
	|	*string_const*	string constant
	|	*var_id*	variable (applied occurrence)
	|	`(` [*term* (`,` *term*)*] `)`	tuple term
	|	*fun_id* `(` *term* `)`	function application
	|	`if` *term* `then` *term* `else` *term* `endif`	(basic) conditional term

	|	`case` *term* `of`
		patt `:` *term* `;`
		`otherwise` *term*
		`endcase`
			(basic) case term

	|	`[` *term* `|` *patt* `in` *term* [`with` *term*] `]`	list comprehension
	|	`FUN_TO_MAP` *fun_id*	function to finite map conversion
	|	`REL_TO_SET` *fun_id*	relation to finite set conversion

Derived Forms

term	→	*fun_id*	function application (nullary)
	|	*fun_id* `(` *term* `,` *term* (`,` *term*)* `)`	function application (n-ary, $n > 1$)
	|	*term infix_op term*	function application (infixed)
	|	`if` *term* `then` *term* (`elseif` *term* `then` *term*)* [`else` *term*] `endif`	(derived) conditional term

	|	`case` *term* `of`
		patt `:` *term*
		(`;` *patt* `:` *term*)*
		[`;` `otherwise` *term*]
		`endcase`
			(derived) case term

	|	`let` *patt* `==` *term* `in` *term* `endlet`	let term
	|	`[ ]`	empty list
	|	`[` *term* (`,` *term*)* `]`	list enumeration (n elements, $n > 0$)
	|	{ }	empty set
	|	{ *term* (`,` *term*)* }	set enumeration (n elements, $n > 0$)
	|	{ *term* `->` *term* (`,` *term* `->` *term*)* }	map enumeration (n elements, $n > 0$)

|	[*term* .. *term*]	list interval
|	{ *term* .. *term* }	set interval
|	{ *term* | *patt* in *term* [with *term*] }	set comprehension
|	{ $term_1$ -> $term_2$ | *patt* in *term* [with *term*] }	map comprehension
|	(exists *patt* in *term* [: *term*])	(finite) existential quantification
|	(forall *patt* in *term* [: *term*])	(finite) universal quantification

Derived form	**Equivalent basic form**
fun_id	*fun_id* ( )
fun_id ($term_1$, $term_2$ (, $term_i$)*)	*fun_id* (($term_1$, $term_2$ (, $term_i$)*))
$term_1$ *infix_op* $term_2$	op *infix_op* ($term_1$, $term_2$)
if G then t endif	if G then t else undef endif
if G_1 then t_1 elseif G_2 then t_2 ... elseif G_n then t_n endif	if G then t_1 else if G_2 then t_2 ... else if G_n then t_n endif ... endif endif
case t_0 of p_1 : t_1 endcase	case t_0 of p_1 : t_1 ; otherwise undef endcase
case t_0 of p_1 : t_1 ; p_2 : t_2 ; ... p_n : t_n endcase	case t_0 of p_1 : t_1 ; otherwise case t_0 of p_2 : t_2 ; otherwise ... case t_0 of p_n : t_n endcase ... endcase endcase
let p == t_1 in t_2 endlet	case t_1 of p : t_2 endcase
[ ]	nil
[t_1 (, t_i)*]	t_1 (:: t_i)* :: nil
{ }	emptyset
{ t_1 (, t_i)* }	list_to_set ([t_1 (, t_i)*])
{ t_{11} -> t_{12} (, t_{i1} -> t_{i2})* }	set_to_map ({ (t_{11}, t_{12}) (, (t_{i1}, t_{i2}))* })
[t_1 .. t_2]	list_interval (t_1 , t_2 , 1)
{ t_1 .. t_2 }	set_interval (t_1 , t_2 , 1)
{ t | v in A with G }	list_to_set ([t | v in set_to_list(A) with G])
{ t_1 -> t_2 | v in A with G }	set_to_map ({ (t_1,t_2) | v in A with G})
(exists v in A : G)	({ v | v in A with G } != emptyset)
(forall v in A : G)	({ v | v in A with G } = A)

C.4 Transition Rules

Basic Forms

rule	→	`skip`	skip rule
	|	*term* `:=` *term*	update rule
	|	*rule rule* (*rule*)*	block rule (implicit)
	|	`block` (*rule*)* `endblock`	block rule (explicit)
	|	`if` *term* `then` *rule* `else` *rule* `endif`	(basic) conditional rule

	|	`case` *term* `of`
		patt `:` *rule* `;`
		`otherwise` *rule*
		`endcase`
			(basic) case rule

	|	`do forall` *p* `in` *A* [`with` *G*] *rule* `enddo`	do-forall rule
	|	`choose` *p* `in` *A* [`with` *G*] *rule* `endchoose`	choose rule
	|	*rule_id* `(` *term* `)`	named rule application

Derived Forms

rule	→	`if` *term* `then` *rule* (`elseif` *term* `then` *rule*)* [`else` *rule*] `endif`	(derived) conditional rule

	|	`case` *term* `of`
		patt `:` *rule*
		(`;` *patt* `:` *rule*)*
		[`;` `otherwise` *rule*]
		`endcase`
			(derived) case rule

	|	`let` *patt* `==` *term* `in` *rule* `endlet`	let rule
	|	*rule_id*	named rule application (nullary)
	|	*rule_id* `(` *term* `,` *term* (`,` *term*)* `)`	named rule appl. (n-ary, $n > 1$)

Derived form	Equivalent basic form
`if` G `then` t `endif`	`if` G `then` t `else skip endif`

`case` t_0 `of`
p_1 `:` t_1
`endcase`
	`case` t_0 `of`
	p_1 `:` t_1 `;`
	`otherwise skip`
	`endcase`

All other derived forms	Like the derived forms for the corresponding terms

C.5 Function Expressions

Basic Forms

fexpr	→	`fn (` *patt* `) ->` *term*	lambda-term
	|	`MAP_TO_FUN` M	finite map to function conversion
	|	`SET_TO_REL` A	finite set to relation conversion

Derived Forms

fexpr	→	`fn ( ) ->` *term*	lambda-term (nullary)
	|	`fn (` *patt* `,` *patt* `(,` *patt*$)^*$ `) ->` *term*	lambda-term (n-ary, $n > 1$)

Derived form	Equivalent basic form
`fn ( ) ->` *term*	`fn ( ( ) ) ->` *term*
`fn (` $patt_1$ `,` $patt_2$ `(,` $patt_i$$)^*$ `) ->` *term*	`fn ( (` $patt_1$ `,` $patt_2$ `(,` $patt_i$$)^*$ `) ) ->` *term*

C.6 Transition Expressions

Basic Forms

texpr	→	`tn (` *patt* `) ->` *rule*	lambda-rule

Derived Forms

texpr	→	`tn ( ) ->` *rule*	lambda-rule (nullary)
	|	`tn (` *patt* `,` *patt* `(,` *patt*$)^*$ `) ->` *rule*	lambda-rule (n-ary, $n > 1$)

Derived form	Equivalent basic form
`tn ( ) ->` *rule*	`tn ( ( ) ) ->` *rule*
`tn (` $patt_1$ `,` $patt_2$ `(,` $patt_i$$)^*$ `) ->` *rule*	`tn ( (` $patt_1$ `,` $patt_2$ `(,` $patt_i$$)^*$ `) ) ->` *rule*

C.7 Definitions

C.7.1 Type Definitions

def	→	`typealias` T [`(` α (`,` α)* `)`] `==` *type*	type alias definition
	|	`freetype` *ft_def*	free type definition
	|	`freetypes` {	simultaneous free type definitions
		ft_def ft_def (*ft_def*)*	
		}	
ft_def	→	T [`(` α (`,` α)* `)`] `==` {	
		fun_id [`:` *type*]	
		(`,` *fun_id* [`:` *type*])*	
		}	

C.7.2 Function Definitions

Basic Forms

def	→	`static function` *fn_eqn*	static function definition
	|	`derived function` *fn_eqn*	derived function definition
	|	`dynamic function` *fun_id* [`:` *func_type*] [`with` *fun_id* [`(` v_1 `,` ... `,` v_n `)`] `in` t] `initially` *fexpr*	dynamic function definition
	|	`external function` *fun_id* `:` *func_type* [`with` *fun_id* [`(` v_1 `,` ... `,` v_n `)`] `in` t]	external function definition
	|	(`static` | `derived`) `functions` {	simultaneous function definitions
		fn_eqn fn_eqn (*fn_eqn*)*	
		}	
fn_eqn	→	*fun_id* [`:` *func_type*] `==` *fexpr*	

Derived Forms

def	→	`dynamic function` *fun_id* [`with` *fun_id* `in` t] `initially` t	nullary dynamic function definition
fn_eqn	→	*fun_id* `==` t	
	|	*fun_id* `(` p (`,` p)* `)` `==` t	

Derived form	Equivalent basic form
`dynamic function` *fun_id* [`with` *fun_id* `in` t] `initially` t_0	`dynamic function` *fun_id* [`with` *fun_id* `in` t] `initially MAP_TO_FUN { ( ) ->` t_0 `}`
fun_id `==` t	*fun_id* `== fn ( ) ->` t
fun_id `(` p_1 (`,` p_i)* `)` `==` t	*fun_id* `== fn (` p_1 (`,` p_i)* `) ->` t

C.7.3 Named Rule Definitions

Basic Forms

def → `transition` *rule_id* [`:` *type*] `==` *texpr*

Derived Forms

def → `transition` *rule_id* `==` R
| `transition` *rule_id* `(` p [`,` p]* `)` `==` R

Derived form	Equivalent basic form
`transition` r `==` R	`transition` r `==` `tn ( ) ->` R
`transition` r `(` p_1 [`,` p_i]* `)` `==` R	`transition` r `==` `tn (` p_1 [`,` p_i]* `) ->` R

Appendix D

Proofs

This appendix contains the proofs of the lemmas that have been omitted in Chapter 7 because of their length. To carry out these proofs an additional lemma is needed, Lemma D.1, which is formulated and proved here as well.

For better readability, this appendix is subdivided into (sub-)sections, according to the purpose of the proved lemmas. In the proofs, for conciseness and to avoid ambiguities, the semantics of terms t is denoted by $S_\rho(t)$ and the semantics of rules R by $\Delta_{S,\rho}(R)$. That is, according to the notation of Chapter 5, $S_\rho(t) = \mathcal{E}[\![\, t \,]\!](S, \rho)$ and $\Delta_{S,\rho}(R) = \Delta[\![\, R \,]\!](S, \rho)$.

D.1 Auxiliary Lemma

Lemma D.1 Let p be a pattern and x a value. If p and x match, then $\mathcal{M}[\![\, p \,]\!](x)$ contains bindings for exactly all variables occurring in p. That is, more concisely:

$$\mathcal{M}[\![\, p \,]\!](x) \neq \text{FAIL} \;\Rightarrow\; \text{dom}\,(\,\mathcal{M}[\![\, p \,]\!](x)\,) = \text{vars}(p)$$

Proof By induction on the structure of the pattern p. In all cases it is assumed that $\mathcal{M}[\![\, p \,]\!](x) \neq \text{FAIL}$.

1. Special Constants: $p \equiv$ *int_const* | *float_const* | *string_const*

   Special case: application of nullary constructor name (see below).

2. Placeholders: $p \equiv \_$

   $$\text{dom}\,(\,\mathcal{M}[\![\, \_ \,]\!](x)\,) = \text{dom}\,(\,\{\,\}\,) = \{\,\} = \text{vars}(\_)$$

3. Variables (Defining Occurrences): $p \equiv v$

   $$\text{dom}\,(\,\mathcal{M}[\![\, v \,]\!](x)\,) = \text{dom}\,(\,\{\, v \mapsto x \,\}\,) = \{\, v \,\} = \text{vars}(v)$$

4. TUPLE PATTERNS: $p \equiv (p_1, \ldots, p_n)$

   From $\mathcal{M}[\![\, p \,]\!](x) \neq \text{FAIL}$ follows that x is a tuple: $x = (x_1, \ldots, x_n)$. Then:

   $$\begin{aligned} & \operatorname{dom}(\, \mathcal{M}[\![\, (p_1, \ldots, p_n) \,]\!](x_1, \ldots, x_n) \,) = && \text{by def. of } \mathcal{M} \\ &= \operatorname{dom}(\, \textstyle\bigcup_{i=1}^{n} \mathcal{M}[\![\, p_i \,]\!](x_i) \,) = && \text{by set/map axioms} \\ &= \textstyle\bigcup_{i=1}^{n} \operatorname{dom}(\, \mathcal{M}[\![\, p_i \,]\!](x_i) \,) = && \text{by inductive hypothesis} \\ &= \textstyle\bigcup_{i=1}^{n} \operatorname{vars}(p_i) = && \text{by def. of ``vars''} \\ &= \operatorname{vars}((p_1, \ldots, p_n)) \end{aligned}$$

5. CONSTRUCTOR APPLICATIONS: $p \equiv f(p')$

   From $\mathcal{M}[\![\, p \,]\!](x) \neq \text{FAIL}$ follows that x is a value of a free type, more precisely: $x = C(x')$ where $C \equiv f$. Then:

   $$\begin{aligned} & \operatorname{dom}(\, \mathcal{M}[\![\, f(p') \,]\!](C(x')) \,) = && \text{by def. of } \mathcal{M} \\ &= \operatorname{dom}(\, \mathcal{M}[\![\, p' \,]\!](x') \,) = && \text{by inductive hypothesis} \\ &= \operatorname{vars}(p') = && \text{by def. of ``vars''} \\ &= \operatorname{vars}(f(p')) \end{aligned}$$

D.2 Correctness

D.2.1 Term Simplification

Lemma D.2 Let t be a (partially evaluated) term of ASM-SL and ρ, ρ' two environments such that there is a binding in ρ or in ρ' for each variable occurring free in t. Then, in every state S, $S_\rho([\![t]\!]_{\rho'}) = S_{\rho \oplus \rho'}(t)$. That is, more concisely:

$$\operatorname{freevars}(t) \subseteq \operatorname{dom} \rho \cup \operatorname{dom} \rho' \;\Rightarrow\; S_\rho([\![t]\!]_{\rho'}) = S_{\rho \oplus \rho'}(t).$$

Proof The proof is done by induction on the structure of the term t. There is a case for each basic form of term. For the sake of conciseness, term interpretation and term simplification are referred to as S and $[\![\cdot]\!]$, respectively.

1. VALUES: $t \equiv x$

   For all environments ρ, $S_\rho(x) = x$ and $[\![x]\!]_\rho = x$ (by def. of S and $[\![\cdot]\!]$). Therefore, for every ρ and ρ':

   $$S_\rho([\![x]\!]_{\rho'}) = S_\rho(x) = x = S_{\rho \oplus \rho'}(x)$$

2. LOCATIONS: $t \equiv l \equiv (f, x)$

   By the same argument:

   $$S_\rho([\![l]\!]_{\rho'}) = S_\rho(l) = l = S_{\rho \oplus \rho'}(l)$$

3. Special Constants: $t \equiv$ *int_const* | *float_const* | *string_const*

   Special case: application of nullary static function name (see below).

4. Variables (Applied Occurrences): $t \equiv v$

   (a) If $v \in \text{dom } \rho'$:

   $$\begin{aligned} & S_\rho([\![v]\!]_{\rho'}) && \\ = \ & S_\rho(\rho'(v)) && \text{by def. of } [\![\cdot]\!] \\ = \ & \rho'(v) && \text{by def. of } S, \text{ as } \rho'(v) \text{ is a value} \\ = \ & (\rho \oplus \rho')(v) && \text{by def. of } \oplus, \text{ as } v \in \text{dom}(\rho') \\ = \ & S_{\rho\oplus\rho'}(v) && \text{by def. of } S \end{aligned}$$

   (b) If $v \notin \text{dom } \rho'$:

   $$\begin{aligned} & S_\rho([\![v]\!]_{\rho'}) && \\ = \ & S_\rho(v) && \text{by def. of } [\![\cdot]\!] \\ = \ & \rho(v) && \text{by def. of } S, \text{ as } \rho'(v) \text{ is a variable} \\ = \ & (\rho \oplus \rho')(v) && \text{by def. of } \oplus, \text{ as } v \notin \text{dom}(\rho') \\ = \ & S_{\rho\oplus\rho'}(v) && \text{by def. of } S \end{aligned}$$

   Note that, here, we make use of the assumption $\text{FV}(t) \subseteq \text{dom } \rho \cup \text{dom } \rho'$, i.e., that there is a binding in ρ or ρ' for each variable occurring free in t. Otherwise $S_{\rho\oplus\rho'}(v)$ and $S_\rho([\![v]\!]_{\rho'})$ may be undefined.

5. Tuple Terms: $t \equiv (t_1, \ldots, t_n)$

   (a) If $[\![t_i]\!]_{\rho'} = x_i$ for all $i \in \{1, \ldots, n\}$ (i.e., each $[\![t_i]\!]_{\rho'}$ is a value):

   $$\begin{aligned} & S_\rho([\![(t_1, \ldots, t_n)]\!]_{\rho'}) && \\ = \ & S_\rho((x_1, \ldots, x_n)) && \text{by def. of } [\![\cdot]\!] \\ = \ & (x_1, \ldots, x_n) && \text{by def. of } S\text{: } (x_1, \ldots, x_n) \text{ is a value} \\ = \ & (S_\rho(x_1), \ldots, S_\rho(x_n)) && \text{by def. of } S\text{: each } x_i \text{ is a value} \\ = \ & (S_\rho([\![t_1]\!]_{\rho'}), \ldots, S_\rho([\![t_n]\!]_{\rho'})) && \text{by hypothesis: case (a)} \\ = \ & (S_{\rho\oplus\rho'}(t_1), \ldots, S_{\rho\oplus\rho'}(t_n)) && \text{by inductive hypothesis} \\ = \ & S_{\rho\oplus\rho'}((t_1, \ldots, t_n)) && \text{by def. of } S \end{aligned}$$

   (b) Otherwise:

   $$\begin{aligned} & S_\rho([\![(t_1, \ldots, t_n)]\!]_{\rho'}) && \\ = \ & S_\rho(([\![t_1]\!]_{\rho'}, \ldots, [\![t_n]\!]_{\rho'})) && \text{by def. of } [\![\cdot]\!] \\ = \ & (S_\rho([\![t_1]\!]_{\rho'}), \ldots, S_\rho([\![t_n]\!]_{\rho'})) && \text{by def. of } S \text{ on tuple terms} \\ = \ & (S_{\rho\oplus\rho'}(t_1), \ldots, S_{\rho\oplus\rho'}(t_n)) && \text{by inductive hypothesis} \\ = \ & S_{\rho\oplus\rho'}((t_1, \ldots, t_n)) && \text{by def. of } S \end{aligned}$$

6. Function Applications: $t \equiv f(t')$

   (a_1) If $[\![t']\!]_{\rho'} = x$ and f is a static function name:

$$\begin{array}{rcll} & S_\rho([\![f(t')]\!]_{\rho'}) & = & \\ = & S_\rho(\mathbf{f}(x)) & = & \text{by def. of } [\![\cdot]\!] \\ = & \mathbf{f}(x) & = & \text{by def. of } S, \text{ as } \mathbf{f}(x) \text{ is a value} \\ = & \mathbf{f}(S_\rho(x)) & = & \text{by def. of } S, \text{ as } x \text{ is a value} \\ = & \mathbf{f}(S_\rho([\![t']\!]_{\rho'})) & = & \text{by hypothesis: case } (a_1) \\ = & \mathbf{f}(S_{\rho\oplus\rho'}(t')) & = & \text{by inductive hypothesis} \\ = & S_{\rho\oplus\rho'}(f(t')) & & \text{by def. of } S \end{array}$$

   (a_2) If $[\![t']\!]_{\rho'} = x$ and f is a dynamic/external function name:

$$\begin{array}{rcll} & S_\rho([\![f(t')]\!]_{\rho'}) & = & \\ = & S_\rho((f,x)) & = & \text{by def. of } [\![\cdot]\!] \\ = & \mathbf{f}_S(x) & = & \text{by def. of } S, \text{ as } (f,x) \text{ is a location} \\ = & \mathbf{f}_S(S_\rho(x)) & = & \text{by def. of } S, \text{ as } x \text{ is a value} \\ = & \mathbf{f}_S(S_\rho([\![t']\!]_{\rho'})) & = & \text{by hypothesis: case } (a_2) \\ = & \mathbf{f}_S(S_{\rho\oplus\rho'}(t')) & = & \text{by inductive hypothesis} \\ = & S_{\rho\oplus\rho'}(f(t')) & & \text{by def. of } S \end{array}$$

   (b) Otherwise:

$$\begin{array}{rcll} & S_\rho([\![f(t')]\!]_{\rho'}) & = & \\ = & S_\rho(f([\![t']\!]_{\rho'})) & = & \text{by def. of } [\![\cdot]\!] \\ = & \mathbf{f}_S(S_\rho([\![t']\!]_{\rho'})) & = & \text{by def. of } S \\ = & \mathbf{f}_S(S_{\rho\oplus\rho'}(t')) & = & \text{by inductive hypothesis} \\ = & S_{\rho\oplus\rho'}(f(t')) & & \text{by def. of } S \end{array}$$

7. Conditional Terms: $t \equiv \texttt{if}\ G\ \texttt{then}\ t_T\ \texttt{else}\ t_F$

   (a_1) If $[\![G]\!]_{\rho'} = \mathbf{true}$:

$$\begin{array}{rcll} & S_\rho([\![\texttt{if}\ G\ \texttt{then}\ t_T\ \texttt{else}\ t_F]\!]_{\rho'}) & = & \\ = & S_\rho([\![t_T]\!]_{\rho'}) & = & \text{by def. of } [\![\cdot]\!] \\ = & S_{\rho\oplus\rho'}(t_T) & = & \text{by inductive hypothesis} \\ = & S_{\rho\oplus\rho'}(\texttt{if}\ \mathit{true}\ \texttt{then}\ t_T\ \texttt{else}\ t_F) & = & \text{subst. by equiv. term} \\ = & S_{\rho\oplus\rho'}(\texttt{if}\ G\ \texttt{then}\ t_T\ \texttt{else}\ t_F) & & \text{subst. by equiv. term} \end{array}$$

   The last equality is justified as follows. It follows from the inductive hypothesis and (a_1) that $S_{\rho\oplus\rho'}(G) = S_\rho([\![G]\!]_{\rho'}) = S_\rho(\mathbf{true}) = \mathbf{true}$. Thus, it admissible to substitute *true* by G in the conditional term.

(a_2) If $[\![G]\!]_{\rho'} = \mathbf{false}$:

Same as (a_1), with t_T replaced by t_F in the second and third lines and *true* replaced by *false* in the fourth line.

(b) Otherwise:

$$\begin{aligned}
& S_\rho([\![\texttt{if}\ G\ \texttt{then}\ t_T\ \texttt{else}\ t_F]\!]_{\rho'}) && = && \text{by def. of } [\![\cdot]\!] \\
&= S_\rho(\texttt{if}\ [\![G]\!]_{\rho'}\ \texttt{then}\ [\![t_T]\!]_{\rho'}\ \texttt{else}\ [\![t_F]\!]_{\rho'}) && = && \text{by def. of } S \\
&= \begin{cases} S_\rho([\![t_T]\!]_{\rho'}) & \text{if } S_\rho([\![G]\!]_{\rho'}) = \mathbf{true} \\ S_\rho([\![t_F]\!]_{\rho'}) & \text{otherwise} \end{cases} && && \text{by ind. hyp.} \\
&= \begin{cases} S_{\rho\oplus\rho'}(t_T) & \text{if } S_{\rho\oplus\rho'}(G) = \mathbf{true} \\ S_{\rho\oplus\rho'}(t_F) & \text{otherwise} \end{cases} && && \text{by def. of } S \\
&= S_{\rho\oplus\rho'}(\texttt{if}\ G\ \texttt{then}\ t_T\ \texttt{else}\ t_F)
\end{aligned}$$

8. Case Terms: $t \equiv \texttt{case}\ t_0\ \texttt{of}\ p : t_1\texttt{;}\ \texttt{otherwise}\ t_2$

(a_1) If $[\![t_0]\!]_{\rho'} = x$ and $\mathcal{M}[\![\, p\,]\!](x) = \rho'' \neq \text{FAIL}$:

$$\begin{aligned}
& S_\rho([\![\texttt{case}\ t_0\ \texttt{of}\ p : t_1\texttt{;}\ \texttt{otherwise}\ t_2]\!]_{\rho'}) && = && \text{by def. of } [\![\cdot]\!] \\
&= S_\rho([\![t_1]\!]_{\rho'\oplus\rho''}) && = && \text{by ind. hyp.} \\
&= S_{\rho\oplus(\rho'\oplus\rho'')}(t_1) && = && \text{associativity of } \oplus \\
&= S_{(\rho\oplus\rho')\oplus\rho''}(t_1) && = && \text{see below} \\
&= S_{(\rho\oplus\rho')\oplus(\mathcal{M}[\![p]\!](S_{\rho\oplus\rho'}(t_0)))}(t_1) && = && \text{by def. of } S \\
&= S_{\rho\oplus\rho'}(\texttt{case}\ t_0\ \texttt{of}\ p : t_1\texttt{;}\ \texttt{otherwise}\ t_2)
\end{aligned}$$

It remains to be proved that $\rho'' = \mathcal{M}[\![\, p\,]\!](S_{\rho\oplus\rho'}(t_0))$. It follows from the inductive hypothesis and assumption (a_1) that

$$S_{\rho\oplus\rho'}(t_0) = S_\rho([\![t_0]\!]_{\rho'}) = S_\rho(x) = x.$$

Therefore, $\mathcal{M}[\![\, p\,]\!](S_{\rho\oplus\rho'}(t_0)) = \mathcal{M}[\![\, p\,]\!](x) = \rho''$.

(a_2) If $[\![t_0]\!]_{\rho'} = x$ and $\mathcal{M}[\![\, p\,]\!](x) = \text{FAIL}$:

$$\begin{aligned}
& S_\rho([\![\texttt{case}\ t_0\ \texttt{of}\ p : t_1\texttt{;}\ \texttt{otherwise}\ t_2]\!]_{\rho'}) && = && \text{by def. of } [\![\cdot]\!] \\
&= S_\rho([\![t_2]\!]_{\rho'}) && = && \text{by ind. hyp.} \\
&= S_{\rho\oplus\rho'}(t_2) && = && \text{by def. of } S \\
&= S_{\rho\oplus\rho'}(\texttt{case}\ t_0\ \texttt{of}\ p : t_1\texttt{;}\ \texttt{otherwise}\ t_2)
\end{aligned}$$

The last equality holds as $\mathcal{M}[\![\, p\,]\!](S_{\rho\oplus\rho'}(t_0)) = \mathcal{M}[\![\, p\,]\!](x) = \text{FAIL}$.

(b) Otherwise ($[\![t_0]\!]_{\rho'}$ is not a value):

By definition of $[\![\cdot]\!]$:

$$\begin{aligned}
& S_\rho([\![\texttt{case}\ t_0\ \texttt{of}\ p : t_1\texttt{;}\ \texttt{otherwise}\ t_2]\!]_{\rho'}) \ = \\
&= S_\rho(\texttt{case}\ [\![t_0]\!]_{\rho'}\ \texttt{of}\ p : [\![t_1]\!]_{\rho'\setminus\text{vars}(p)}\texttt{;}\ \texttt{otherwise}\ [\![t_2]\!]_{\rho'})
\end{aligned}$$

Now, let $\rho'' = \mathcal{M}[\![\, p \,]\!](S_\rho([\![t_0]\!]_{\rho'}))$.

(b_1) If $\rho'' \neq$ FAIL:

By definition of $[\![\cdot]\!]$ and by inductive hypothesis:

$$\begin{aligned} & S_\rho(\texttt{case}\ [\![t_0]\!]_{\rho'}\ \texttt{of}\ p : [\![t_1]\!]_{\rho'\setminus \mathrm{vars}(p)}\texttt{;}\ \texttt{otherwise}\ [\![t_2]\!]_{\rho'}) \ = \\ = {} & S_{\rho\oplus\rho''}([\![t_1]\!]_{\rho'\setminus \mathrm{vars}(p)}) \ = \\ = {} & S_{\rho\oplus\rho''\oplus(\rho'\setminus \mathrm{vars}(p))}(t_1) \end{aligned}$$

By Lemma D.1, dom $\rho'' = \mathrm{vars}(p)$. Thus, it follows from the properties of $\oplus$—mentioned in *Notational Conventions*—that:

$$\begin{aligned} & S_{\rho\oplus\rho''\oplus(\rho'\setminus \mathrm{vars}(p))}(t_1) \ = \\ = {} & S_{\rho\oplus\,(\rho'\setminus \mathrm{vars}(p))\,\oplus\,\rho''}(t_1) = \\ = {} & S_{\rho\oplus\rho'\oplus\rho''}(t_1) \end{aligned} \qquad \begin{array}{l} (\,\mathrm{dom}\,\rho'' \cap \mathrm{dom}\,(\rho'\setminus \mathrm{vars}(p)) = \emptyset\,) \\ (\,\mathrm{vars}(p) \subseteq \mathrm{dom}\ \rho''\,) \end{array}$$

Finally:

$$\begin{aligned} & S_{\rho\oplus\rho'\oplus\rho''}(t_1) && = && \text{by def. of } \rho'' \\ = {} & S_{\rho\oplus\rho'\oplus(\mathcal{M}[\![p]\!](S_\rho([\![t_0]\!]_{\rho'})))}(t_1) && = && \text{by ind. hyp.} \\ = {} & S_{(\rho\oplus\rho')\oplus(\mathcal{M}[\![p]\!](S_{\rho\oplus\rho'}(t_0)))}(t_1) && = && \text{by def. of } S \\ = {} & S_{\rho\oplus\rho'}(\texttt{case}\ t_0\ \texttt{of}\ p : t_1\texttt{;}\ \texttt{otherwise}\ t_2) \end{aligned}$$

(b_2) If $\rho'' =$ FAIL:

By definition of $[\![\cdot]\!]$, inductive hypothesis, and definition of S:

$$\begin{aligned} & S_\rho(\texttt{case}\ [\![t_0]\!]_{\rho'}\ \texttt{of}\ p : [\![t_1]\!]_{\rho'\setminus \mathrm{vars}(p)}\texttt{;}\ \texttt{otherwise}\ [\![t_2]\!]_{\rho'}) \ = \\ = {} & S_\rho([\![t_2]\!]_{\rho'}) \ = \\ = {} & S_{\rho\oplus\rho'}(t_2) \ = \\ = {} & S_{\rho\oplus\rho'}(\texttt{case}\ t_0\ \texttt{of}\ p : t_1\texttt{;}\ \texttt{otherwise}\ t_2) \end{aligned}$$

The last equality holds as, by the inductive hypothesis and (b_2), $\mathcal{M}[\![\, p \,]\!](S_{\rho\oplus\rho'}(t_0)) = \mathcal{M}[\![\, p \,]\!](S_\rho([\![t_0]\!]_{\rho'})) =$ FAIL.

9. Finite Quantifications and Comprehension Terms

With respect to the binding of variables by means of pattern matching, the same reasoning carried out above for `case`-terms applies. See also the proof for the case of `do forall`-rules in Lemma D.4 below, which is essentially the same proof as for quantification and comprehension terms.

Lemma D.3 For every closed term t, in each state S:

$S([\![t]\!]) = S(t)$.

That is, the term-simplifying transformation $[\![\cdot]\!]$ preserves term semantics.

Proof Follows from Lemma D.2, for $\rho = \rho' = \emptyset$, as freevars$(t) = \emptyset$.

D.2.2 Rule Simplification

Lemma D.4 Let R be an ASM-SL rule and ρ, ρ' two environments such that there is a binding in ρ or in ρ' for each variable occurring free in R. Then, in every state S, $\Delta_{S,\rho}([\![R]\!]_{\rho'}) = \Delta_{S,\rho\oplus\rho'}(R)$. That is, more concisely:

$$\text{freevars}(R) \subseteq \text{dom}\ \rho \cup \text{dom}\ \rho' \ \Rightarrow\ \Delta_{S,\rho}([\![R]\!]_{\rho'}) = \Delta_{S,\rho\oplus\rho'}(R).$$

Proof The proof is done by induction on the structure of the rule R. There is a case for each basic form of rule. For the sake of conciseness, rule interpretation and rule simplification are referred to as Δ and $[\![\cdot]\!]$, respectively.

1. Skip Rules: $R \equiv \texttt{skip}$

   By definition of Δ and $[\![\cdot]\!]$, for all environments ρ, $\Delta_{S,\rho}(\texttt{skip}) = \emptyset$ and $[\![\texttt{skip}]\!]_\rho = \texttt{skip}$. Therefore, for every ρ and ρ':

   $$\Delta_{S,\rho}([\![\texttt{skip}]\!]_{\rho'}) = \Delta_{S,\rho}(\texttt{skip}) = \emptyset = \Delta_{S,\rho\oplus\rho'}(\texttt{skip})$$

2. Update Rules: Two cases must be distinguished here. In the first case the left-hand side of R is a location ($R \equiv l := t'$), in the second case it is an application term ($R \equiv f(t) := t'$). The second case has to be split further in two subcases, depending on whether t simplifies to a value (case b_1) or not (case b_2).

   (a) $R \equiv l := t'$:

   $$\begin{aligned}
   & \Delta_{S,\rho}([\![\, l := t' \,]\!]_{\rho'}) && \text{by def. of } [\![\cdot]\!] \text{ on rules}\\
   = {} & \Delta_{S,\rho}([\![\, l \,]\!]_{\rho'} := [\![t']\!]_{\rho'}) && \text{by def. of } [\![\cdot]\!] \text{ on term}\\
   = {} & \Delta_{S,\rho}(\, l := [\![t']\!]_{\rho'}) && \text{by def. of } \Delta\\
   = {} & \{\, (\, l,\, S_\rho([\![t']\!]_{\rho'})\,)\,\} && \text{by Lemma D.2}\\
   = {} & \{\, (\, l,\, S_{\rho\oplus\rho'}(t')\,)\,\} && \text{by def. of } \Delta\\
   = {} & \Delta_{S,\rho\oplus\rho'}(\, l := t' \,)
   \end{aligned}$$

   (b_1) $R \equiv f(t) := t'$ and $[\![t]\!]_{\rho'} = x$:

   $$\begin{aligned}
   & \Delta_{S,\rho}([\![f(t) := t']\!]_{\rho'}) && \text{by def. of } [\![\cdot]\!] \text{ on rules}\\
   = {} & \Delta_{S,\rho}([\![f(t)]\!]_{\rho'} := [\![t']\!]_{\rho'}) && \text{by def. of } [\![\cdot]\!] \text{ on terms}\\
   = {} & \Delta_{S,\rho}(\, (f, x) := [\![t']\!]_{\rho'}\,) && \text{by def. of } \Delta\\
   = {} & \{\, ((f, S_\rho(x)),\, S_\rho([\![t']\!]_{\rho'}))\,\} && \text{by assumption (a)}\\
   = {} & \{\, ((f, S_\rho([\![t]\!]_{\rho'})),\, S_\rho([\![t']\!]_{\rho'}))\,\} && \text{by Lemma D.2}\\
   = {} & \{\, ((f, S_{\rho\oplus\rho'}(t)),\, S_{\rho\oplus\rho'}(t'))\,\} && \text{by def. of } \Delta\\
   = {} & \Delta_{S,\rho\oplus\rho'}(\, f(t) := t' \,)
   \end{aligned}$$

(b$_2$) $R \equiv f(t) := t'$ and $[\![t]\!]_{\rho'}$ is not a value:

$$\begin{array}{lcl}
\Delta_{S,\rho}([\![f(t) := t']\!]_{\rho'}) & = & \\
= \Delta_{S,\rho}([\![f(t)]\!]_{\rho'} := [\![t']\!]_{\rho'}) & = & \text{by def. of } [\![\cdot]\!] \text{ on rules} \\
= \Delta_{S,\rho}(f([\![t]\!]_{\rho'}) := [\![t']\!]_{\rho'}) & = & \text{by def. of } [\![\cdot]\!] \text{ on terms} \\
= \{\, ((f, S_\rho([\![t]\!]_{\rho'})),\, S_\rho([\![t']\!]_{\rho'}))\,\} & = & \text{by def. of } \Delta \\
= \{\, ((f, S_{\rho\oplus\rho'}(t)),\, S_{\rho\oplus\rho'}(t'))\,\} & = & \text{by Lemma D.2} \\
= \Delta_{S,\rho\oplus\rho'}(\, f(t) := t'\,) & & \text{by def. of } \Delta
\end{array}$$

3. Block Rules: $R \equiv R_1 \ldots R_n$

$$\begin{array}{lcl}
\Delta_{S,\rho}([\![R_1 \ldots R_n]\!]_{\rho'}) & = & \\
= \Delta_{S,\rho}([\![R_1]\!]_{\rho'} \ldots [\![R_n]\!]_{\rho'}) & = & \text{by def. of } [\![\cdot]\!] \text{ on rules} \\
= \textstyle\sum_{i=1}^{n} \Delta_{S,\rho}([\![R_i]\!]_{\rho'}) & = & \text{by def. of } \Delta \\
= \textstyle\sum_{i=1}^{n} \Delta_{S,\rho\oplus\rho'}(R_i) & = & \text{by inductive hypothesis} \\
= \Delta_{S,\rho\oplus\rho'}(\, R_1 \ldots R_n\,) & & \text{by def. of } \Delta
\end{array}$$

4. Conditional Rules: $R \equiv \texttt{if}\ G\ \texttt{then}\ R_T\ \texttt{else}\ R_F$

   Same proof as for conditional terms.

5. Case Rules: $R \equiv \texttt{case}\ t_0\ \texttt{of}\ p\ \texttt{:}\ R_1\texttt{;}\ \texttt{otherwise}\ R_2$

   Same proof as for `case`-terms.

6. Do-Forall Rules: $R \equiv \texttt{do forall}\ p\ \texttt{in}\ A\ \texttt{with}\ G\ \ R'$

   To simplify the proof, we first observe that a rule of the form

   ```
   do forall p in A with G
      R'
   ```

   is obviously equivalent to

   ```
   do forall p in A
      if G then R'.
   ```

   Thus, it suffices to show that

   $$\Delta_{S,\rho}([\![\texttt{do forall}\ p\ \texttt{in}\ A\ \ R']\!]_{\rho'}) = \Delta_{S,\rho\oplus\rho'}(\,\texttt{do forall}\ p\ \texttt{in}\ A\ \ R'\,).$$

   Two cases have to be distinguished, depending on whether A simplifies to a value (i.e., $[\![A]\!]_{\rho'} = X$) or not.

   (a) If $[\![A]\!]_{\rho'} = X$ ($[\![A]\!]_{\rho'}$ is a value):

   First we observe that, due to the typing rule for `do-forall`, X is a finite set. Let $X' = \{x_1, \ldots, x_n\}$ be the subset of X defined by

   $$X' = \{\, x \in X \mid \mathcal{M}[\![\, p\,]\!](x) \neq \text{FAIL}\,\}$$

(like in the definition of $[\![\cdot]\!]$). Then the statement is proved by the following chain of equalities:

$$\begin{aligned}
&\Delta_{S,\rho}([\![\texttt{do forall } p \texttt{ in } A \;\; R']\!]_{\rho'}) = && \text{by def. of } [\![\cdot]\!] \text{ on rules}\\
&= \Delta_{S,\rho}\begin{pmatrix} [\![R']\!]_{\rho'\oplus\mathcal{M}[\![p]\!](x_1)} \\ \vdots \\ [\![R']\!]_{\rho'\oplus\mathcal{M}[\![p]\!](x_n)} \end{pmatrix} = && \\
&= \textstyle\sum_{i=1}^{n} \Delta_{S,\rho}([\![R']\!]_{\rho'\oplus\mathcal{M}[\![p]\!](x_i)}) = && \text{by def. of } \Delta\\
&= \textstyle\sum_{x\in X'} \Delta_{S,\rho}([\![R']\!]_{\rho'\oplus\mathcal{M}[\![p]\!](x)}) = && \text{as } X' = \{x_1,\ldots,x_n\}\\
&= \textstyle\sum_{x\in X'} \Delta_{S,\rho\oplus\rho'\oplus\mathcal{M}[\![p]\!](x)}(R') = && \text{by inductive hypothesis}\\
&= \Delta_{S,\rho\oplus\rho'}(\texttt{do forall } p \texttt{ in } A \;\; R') && \text{by def. of } \Delta
\end{aligned}$$

(b) Otherwise ($[\![A]\!]_{\rho'}$ is not a value):

$$\begin{aligned}
&\Delta_{S,\rho}([\![\texttt{do forall } p \texttt{ in } A \;\; R']\!]_{\rho'}) = && \text{by def. of } [\![\cdot]\!]\\
&= \Delta_{S,\rho}(\texttt{do forall } p \texttt{ in } [\![A]\!]_{\rho'} \;\; [\![R']\!]_{\rho'\setminus\mathrm{vars}(p)}) = && \text{by def. of } \Delta\\
&= \textstyle\sum_{x\in X'} \Delta_{S,\rho\oplus\mathcal{M}[\![p]\!](x)}([\![R']\!]_{\rho'\setminus\mathrm{vars}(p)}) &&
\end{aligned}$$

where $X' = \mathrm{range}_{S,\rho}(p, [\![A]\!]_{\rho'}) =$
$= \{\, x \in S_\rho([\![A]\!]_{\rho'}) \mid \mathcal{M}[\![\, p \,]\!](x) \neq \mathrm{FAIL} \,\} =$
$= \{\, x \in S_{\rho\oplus\rho'}(A) \mid \mathcal{M}[\![\, p \,]\!](x) \neq \mathrm{FAIL} \,\} =$
$= \mathrm{range}_{S,\rho\oplus\rho'}(p, A).$

As shown in the proof of Lemma D.2 (case 8., `case`-terms, b_1):

$$\rho \oplus \mathcal{M}[\![\, p \,]\!](x) \oplus (\rho' \setminus \mathrm{vars}(p)) \;=\; \rho \oplus \rho' \oplus \mathcal{M}[\![\, p \,]\!](x).$$

Thus, the following equalities hold:

$$\begin{aligned}
&\textstyle\sum_{x\in X'} \Delta_{S,\rho\oplus\mathcal{M}[\![p]\!](x)}([\![R']\!]_{\rho'\setminus\mathrm{vars}(p)}) = && \text{by ind. hyp.}\\
&= \textstyle\sum_{x\in X'} \Delta_{S,\rho\oplus\mathcal{M}[\![p]\!](x)\oplus(\rho'\setminus\mathrm{vars}(p))}(R') = && \text{see above}\\
&= \textstyle\sum_{x\in X'} \Delta_{S,\rho\oplus\rho'\oplus\mathcal{M}[\![p]\!](x)}(R') = && \text{by def. of } \Delta\\
&= \Delta_{S,\rho\oplus\rho'}(\texttt{do forall } p \texttt{ in } A \;\; R') &&
\end{aligned}$$

This concludes the proof.

Lemma D.5 For every closed rule R, in each state S:

$$\Delta_S([\![R]\!]) = \Delta_S(R).$$

That is, the rule-simplifying transformation $[\![\cdot]\!]$ preserves rule semantics.

Proof Follows from Lemma D.4, for $\rho = \rho' = \emptyset$, as $\mathrm{freevars}(R) = \emptyset$.

D.2.3 Rule Unfolding

Lemma D.6 Let R be a closed ASM-SL rule. If $\mathcal{E}(R)$ is defined (i.e., if $\mathcal{E}$ applied to R terminates), then:

$$\Delta_S(\mathcal{E}(R)) = \Delta_S(R).$$

in every state S. That is, the unfolding trasformation $\mathcal{E}$ preserves the semantics of rules.

Proof Assume that there exists a (possibly partial) mapping $\overline{\mathcal{E}}$ from update sets to update sets such that $\overline{\mathcal{E}}(\Delta_S(R)) = \Delta_S(\mathcal{E}(R))$, i.e., such that the following diagram commutes:

$$\begin{array}{ccc} \textsc{Rule} & \xrightarrow{\mathcal{E}} & \textsc{Rule} \\ \big\downarrow \Delta_S & & \big\downarrow \Delta_S \\ \mathit{UPDATE\_SET} & \xrightarrow{\overline{\mathcal{E}}} & \mathit{UPDATE\_SET} \end{array}$$

Note that, if such a mapping exists, it is clearly unique.

From the definition of the unfolding transformation $\mathcal{E}$ and from the semantics of rules Δ, we obtain the following equations:

1. If R contains no locations, except as left-hand side of update rules:
$$\overline{\mathcal{E}}(\Delta_S(R)) \;=\; \Delta_S(\mathcal{E}(R)) \;=\; \Delta_S(R).$$

2. Otherwise:
$$\begin{aligned} \overline{\mathcal{E}}(\Delta_S(R)) &= \Delta_S(\mathcal{E}(R)) \\ &= \Delta_S \begin{pmatrix} \texttt{if } l = x_1 \texttt{ then } \mathcal{E}([\![R[x_1/l]]\!]) \\ \texttt{else if } l = x_2 \texttt{ then } \mathcal{E}([\![R[x_2/l]]\!]) \\ \ldots \\ \texttt{else if } l = x_n \texttt{ then } \mathcal{E}([\![R[x_n/l]]\!]) \end{pmatrix} \\ &= \begin{cases} \Delta_S(\mathcal{E}([\![R[x_1/l]]\!])) & \text{if } S(l) = x_1 \\ \Delta_S(\mathcal{E}([\![R[x_2/l]]\!])) & \text{if } S(l) = x_2 \\ \vdots & \\ \Delta_S(\mathcal{E}([\![R[x_n/l]]\!])) & \text{if } S(l) = x_n \end{cases} \end{aligned}$$

   where l is the first location occurring in R (but not as left-hand side of an update rule) and $\{x_1, \ldots, x_n\} = \operatorname{ran} l$ is the range of l.

   Then, for each $x_i \in \operatorname{ran} l$, if $S(l) = x_i$, the equations are:

$$\begin{array}{lll} & \overline{\mathcal{E}}(\Delta_S(R)) = & \\ = & \Delta_S(\mathcal{E}([\![R[x_i/l]]\!])) & = \quad \text{see above} \\ = & \overline{\mathcal{E}}(\Delta_S([\![R[x_i/l]]\!])) & = \quad \text{by hypothesis} \\ = & \overline{\mathcal{E}}(\Delta_S([\![R]\!])) & \quad \text{as } S(l) = x_i \end{array}$$

By choosing $\overline{\mathcal{E}} = id$ (the identity function), all the equations are satisfied. In fact, the equations in (2.) are satisfied by any $\overline{\mathcal{E}}$, as $\Delta_S([\![R]\!]) = \Delta_S(R)$ by Lemma D.5, while the equation in (1.) are satisfied iff $\overline{\mathcal{E}} = id$. Then, we can conclude that, in every state S and for all rules R, $\Delta_S(\mathcal{E}(R)) = \Delta_S(R)$.

D.3 Termination

Definition D.1 (Measure) The *measure* of a (partially evaluated) term t or rule R is a triple $M = (b, s, n)$, where:

- $b = b(M) \in B$ is a measure of the number and of the nesting depth of variable-binding constructs, where
$$B = \{\, (x_{m-1}, \ldots, x_0) \in \mathbf{N}^* \mid x_{m-1} > 0 \,\};$$
- $s = s(M) \in \mathbf{N}$ is the size of the term or rule, not including values and locations;
- $n = n(M) \in \mathbf{N}$ the number of occurrences of locations.

An addition operation and a total order are defined on B as follows:

$$\begin{array}{l}(x_{m-1}, \ldots, x_0) + (y_{p-1}, \ldots, y_0) \;= \\ \quad = \;(x_{\max(m,p)-1},\, \ldots,\, x_k,\, x_{k-1} + y_{k-1},\, \ldots,\, x_0 + y_0) \\ \text{where } k = \min(m, p);\end{array}$$

$$\begin{array}{l}(x_{m-1}, \ldots, x_0) < (y_{p-1}, \ldots, y_0) \;\Leftrightarrow \\ \quad \Leftrightarrow\; m < p \;\vee \\ \qquad (\, m = p \;\wedge\; \exists k,\, 0 \le k < m,\, (\forall i : k < i < m \Rightarrow x_i = y_i) \;\wedge\; x_k < y_k \,).\end{array}$$

Moreover, a *length* function $|\cdot|$ and a *concatenation* $\circ$ are defined on B:

$$\begin{array}{l}|\, (x_{m-1}, \ldots, x_0) \,| \;=\; m, \\ (x_{m-1}, \ldots, x_0) \circ (y_{p-1}, \ldots, y_0) = (x_{m-1}, \ldots, x_0, y_{p-1}, \ldots, y_0).\end{array}$$

The symbol ϵ denotes the empty sequence, i.e., $\epsilon \in \mathbf{N}^0$.

An addition operation and a total (lexicographic) order are defined on measures as follows:

$$(b, s, n) + (b', s', n') \;=\; (b + b',\, s + s',\, n + n')$$

$$\begin{array}{lll}(b, s, n) < (b', s', n') & \Leftrightarrow & (b < b') \;\vee \\ & & (b = b' \;\wedge\; s < s') \;\vee \\ & & (b = b' \;\wedge\; s = s' \;\wedge\; n < n').\end{array}$$

It is also convenient to introduce an auxiliary operation $\bullet$, defined as follows, for a measure (b, s, n) and $b' \in B$:

$$b' \bullet (b, s, n) \;=\; (b' \circ b, s, n).$$

Note that, given the above definitions, the following properties—which are easy

to prove and will be exploited in the proofs of Lemmas D.7 and D.9 below—hold for measures M, M', M_1, M_1', M_2, M_2' and $b, b', b_1, b_2 \in B$:

$$\begin{array}{rcl}
 & & M + (\epsilon, 0, 0) = M \\
M' > (\epsilon, 0, 0) & \Rightarrow & M + M' > M \\
M_1 \leq M_1' \wedge M_2 \leq M_2' & \Rightarrow & M_1 + M_2 \leq M_1' + M_2' \\
M_1 \leq M_1' \wedge M_2 < M_2' & \Rightarrow & M_1 + M_2 < M_1' + M_2' \\
 & & \epsilon \bullet M_1 = M_1 \\
b > \epsilon & \Rightarrow & b \bullet M_1 > M_1 \\
b \leq b' \wedge M \leq M' & \Rightarrow & b \bullet M \leq b' \bullet M' \\
b < b' \wedge M \leq M' & \Rightarrow & b \bullet M < b' \bullet M' \\
b \leq b' \wedge M < M' & \Rightarrow & b \bullet M < b' \bullet M' \\
 & & \mid b_1 + b_2 \mid = \max(\mid b_1 \mid, \mid b_2 \mid) \\
\mid b_1 \mid = \mid b_2 \mid & \Rightarrow & \mid b_1 + b_2 \mid = \mid b_1 \mid = \mid b_2 \mid \\
\mid b_1 \mid < \mid b_2 \mid & \Rightarrow & b_1 < b_2.
\end{array}$$

Definition D.2 (Measure Function) The measure of a term t or rule R is given by a *measure function* $\mathbf{M}$ defined as follows.

On terms:

$$\begin{array}{ll}
\mathbf{M}(\, x \,) & = (\epsilon, 0, 0) \\
\mathbf{M}(\, l \,) & = (\epsilon, 0, 1) \\
\mathbf{M}(\, v \,) & = (\epsilon, 1, 0) \\
\mathbf{M}(\, (t_1, \ldots, t_n) \,) & = (\epsilon, 1, 0) + \sum_{i=1}^{n} \mathbf{M}(\, t_i \,) \\
\mathbf{M}(\, f(t) \,) & = (\epsilon, 1, 0) + \mathbf{M}(\, t \,)
\end{array}$$

$$\mathbf{M}(\texttt{if}\ G\ \texttt{then}\ t_T\ \texttt{else}\ t_F) \ = \ (\epsilon, 1, 0) + \mathbf{M}(G) + \mathbf{M}(t_T) + \mathbf{M}(t_F)$$

$$\begin{array}{l}
\mathbf{M}(\texttt{case}\ t_0\ \texttt{of}\ p\ \texttt{:}\ t_1\texttt{;}\ \texttt{otherwise}\ t_2) \ = \\
\quad = \ (\epsilon, 1, 0) + \mathbf{M}(t_0) + (1) \bullet \mathbf{M}(t_1) + \mathbf{M}(t_2)
\end{array}$$

On rules:

$$\begin{array}{ll}
\mathbf{M}(\, \texttt{skip} \,) & = (\epsilon, 1, 0) \\
\mathbf{M}(\, l := t' \,) & = \mathbf{M}(\, t' \,) \\
\mathbf{M}(\, f(t) := t' \,) & = (\epsilon, 1, 0) + \mathbf{M}(\, f(t) \,) + \mathbf{M}(\, t' \,) \\
\mathbf{M}(\, R_1 \ldots R_n \,) & = \sum_{i=1}^{n} \mathbf{M}(\, R_i \,)
\end{array}$$

$$\mathbf{M}(\texttt{if}\ G\ \texttt{then}\ R_T\ \texttt{else}\ R_F) \ = \ (\epsilon, 1, 0) + \mathbf{M}(G) + \mathbf{M}(R_T) + \mathbf{M}(R_F)$$

$$\begin{array}{l}
\mathbf{M}(\texttt{case}\ t_0\ \texttt{of}\ p\ \texttt{:}\ R_1\texttt{;}\ \texttt{otherwise}\ R_2) \ = \\
\quad = \ (\epsilon, 1, 0) + \mathbf{M}(t_0) + (1) \bullet \mathbf{M}(R_1) + \mathbf{M}(R_2)
\end{array}$$

$$\mathbf{M}(\texttt{do forall}\ p\ \texttt{in}\ A \ \ R') \ = \ (\epsilon, 1, 0) + \mathbf{M}(A) + (1) \bullet \mathbf{M}(R')$$

Note that update rules with a location on the left-hand side are treated specially. In fact, for the purpose of the proof, it is convenient to define the measure function on rules in such a way that update rules of the form $l := x$, as well as blocks consisting only of update rules of that form, have size 0.

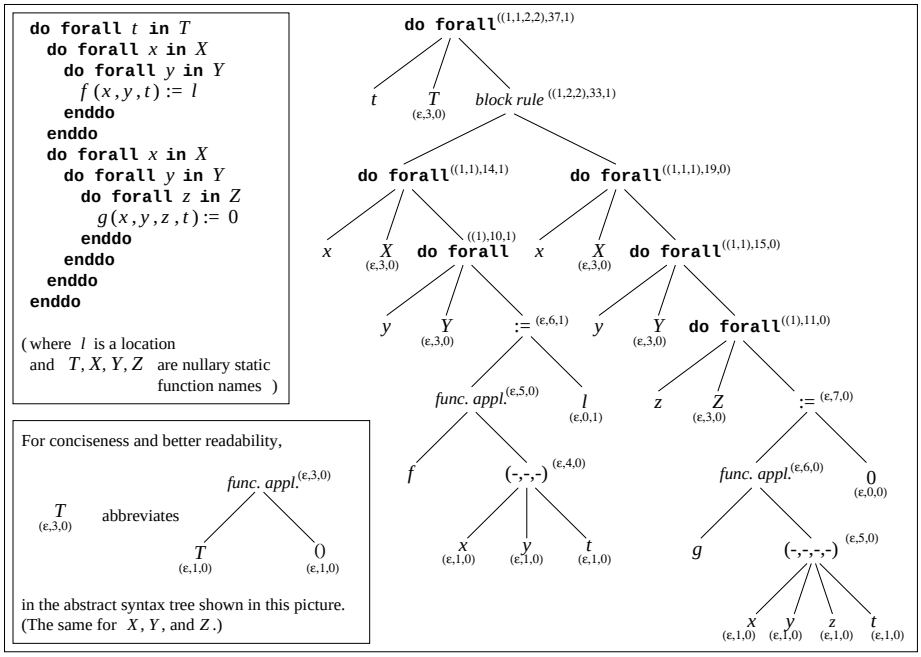


Figure D.1: Measure Function: Example

In this definition and in the following proofs, we only consider `do forall` rules without guard, as `do forall` rules with guard can always be reduced to the form without guard, as discussed in the proof of Lemma D.4, case 6.

Example Figure D.1 shows an example ASM program, where each node of the syntax tree has been annotated with the corresponding measure, computed according to the above definitions.

Observation The following facts follow from the definition of the measure function $\mathbf{M}$ given above. For terms:

- $\mathbf{M}(t) \geq (\epsilon, 0, 0)$ for any term t;
- $\mathbf{M}(t) = (\epsilon, 0, 0)$ iff t is a value;
- $\mathbf{M}(t) = (\epsilon, 0, 1)$ iff t is a location.

For rules:

- $\mathbf{M}(R) \geq (\epsilon, 0, 0)$ for any rule R;
- $\mathbf{M}(R) = (\epsilon, 0, 0)$ iff $R \equiv l_1 := x_1 \ \ldots \ l_n := x_n$,
  i.e., iff R is a block rule consisting only of update rules of the form *location* := *value* ("elementary update block").

D.3.1 Term Unfolding

Lemma D.7 Let t be a (partially evaluated) term of ASM-SL, ρ an environment, x a value, and l a location. Let $t_x = [\![t[x/l]]\!]_\rho$. Then:

(L_1) $\mathbf{M}(\, t_x \,) \leq \mathbf{M}(\, t \,)$.

(L_2) If: *(i)* l occurs in t, or *(ii)* t contains free variables and ρ contains bindings for all free variables in t, i.e., $\emptyset \neq \text{freevars}(t) \subseteq \text{dom } \rho$; then: $\mathbf{M}(\, t_x \,) < \mathbf{M}(\, t \,)$.

Proof As for Lemma D.2, the proof is done by induction on the structure of the term t. There is a case for each basic form of term.

1. Values: $t \equiv x'$

   $t_x = [\![x'[x/l]]\!]_\rho = [\![x']\!]_\rho = x'$.

   Thus, $\mathbf{M}(\, t_x \,) = \mathbf{M}(\, x' \,) = (\epsilon, 0, 0) \leq (\epsilon, 0, 0) = \mathbf{M}(\, x' \,) = \mathbf{M}(\, t \,)$.

   (L_2) holds trivially, as l does not occur in t and $\text{freevars}(t) = \emptyset$.

2. Locations: $t \equiv l' \equiv (f, x')$

   (a) If $l' \neq l$, then:

   $t_x = [\![l'[x/l]]\!]_\rho = [\![l']\!]_\rho = l'$.

   Thus, $\mathbf{M}(\, t_x \,) = \mathbf{M}(\, l' \,) = (\epsilon, 0, 1) \leq (\epsilon, 0, 1) = \mathbf{M}(\, l' \,) = \mathbf{M}(\, t \,)$.

   (L_2) holds, as l does not occur in t and $\text{freevars}(t) = \emptyset$.

   (b) If $l' = l$, then:

   $t_x = [\![l'[x/l]]\!]_\rho = [\![l[x/l]]\!]_\rho = [\![x]\!]_\rho = x$.

   Thus, $\mathbf{M}(\, t_x \,) = \mathbf{M}(\, x \,) = (\epsilon, 0, 0) < (\epsilon, 0, 1) = \mathbf{M}(\, l' \,) = \mathbf{M}(\, t \,)$.

   (L_2) holds, as $\mathbf{M}(\, t_x \,) < \mathbf{M}(\, t \,)$.

3. Special Constants: $t \equiv$ *int_const* | *float_const* | *string_const*

   Special case: application of nullary static function name (see below).

4. Variables (Applied Occurrences): $t \equiv v$

   (a) If $v \in \text{dom } \rho$:

   $t_x = [\![v[x/l]]\!]_\rho = [\![v]\!]_\rho = \rho(v)$.

   Thus, as $\rho(v)$ is a value:

   $\mathbf{M}(\, t_x \,) = \mathbf{M}(\, \rho(v) \,) = (\epsilon, 0, 0) < (\epsilon, 1, 0) = \mathbf{M}(\, v \,) = \mathbf{M}(\, t \,)$

   (L_2) holds, as $\mathbf{M}(\, t_x \,) < \mathbf{M}(\, t \,)$.

(b) If $v \notin \text{dom } \rho$:

$t_x = [\![v[x/l]]\!]_\rho = [\![v]\!]_\rho = v.$

Thus, $\mathbf{M}(\, t_x \,) = \mathbf{M}(\, v \,) = (\epsilon, 1, 0) \leq (\epsilon, 1, 0) = \mathbf{M}(\, v \,) = \mathbf{M}(\, t \,)$.

(L_2) holds, as l does not occur in t and freevars$(t) = \{v\} \not\subseteq \text{dom } \rho$, as $v \notin \text{dom } \rho$.

5. Tuple Terms: $t \equiv (t_1, \ldots, t_n)$

$t_x = [\![(t_1, \ldots, t_n)[x/l]]\!]_\rho = [\![(t_1[x/l], \ldots, t_n[x/l])]\!]_\rho.$

(a) If $[\![t_i[x/l]]\!]_\rho = x_i$ for all $i \in \{1, \ldots, n\}$ (i.e., each $[\![t_i[x/l]]\!]_\rho$ is a value):

$$\begin{aligned}
\mathbf{M}(\, t_x \,) &= \mathbf{M}(\, [\![(t_1[x/l], \ldots, t_n[x/l])]\!]_\rho \,) \\
&= \mathbf{M}(\, (x_1, \ldots, x_n) \,) \\
&= (\epsilon, 0, 0) \\
&< (\epsilon, 1, 0) + \textstyle\sum_{i=1}^{n} \mathbf{M}(\, t_i \,) \\
&= \mathbf{M}(\, (t_1, \ldots, t_n \,)) \\
&= \mathbf{M}(\, t \,)
\end{aligned}$$

(L_2) holds, as $\mathbf{M}(\, t_x \,) < \mathbf{M}(\, t \,)$.

(b) Otherwise:

$$\begin{aligned}
\mathbf{M}(\, t_x \,) &= \mathbf{M}(\, [\![(t_1[x/l], \ldots, t_n[x/l])]\!]_\rho \,) \\
&= \mathbf{M}(\, ([\![t_1[x/l]]\!]_\rho, \ldots, [\![t_n[x/l]]\!]_\rho) \,) \\
&= (\epsilon, 1, 0) + \textstyle\sum_{i=1}^{n} \mathbf{M}(\, [\![t_i[x/l]]\!]_\rho \,)
\end{aligned}$$

By inductive hypothesis, $\mathbf{M}([\![t_i[x/l]]\!]_\rho) \leq \mathbf{M}(t_i)$ for all $i \in \{1, \ldots, n\}$. Therefore:

$$\begin{aligned}
\mathbf{M}(\, t_x \,) &= (\epsilon, 1, 0) + \textstyle\sum_{i=1}^{n} \mathbf{M}(\, [\![t_i[x/l]]\!]_\rho \,) \\
&\leq (\epsilon, 1, 0) + \textstyle\sum_{i=1}^{n} \mathbf{M}(\, t_i \,) \\
&= \mathbf{M}(\, (t_1, \ldots, t_n) \,) \\
&= \mathbf{M}(\, t \,).
\end{aligned}$$

(L_2) holds for the following reason.

In the case that l occurs in $t \equiv (t_1, \ldots, t_n)$, l must occur in at least one of its subterms t_k. Then, by inductive hypothesis, $\mathbf{M}(\, [\![t_k[x/l]]\!]_\rho \,) < \mathbf{M}(\, t_k \,)$. As a consequence, $\mathbf{M}(\, t_x \,) < \mathbf{M}(\, t \,)$.

Similarly, in the case that $\emptyset \neq \text{freevars}(t) \subseteq \text{dom } \rho$, there must at least a subterm t_k with freevars$(t_k) \neq \emptyset$, as a variable occurring free in $(t_1, \ldots, t_n)$ must occur free in at least one of its subterms. Moreover, freevars$(t_k) \subseteq$ freevars$(t) \subseteq$ dom ρ. Thus, by inductive hypothesis, $\mathbf{M}(\, [\![t_k[x/l]]\!]_\rho \,) < \mathbf{M}(\, t_k \,)$. As a consequence, $\mathbf{M}(\, t_x \,) < \mathbf{M}(\, t \,)$.

6. Function Applications: $t \equiv f(t')$

$t_x = [\![(f(t'))[x/l]]\!]_\rho = [\![f(t'[x/l])]\!]_\rho.$

(a_1) If $[\![t'[x/l]]\!]_\rho = x'$ and f is a static function name:

$$\begin{aligned}\mathbf{M}(\,t_x\,) &= \mathbf{M}(\,[\![f(t'[x/l])]\!]_\rho\,)\\ &= \mathbf{M}(\,\mathbf{f}(x')\,)\\ &= (\epsilon,0,0)\\ &< (\epsilon,1,0)+\mathbf{M}(\,t'\,)\\ &= \mathbf{M}(\,f(t')\,)\\ &= \mathbf{M}(\,t\,)\end{aligned}$$

(L_2) holds, as $\mathbf{M}(\,t_x\,) < \mathbf{M}(\,t\,)$.

(a_2) If $[\![t'[x/l]]\!]_\rho = x'$ and f is a dynamic/external function name:

$$\begin{aligned}\mathbf{M}(\,t_x\,) &= \mathbf{M}(\,[\![f(t'[x/l])]\!]_\rho\,)\\ &= \mathbf{M}(\,(f,x')\,)\\ &= (\epsilon,0,1)\\ &< (\epsilon,1,0)+\mathbf{M}(\,t'\,)\\ &= \mathbf{M}(\,f(t')\,)\\ &= \mathbf{M}(\,t\,)\end{aligned}$$

(L_2) holds, as $\mathbf{M}(\,t_x\,) < \mathbf{M}(\,t\,)$.

(b) Otherwise:

$$\begin{aligned}\mathbf{M}(\,t_x\,) &= \mathbf{M}(\,[\![f(t'[x/l])]\!]_\rho\,)\\ &= \mathbf{M}(\,f([\![t'[x/l]]\!]_\rho)\,)\\ &= (\epsilon,1,0)+\mathbf{M}(\,[\![t'[x/l]]\!]_\rho\,)\end{aligned}$$

By inductive hypothesis, $\mathbf{M}([\![t'[x/l]]\!]_\rho) \le \mathbf{M}(t')$. Therefore:

$$\begin{aligned}\mathbf{M}(\,t_x\,) &= (\epsilon,1,0)+\mathbf{M}(\,[\![t'[x/l]]\!]_\rho\,)\\ &\le (\epsilon,1,0)+\mathbf{M}(\,t'\,)\\ &= \mathbf{M}(\,f(t')\,)\\ &= \mathbf{M}(\,t\,).\end{aligned}$$

In order to show that (L_2) holds, the same reasoning as in Tuple Terms, case (b), applies (here, t has exactly one subterm t').

7. Conditional Terms: $t \equiv \texttt{if}\ G\ \texttt{then}\ t_T\ \texttt{else}\ t_F$

$$\begin{aligned}t_x &= [\![(\texttt{if}\ G\ \texttt{then}\ t_T\ \texttt{else}\ t_F)[x/l]]\!]_\rho\\ &= [\![\texttt{if}\ G[x/l]\ \texttt{then}\ t_T[x/l]\ \texttt{else}\ t_F[x/l]]\!]_\rho.\end{aligned}$$

(a_1) If $[\![G[x/l]]\!]_\rho = \mathbf{true}$:

$$\begin{aligned}\mathbf{M}(\,t_x\,) &= \mathbf{M}(\,[\![\texttt{if}\ G[x/l]\ \texttt{then}\ t_T[x/l]\ \texttt{else}\ t_F[x/l]]\!]_\rho\,)\\ &= \mathbf{M}(\,[\![t_T[x/l]]\!]_\rho\,)\\ &\le \mathbf{M}(\,t_T\,)\\ &< (\epsilon,1,0)+\mathbf{M}(\,G\,)+\mathbf{M}(\,t_T\,)+\mathbf{M}(\,t_F\,)\\ &= \mathbf{M}(\,\texttt{if}\ G\ \texttt{then}\ t_T\ \texttt{else}\ t_F\,)\\ &= \mathbf{M}(\,t\,)\end{aligned}$$

(L_2) holds, as $\mathbf{M}(\,t_x\,) < \mathbf{M}(\,t\,)$.

(a$_2$) If $[\![G[x/l]]\!]_\rho = \mathbf{false}$:

Similar to case (a$_1$).

(b) Otherwise ($[\![G[x/l]]\!]_\rho$ is not a value):

$$\begin{aligned}
\mathbf{M}(\, t_x \,) &= \mathbf{M}(\, [\![\texttt{if } G[x/l] \texttt{ then } t_T[x/l] \texttt{ else } t_F[x/l]]\!]_\rho \,) \\
&= \mathbf{M}(\, \texttt{if } [\![G[x/l]]\!]_\rho \texttt{ then } [\![t_T[x/l]]\!]_\rho \texttt{ else } [\![t_F[x/l]]\!]_\rho \,) \\
&= (\epsilon, 1, 0) + \mathbf{M}(\, [\![G[x/l]]\!]_\rho \,) + \\
&\quad + \mathbf{M}(\, [\![t_T[x/l]]\!]_\rho \,) + \mathbf{M}(\, [\![t_F[x/l]]\!]_\rho \,) \\
&\leq (\epsilon, 1, 0) + \mathbf{M}(\, G \,) + \mathbf{M}(\, t_T \,) + \mathbf{M}(\, t_F \,) \\
&= \mathbf{M}(\, \texttt{if } G \texttt{ then } t_T \texttt{ else } t_F \,) \\
&= \mathbf{M}(\, t \,)
\end{aligned}$$

In order to show that (L_2) holds, the same reasoning as in Tuple Terms, case (b), and Application Terms, case (b), applies (subterms of t are here G, t_T, and t_F).

8. Case Terms: $t \equiv \texttt{case } t_0 \texttt{ of } p \texttt{ : } t_1\texttt{; otherwise } t_2$

$$\begin{aligned}
t_x &= [\![(\texttt{case } t_0 \texttt{ of } p \texttt{ : } t_1\texttt{; otherwise } t_2)[x/l]]\!]_\rho \\
&= [\![\texttt{case } t_0[x/l] \texttt{ of } p \texttt{ : } t_1[x/l]\texttt{; otherwise } t_2[x/l]]\!]_\rho.
\end{aligned}$$

(a$_1$) If $[\![t_0]\!]_\rho = x'$ and $\mathcal{M}[\![\, p \,]\!](x') = \rho' \neq \text{FAIL}$:

$$\begin{aligned}
\mathbf{M}(\, t_x \,) &= \mathbf{M}([\![\texttt{case } t_0[x/l] \texttt{ of } p \texttt{ : } t_1[x/l]\texttt{; otherwise } t_2[x/l]]\!]_\rho) \\
&= \mathbf{M}(\, [\![t_1[x/l]]\!]_{\rho \oplus \rho'} \,) \\
&\leq \mathbf{M}(\, t_1 \,) \\
&< (1) \bullet \mathbf{M}(\, t_1 \,) \\
&< (\epsilon, 1, 0) + \mathbf{M}(t_0) + (1) \bullet \mathbf{M}(t_1) + \mathbf{M}(t_2) \\
&= \mathbf{M}(\, \texttt{case } t_0 \texttt{ of } p \texttt{ : } t_1 \texttt{ ; otherwise } t_2 \,) \\
&= \mathbf{M}(\, t \,)
\end{aligned}$$

(L_2) holds, as $\mathbf{M}(\, t_x \,) < \mathbf{M}(\, t \,)$.

(a$_2$) If $[\![t_0]\!]_\rho = x'$ and $\mathcal{M}[\![\, p \,]\!](x') = \text{FAIL}$:

$$\begin{aligned}
\mathbf{M}(\, t_x \,) &= \mathbf{M}([\![\texttt{case } t_0[x/l] \texttt{ of } p \texttt{ : } t_1[x/l]\texttt{; otherwise } t_2[x/l]]\!]_\rho) \\
&= \mathbf{M}(\, [\![t_2[x/l]]\!]_\rho \,) \\
&\leq \mathbf{M}(\, t_2 \,) \\
&< (\epsilon, 1, 0) + \mathbf{M}(t_0) + (1) \bullet \mathbf{M}(t_1) + \mathbf{M}(t_2) \\
&= \mathbf{M}(\, \texttt{case } t_0 \texttt{ of } p \texttt{ : } t_1 \texttt{ ; otherwise } t_2 \,) \\
&= \mathbf{M}(\, t \,)
\end{aligned}$$

(L_2) holds, as $\mathbf{M}(\, t_x \,) < \mathbf{M}(\, t \,)$.

(b) Otherwise ($[\![t_0]\!]_\rho$ is not a value):

$$\begin{aligned}
&\mathbf{M}(\, t_x \,) = \mathbf{M}(\llbracket \texttt{case}\ t_0[x/l]\ \texttt{of}\ p\ \texttt{:}\ t_1[x/l]\texttt{;}\ \texttt{otherwise}\ t_2[x/l] \rrbracket_\rho) \\
&= \mathbf{M}(\texttt{case}\ \llbracket t_0[x/l] \rrbracket_\rho\ \texttt{of}\ p\ \texttt{:}\ \llbracket t_1[x/l] \rrbracket_{\rho \setminus \mathrm{vars}(p)}\texttt{;}\ \texttt{otherwise}\ \llbracket t_2[x/l] \rrbracket_\rho) \\
&= (\epsilon, 1, 0) + \mathbf{M}(\, \llbracket t_0[x/l] \rrbracket_\rho \,) + \\
&\quad + (1) \bullet \mathbf{M}(\, \llbracket t_1[x/l] \rrbracket_{\rho \setminus \mathrm{vars}(p)} \,) + \mathbf{M}(\, \llbracket t_2[x/l] \rrbracket_\rho \,) \\
&\leq (\epsilon, 1, 0) + \mathbf{M}(\, t_0 \,) + (1) \bullet \mathbf{M}(\, t_1 \,) + \mathbf{M}(\, t_2 \,) \\
&= \mathbf{M}(\, \texttt{case}\ t_0\ \texttt{of}\ p\ \texttt{:}\ t_1\texttt{;}\ \texttt{otherwise}\ t_2 \,) \\
&= \mathbf{M}(\, t \,)
\end{aligned}$$

(L_2) holds for the following reason.

In the case that l occurs in $t \equiv \texttt{case}\ t_0\ \texttt{of}\ p\ \texttt{:}\ t_1\texttt{;}\ \texttt{otherwise}\ t_2$, l must occur in at least one of its subterms t_0, t_1, or t_2. Then, by inductive hypothesis, $\mathbf{M}(\, \llbracket t_0[x/l] \rrbracket_\rho \,) < \mathbf{M}(t_0)$, $\mathbf{M}(\, \llbracket t_1[x/l] \rrbracket_{\rho \setminus \mathrm{vars}(p)} \,) < \mathbf{M}(t_1)$, or $\mathbf{M}(\, \llbracket t_2[x/l] \rrbracket_\rho \,) < \mathbf{M}(t_2)$. As a consequence, $\mathbf{M}(\, t_x \,) < \mathbf{M}(\, t \,)$.

In the case that $\emptyset \neq \mathrm{freevars}(t) \subseteq \mathrm{dom}\ \rho$, there must be a t_k, $k \in \{0, 1, 2\}$, with $\mathrm{freevars}(t_k) \neq \emptyset$, as a variable occurring free in t must occur free in at least one of its subterms. For $k \in \{0, 2\}$, $\mathrm{freevars}(t_k) \subseteq \mathrm{freevars}(t) \subseteq \mathrm{dom}\ \rho$, such that, by inductive hypothesis, $\mathbf{M}(\, \llbracket t_k[x/l] \rrbracket_\rho \,) < \mathbf{M}(\, t_k \,)$. For $k = 1$, we observe that the variables $\mathrm{vars}(p)$ occurring in p are bound in t_1, such that

$$\begin{aligned}
\mathrm{freevars}(t_1) &\subseteq \mathrm{freevars}(t) \setminus \mathrm{vars}(p) \\
&\subseteq \mathrm{dom}\, \rho \setminus \mathrm{vars}(p) \;=\; \mathrm{dom}\ (\rho \setminus \mathrm{vars}(p)).
\end{aligned}$$

Then, by inductive hypothesis, $\mathbf{M}(\, \llbracket t_1[x/l] \rrbracket_{\rho \setminus \mathrm{vars}(p)} \,) < \mathbf{M}(\, t_1 \,)$. As a consequence, $\mathbf{M}(\, t_x \,) < \mathbf{M}(\, t \,)$.

9. Finite Quantifications and Comprehension Terms

   With respect to the binding of variables by means of pattern matching, the same reasoning carried out above for `case`-terms applies. See also the proof for the case of `do forall`-rules in Lemma D.9 below, which is essentially the same proof as for quantification and comprehension terms.

Lemma D.8 Let t be a (partially evaluated) term of ASM-SL and ρ an environment containing bindings for all variables occurring free in t, i.e., such that $\mathrm{freevars}(t) \subseteq \mathrm{dom}\ \rho$. Then, either $\llbracket t \rrbracket_\rho$ contains locations or $\llbracket t \rrbracket_\rho$ is a value.

Proof The proof is by induction on the structure of the term t. There is a case for each basic form of term.

1. Values: $t \equiv x$

   $\llbracket t \rrbracket_\rho = \llbracket x \rrbracket_\rho = x$ is a value.

2. Locations: $t \equiv l \equiv (f, x)$

   $\llbracket t \rrbracket_\rho = \llbracket l \rrbracket_\rho = l$ is a location.

   Thus, $\llbracket t \rrbracket_\rho$ is not a value and in fact it contains locations, namely the location l itself.

3. Special Constants: $t \equiv \mathit{int\_const} \mid \mathit{float\_const} \mid \mathit{string\_const}$

   Special case: application of nullary static function name (see below).

4. Variables (Applied Occurrences): $t \equiv v$

   By hypothesis, ρ contains bindings for all variables occurring free in t. This means, in particular, that $v \in \operatorname{dom} \rho$. Therefore $[\![t]\!]_\rho = [\![v]\!]_\rho = \rho(v)$ is a value.

5. Tuple Terms: $t \equiv (t_1, \ldots, t_n)$

   (a) If $[\![t_i]\!]_\rho = x_i$ for all $i \in \{1, \ldots, n\}$, i.e., each $[\![t_i]\!]_\rho$ is a value:

   $$[\![t]\!]_\rho = [\![(t_1, \ldots, t_n)]\!]_\rho = (x_1, \ldots, x_n) \text{ is a value.}$$

   (b) Otherwise, there is at least one t_k, $1 \leq k \leq n$, such that $[\![t_k]\!]_\rho$ is not a value. By inductive hypothesis, $[\![t_k]\!]_\rho$ contains locations. As

   $$[\![t]\!]_\rho = [\![(t_1, \ldots, t_n)]\!]_\rho = ([\![t_1]\!]_\rho, \ldots, [\![t_n]\!]_\rho)$$

   by definition of $[\![\cdot]\!]$, $[\![t_k]\!]_\rho$ is a subterm of $[\![t]\!]_\rho$. Therefore, $[\![t]\!]_\rho$ contains locations (namely: at least all locations in $[\![t_k]\!]_\rho$).

6. Function Applications: $t \equiv f(t')$

   (a_1) If $[\![t']\!]_\rho = x$ (i.e., $[\![t']\!]_\rho$ is a value) and f is a static function name:

   $$[\![t]\!]_\rho = [\![f(t')]\!]_\rho = \mathbf{f}(x) \text{ is a value.}$$

   (a_2) If $[\![t']\!]_\rho = x$ and f is a dynamic/external function name:

   $$[\![t]\!]_\rho = [\![f(t')]\!]_\rho = (f, x) \text{ is a location (see case 2., Locations).}$$

   (b) Otherwise ($[\![t']\!]_\rho$ is not a value):

   $$[\![t]\!]_\rho = [\![f(t')]\!]_\rho = f([\![t']\!]_\rho).$$

   As $[\![t']\!]_\rho$ is not a value, it contains locations (inductive hypothesis). Then, $[\![t]\!]_\rho = f([\![t']\!]_\rho)$ contains locations. In particular, it contains the same locations as $[\![t']\!]_\rho$.

7. Conditional Terms: $t \equiv \texttt{if}\ G\ \texttt{then}\ t_T\ \texttt{else}\ t_F$

   (a_1) If $[\![G]\!]_\rho = \mathbf{true}$:

   $$[\![t]\!]_\rho \;=\; [\![\texttt{if}\ G\ \texttt{then}\ t_T\ \texttt{else}\ t_F]\!]_\rho \;=\; [\![t_T]\!]_\rho$$

   By inductive hypothesis, $[\![t_T]\!]_\rho$ is either a value or contains locations, and so does $[\![t]\!]_\rho$.

   (a_2) If $[\![G]\!]_\rho = \mathbf{false}$:

   The same as in the case $[\![G]\!]_\rho = \mathbf{true}$, with t_T replaced by t_F.

(b) Otherwise ($[\![G]\!]_\rho$ is not a value):

$$\begin{array}{rcl} [\![t]\!]_\rho & = & [\![\texttt{if } G \texttt{ then } t_T \texttt{ else } t_F]\!]_\rho \\ & = & \texttt{if } [\![G]\!]_\rho \texttt{ then } [\![t_T]\!]_\rho \texttt{ else } [\![t_F]\!]_\rho \end{array}$$

As $[\![G]\!]_\rho$ is not a value, it contains locations (inductive hypothesis). Then, $[\![t]\!]_\rho$ contains locations (at least all locations in $[\![G]\!]_\rho$).

8. Case Terms: $t \equiv \texttt{case } t_0 \texttt{ of } p : t_1\texttt{; otherwise } t_2$

(a_1) If $[\![t_0]\!]_\rho = x$ (i.e., $[\![t_0]\!]_\rho$ is a value) and $\mathcal{M}[\![\, p \,]\!](x) \neq \text{FAIL}$:

$$\begin{array}{rcl} [\![t]\!]_\rho & = & [\![\texttt{case } t_0 \texttt{ of } p : t_1\texttt{; otherwise } t_2]\!]_\rho \\ & = & [\![t_1]\!]_{\rho \oplus \mathcal{M}[\![p]\!](x)} \end{array}$$

By inductive hypothesis, $[\![t_1]\!]_{\rho \oplus \mathcal{M}[\![p]\!](x)}$ is either a value or contains locations, and so does $[\![t]\!]_\rho$.

Note that the inductive hypothesis is applicable to subterm t_1, as $\text{freevars}(t) \subseteq \text{dom } \rho$ implies

$$\begin{array}{rcl} \text{freevars}(t_1) & = & \text{freevars}(t) \cup \text{vars}(p) \\ & \subseteq & \text{dom } \rho \cup \text{vars}(p) \\ & = & \text{dom } (\rho \oplus \mathcal{M}[\![\, p \,]\!](x)) \,. \end{array}$$

(a_2) If $[\![t_0]\!]_\rho = x$ and $\mathcal{M}[\![\, p \,]\!](x) = \text{FAIL}$:

$$\begin{array}{rcl} [\![t]\!]_\rho & = & [\![\texttt{case } t_0 \texttt{ of } p : t_1\texttt{; otherwise } t_2]\!]_\rho \\ & = & [\![t_2]\!]_\rho \end{array}$$

By inductive hypothesis, $[\![t_2]\!]_\rho$ is either a value or contains locations, and so does $[\![t]\!]_\rho$.

(b) Otherwise ($[\![t_0]\!]_\rho$ is not a value):

$$\begin{array}{rcl} [\![t]\!]_\rho & = & [\![\texttt{case } t_0 \texttt{ of } p : t_1\texttt{; otherwise } t_2]\!]_\rho \\ & = & \texttt{case } [\![t_0]\!]_\rho \texttt{ of } p : [\![t_1]\!]_{\rho \setminus \text{vars}(p)} \texttt{ ; otherwise } [\![t_2]\!]_\rho \end{array}$$

As $[\![t_0]\!]_\rho$ is not a value, it contains locations (inductive hypothesis). Then, $[\![t]\!]_\rho$ contains locations (at least all locations in $[\![t_0]\!]_\rho$).

9. Finite Quantifications and Comprehension Terms

See the proof for the case of `do forall`-rules in Lemma D.4 below, which is essentially the same proof.

The above lemma can be reformulated in terms of the measure function as follows.

Corollary D.1 Let t be a term, ρ an environment with $\text{freevars}(t) \subseteq \text{dom } \rho$, and $M = (b, s, n) = \mathbf{M}([\![t]\!]_\rho)$. If $n = 0$, then $b = \epsilon$ and $s = 0$.

D.3.2 Rule Unfolding

Lemma D.9 Let R be a (partially evaluated) rule of ASM-SL, ρ an environment, x a value, and l a location. Let $R_x = [\![R[x/l]]\!]_\rho$. Then:

(L_1) $\mathbf{M}(\, R_x \,) \leq \mathbf{M}(\, R \,)$.

(L_2) If: *(i)* l occurs in R, but not only as left-hand side of update rules, or *(ii)* R contains free variables and ρ contains bindings for all free variables in R, i.e., $\emptyset \neq \text{freevars}(t) \subseteq \text{dom } \rho$; then: $\mathbf{M}(\, R_x \,) < \mathbf{M}(\, R \,)$.

Proof As for Lemma D.4, the proof is done by induction on the structure of the rule R. There is a case for each basic form of rule.

1. Skip Rules: $R \equiv$ `skip`

   $R_x = [\![\texttt{skip}[x/l]]\!]_\rho = [\![\texttt{skip}]\!]_\rho =$ *(empty block)*

   Thus, $\mathbf{M}(\, R_x \,) = \mathbf{M}(\,$ *empty block* $\,) = (\epsilon, 0, 0) < (\epsilon, 1, 0) = \mathbf{M}(\,$`skip`$\,) = \mathbf{M}(\, R \,)$.

   (L_2) holds, as $\mathbf{M}(\, R_x \,) < \mathbf{M}(\, R \,)$.

2. Update Rules. We distinguish two cases. In the first case the left-hand side of R is a location ($R \equiv l' := t'$), in the second case it is an application term ($R \equiv f(t) := t'$).

   (a) $R \equiv l' := t'$

   $$\begin{array}{rcl} R_x & = & [\![(l' := t')[x/l]\,]\!]_\rho \\ & = & [\![\, l' := t'[x/l]\,]\!]_\rho \\ & = & [\![\, l'\,]\!]_\rho := [\![t'[x/l]]\!]_\rho \\ & = & l' := [\![t'[x/l]]\!]_\rho \end{array}$$

   $$\begin{array}{rcl} \mathbf{M}(\, R_x \,) & = & \mathbf{M}(\, l' := [\![t'[x/l]]\!]_\rho \,) \\ & = & \mathbf{M}(\, [\![t'[x/l]]\!]_\rho \,) \\ & \leq & \mathbf{M}(\, t' \,) \\ & = & \mathbf{M}(\, l' := t' \,) \\ & = & \mathbf{M}(\, R \,) \end{array}$$

   (L_2) holds for the following reason.

   If l occurs in R, but not only as left-hand side of update rules, then l must occur in t'. It follows, by Lemma D.9, $\mathbf{M}(\, [\![t'[x/l]]\!]_\rho \,) < \mathbf{M}(t')$. As a consequence, $\mathbf{M}(\, t_x \,) < \mathbf{M}(\, t \,)$.

   For the case $\emptyset \neq \text{freevars}(t) \subseteq \text{dom } \rho$, the proof is similar: in fact, a variable x can only occur free in $R \equiv l' := t'$ if it occurs free in t'.

(b) $R \equiv f(t) := t'$

$$\begin{array}{lcl} R_x & = & [\![(f(t) := t')[x/l]]\!]_\rho \\ & = & [\![(f(t))[x/l] := t'[x/l]]\!]_\rho \\ & = & [\![(f(t))[x/l]]\!]_\rho := [\![t'[x/l]]\!]_\rho \end{array}$$

$$\begin{array}{lcl} \mathbf{M}(\, R_x \,) & = & \mathbf{M}(\, [\![(f(t))[x/l]]\!]_\rho := [\![t'[x/l]]\!]_\rho \,) \\ & = & (\epsilon, 1, 0) + \mathbf{M}(\, [\![(f(t))[x/l]]\!]_\rho \,) + \mathbf{M}(\, [\![t'[x/l]]\!]_\rho \,) \\ & \leq & (\epsilon, 1, 0) + \mathbf{M}(\, f(t) \,) + \mathbf{M}(\, t' \,) \\ & = & \mathbf{M}(\, f(t) := t' \,) \\ & = & \mathbf{M}(\, R \,) \end{array}$$

(L_2) can be proved in the same way as above. If l occurs in R, l must occur in at least one of the subterms t, t'. The same applies for the case $\emptyset \neq \text{freevars}(t) \subseteq \text{dom } \rho$.

3. Block Rules: $R \equiv R_1 \ \ldots \ R_n$

$$\begin{array}{lcl} R_x & = & [\![(R_1 \ \ldots \ R_n)[x/l]]\!]_\rho \\ & = & [\![R_1[x/l] \ \ldots \ R_n[x/l]]\!]_\rho \\ & = & [\![R_1[x/l]]\!]_\rho \ \ldots \ [\![R_n[x/l]]\!]_\rho \end{array}$$

$$\begin{array}{lcl} \mathbf{M}(\, R_x \,) & = & \mathbf{M}(\, [\![R_1[x/l]]\!]_\rho \ \ldots \ [\![R_n[x/l]]\!]_\rho \,) \\ & = & \sum_{i=1}^{n} \mathbf{M}(\, [\![R_i[x/l]]\!]_\rho \,) \\ & \leq & \sum_{i=1}^{n} \mathbf{M}(\, R_i \,) \\ & = & \mathbf{M}(\, R_1 \ \ldots \ R_n \,) \\ & = & \mathbf{M}(\, R \,) \end{array}$$

The proof of (L_2) is done in the same way as for the case (b) of Tuple Terms in Lemma D.7. (The subrules $R_1, \ldots, R_n$ of the block rule R play here the same role as the subterms $t_1, \ldots, t_n$ of the tuple term t there.)

4. Conditional Rules: $R \equiv \texttt{if}\ G\ \texttt{then}\ R_T\ \texttt{else}\ R_F$

Same proof as for conditional terms.

5. Case Rules: $R \equiv \texttt{case}\ t_0\ \texttt{of}\ p\ \texttt{:}\ R_1\texttt{;}\ \texttt{otherwise}\ R_2$

Same proof as for `case`-terms.

6. Do-Forall Rules: $R \equiv \texttt{do forall}\ p\ \texttt{in}\ A \ \ R'$

$$\begin{array}{lcl} R_x & = & [\![(\texttt{do forall}\ p\ \texttt{in}\ A \ \ R')[x/l]]\!]_\rho \\ & = & [\![\texttt{do forall}\ p\ \texttt{in}\ A[x/l] \ \ R'[x/l]]\!]_\rho \end{array}$$

(a) If $[\![A[x/l]]\!]_\rho = X$ (i.e., $[\![A[x/l]]\!]_\rho$ is a value, in particular a finite set):

Let $X' = \{x_1, \ldots, x_n\}$ be the subset of X defined by

$$X' = \{\, x \in X \mid \mathcal{M}[\![\, p \,]\!](x) \neq \text{FAIL} \,\}$$

(like in the definition of $[\![\cdot]\!]$). Then:

$$\begin{array}{rcl}
\mathbf{M}(\,R_x\,) & = & \mathbf{M}(\,[\![\texttt{do forall}\ p\ \texttt{in}\ A[x/l]\ \ R'[x/l]]\!]_\rho\,) \\
& = & \mathbf{M}\begin{pmatrix} [\![R'[x/l]]\!]_{\rho\oplus\mathcal{M}[\![p]\!](x_1)} \\ \vdots \\ [\![R'[x/l]]\!]_{\rho\oplus\mathcal{M}[\![p]\!](x_n)} \end{pmatrix} \\
& = & \sum_{x\in X'} \mathbf{M}(\,[\![R'[x/l]]\!]_{\rho\oplus\mathcal{M}[\![p]\!](x)}\,) \\
& \leq & \sum_{x\in X'} \mathbf{M}(\,R'\,) \\
& < & (1)\bullet\mathbf{M}(\,R'\,) \\
& < & (\epsilon,1,0)+\mathbf{M}(A)+(1)\bullet\mathbf{M}(R') \\
& = & \mathbf{M}(\,\texttt{do forall}\ p\ \texttt{in}\ A\ \ R'\,) \\
& = & \mathbf{M}(\,R\,)
\end{array}$$

The inequality $\sum_{x\in X'} \mathbf{M}(\,R'\,) < (1)\bullet\mathbf{M}(\,R'\,)$ above holds for the following reason:

$$\begin{array}{rl}
 & |\,b\,(\sum_{x\in X'} \mathbf{M}(\,R'\,)\,)\,| \\
= & |\,b\,(\mathbf{M}(\,R'\,)\,)\,| \\
< & 1+|\,b\,(\mathbf{M}(\,R'\,)\,)\,| \\
= & |\,b\,((1)\bullet\mathbf{M}(\,R'\,))\,|
\end{array}$$

Then, $\sum_{x\in X'} \mathbf{M}(\,R'\,) < (1)\bullet\mathbf{M}(\,R'\,)$ follows from the fact that, for all $b_1, b_2 \in B$, $|\,b_1\,| < |\,b_2\,| \Rightarrow\ b_1 < b_2$, and from the lexicographic ordering of measures.

(L_2) holds, as $\mathbf{M}(\,R_x\,) < \mathbf{M}(\,R\,)$.

(b) Otherwise ($[\![A[x/l]]\!]_\rho$ is not a value):

$$\begin{array}{l}
R_x = [\![\texttt{do forall}\ p\ \texttt{in}\ A[x/l]\ \ R'[x/l]]\!]_\rho \\
\quad = \texttt{do forall}\ p\ \texttt{in}\ [\![A[x/l]]\!]_\rho\ \ [\![R'[x/l]]\!]_{\rho\setminus\mathrm{vars}(p)}
\end{array}$$

The proof for this case is analogous with the proof for case (b) of Case Terms, Lemma D.7.

Lemma D.10 Let R be a (partially evaluated) rule of ASM-SL and ρ an environment containing bindings for all variables occurring free in R, i.e., such that freevars(R) $\subseteq$ dom ρ. Then:

- either $[\![R]\!]_\rho$ contains locations that are not left-hand sides of update rules, or
- $[\![R]\!]_\rho$ is a rule of the form $l_1 := x_1\ \ldots\ l_n := x_n$, i.e., a block consisting only of simple update rules *location* := *value* ("elementary update block").

Proof The proof is by induction on the structure of the rule R. There is a case for each basic form of rule.

1. Skip Rules: $R \equiv \texttt{skip}$

   $[\![R]\!]_\rho = [\![\texttt{skip}]\!]_\rho =$ *(empty block)*.

   $[\![R]\!]_\rho$ is an (empty) elementary update block.

2. Update Rules: We distinguish two cases here. In the first case the left-hand side of R is a location ($R \equiv l := t'$), in the second case it is an application term ($R \equiv f(t) := t'$).

   (a) $R \equiv l := t'$

   $$\begin{array}{rcl} [\![R]\!]_\rho & = & [\![\, l := t' \,]\!]_\rho \\ & = & [\![\, l \,]\!]_\rho := [\![t']\!]_\rho \\ & = & l := [\![t']\!]_\rho \end{array}$$

   By Lemma D.8, $[\![t']\!]_\rho$ is either a value or contains locations. If it is a value, then $[\![R]\!]_\rho$ is an elementary update block (of length 1). Otherwise, it contains locations that are not left-hand sides of update rules (the locations in $[\![t']\!]_\rho$).

   (b) $R \equiv f(t) := t'$

   Two subcases has to be considered here, depending on whether $[\![t]\!]_\rho$ is a value or not.

   i. $[\![t]\!]_\rho = x$ (i.e., $[\![t]\!]_\rho$ is a value):

   $$\begin{array}{rcl} [\![R]\!]_\rho & = & [\![\, f(t) := t' \,]\!]_\rho \\ & = & [\![f(t)]\!]_\rho := [\![t']\!]_\rho \\ & = & (f, x) := [\![t']\!]_\rho \end{array}$$

   In this case, the left-hand side of $[\![R]\!]_\rho$ is a location: see (a).

   ii. Otherwise ($[\![t]\!]_\rho$ is not a value):

   $$\begin{array}{rcl} [\![R]\!]_\rho & = & [\![\, f(t) := t' \,]\!]_\rho \\ & = & [\![f(t)]\!]_\rho := [\![t']\!]_\rho \\ & = & f([\![t]\!]_\rho) := [\![t']\!]_\rho \end{array}$$

   By Lemma D.8, $[\![t]\!]_\rho$ contains locations. Thus, $[\![R]\!]_\rho$ contains locations that are not left-hand sides of update rules.[1]

3. Block Rules: $R \equiv R_1 \ldots R_n$

   $$\begin{array}{rcl} [\![R]\!]_\rho & = & [\![R_1 \ldots R_n]\!]_\rho \\ & = & [\![R_1]\!]_\rho \ldots [\![R_n]\!]_\rho \end{array}$$

   By inductive hypothesis, each $[\![R_i]\!]_\rho$ is either an elementary update block or contains locations that are not left-hand sides of update rules.

[1] Note that, in this case, locations occur on the left-hand side of the update rule $[\![R]\!]_\rho$, but *are not* the left-hand side of $[\![R]\!]_\rho$.

If every $[\![R_i]\!]_\rho$ is an elementary update block, then $[\![R]\!]_\rho$—which results from their concatenation—is an elementary update block as well.

Otherwise, at least one $[\![R_i]\!]_\rho$ contains locations that are not left-hand sides of update rules. So does $[\![R]\!]_\rho$, as $[\![R_i]\!]_\rho$ is one of its subrules.

4. CONDITIONAL RULES: $R \equiv \texttt{if}\ G\ \texttt{then}\ R_T\ \texttt{else}\ R_F$

   Similar proof as for conditional terms in Lemma D.8.

5. CASE RULES: $R \equiv \texttt{case}\ t_0\ \texttt{of}\ p\ \texttt{:}\ R_1\texttt{;}\ \texttt{otherwise}\ R_2$

   Similar proof as for `case`-terms in Lemma D.8.

6. DO-FORALL RULES: $R \equiv \texttt{do forall}\ p\ \texttt{in}\ A\ \ R'$

   (a) If $[\![A]\!]_\rho = X$ (i.e., $[\![A]\!]_\rho$ is a value, in particular a finite set):

   Let $X' = \{\, x \in X \mid \mathcal{M}[\![\, p \,]\!](x) \neq \text{FAIL} \,\} = \{x_1, \ldots, x_n\}$. Then:

   $$\begin{array}{rcl} [\![R]\!]_\rho & = & [\![\texttt{do forall}\ p\ \texttt{in}\ A\ \ R']\!]_\rho \\ & = & [\![R']\!]_{\rho \oplus \mathcal{M}[\![p]\!](x_1)} \ \cdots\ [\![R']\!]_{\rho \oplus \mathcal{M}[\![p]\!](x_n)} \end{array}$$

   By inductive hypothesis, for each i, $1 \leq i \leq n$, $[\![R']\!]_{\rho \oplus \mathcal{M}[\![p]\!](x_i)}$ is either an elementary update block or contains locations that are not left-hand sides of update rules. (The inductive hypothesis applies, as $\text{freevars}(R) \subseteq \text{dom}\ \rho$ implies $\text{freevars}(R') \subseteq \text{dom}\ (\rho \oplus \mathcal{M}[\![\, p \,]\!](x_i))$: see discussion of `case`-terms, case a_1, in the proof of Lemma D.8.)

   Then, the proof for this case is the same as for block rules.

   (b) Otherwise ($[\![A]\!]_\rho$ is not a value):

   $$\begin{array}{rcl} [\![R]\!]_\rho & = & [\![\texttt{do forall}\ p\ \texttt{in}\ A\ \ R']\!]_\rho \\ & = & \texttt{do forall}\ p\ \texttt{in}\ [\![A]\!]_\rho\ \ [\![R']\!]_{\rho \setminus \text{vars}(p)} \end{array}$$

   By Lemma D.8, as $[\![A]\!]_\rho$ is not a value, it contains locations. Then, the rule $[\![R]\!]_\rho$—which clearly is not an elementary update block—contains locations that are not left-hand sides of update rules, namely the locations in $[\![A]\!]_\rho$.

The above lemma can be reformulated in terms of the measure function as follows.

Corollary D.2 Let R be a rule, ρ an environment with $\text{freevars}(R) \subseteq \text{dom}\ \rho$, and $M = (b, s, n) = \mathbf{M}([\![R]\!]_\rho)$. If $n = 0$, then $b = \epsilon$ and $s = 0$.

D.4 Main Theorem

Theorem 7.1 For every closed rule R, not containing `choose`-rules, derived functions, or `FUN_TO_MAP`/`REL_TO_SET` operators:

- $\mathcal{E}(R)$ is defined, i.e., $\mathcal{E}$ applied to R terminates.
- $\overline{R} = \mathcal{E}(R)$ is an ASM_0 rule.
- In all states S, $\Delta_S(\overline{R}) = \Delta_S(R)$, i.e., $\overline{R}$ is semantically equivalent to R.

Proof The termination of $\mathcal{E}$ for every rule R and the fact that $R_0 = \mathcal{E}(R)$ is an ASM_0 program follow from Lemma D.9, Lemma D.10, and Corollary D.2.[2] In fact, by looking at the recursive definition of $\mathcal{E}$, it can be observed that:

1. If R contains no locations that are not left-hand sides of update rules, then $\mathcal{E}(R) = R$, i.e., $\mathcal{E}$ terminates immediately.
2. Otherwise, $\mathcal{E}$ is recursively applied to rules $R_{x_i} \equiv [\![R[x_i/l]]\!]$, whose measure (by Lemma D.9) is strictly less than the measure of R. Then, for each chain of recursive applications of $\mathcal{E}$ to rules R_j, $j = 0, 1, \ldots$—where $R_0 = R$ and $R_j = [\![R_{j-1}[x_i/l]]\!]$ for some location l occurring in R_{j-1} and some $x_i \in \text{ran } l$—the measures of R_j build a strictly decreasing sequence

   $$\mathbf{M}(\, R_0\,) > \mathbf{M}(\, R_1\,) > \ldots$$

   Thus, in each chain there will be a rule R_k with $\mathbf{M}(\, R_k\,) = (b, s, n)$ and $n = 0$, i.e., a rule R_k which does not contain any locations which are not left-hand sides of update rules. At this point, the chain of recursive applications terminates (case 1. applies). Moreover, by Lemma D.10 or Corollary D.2, R_k is an elementary update block, i.e., a rule of the form $l_1 := x_1 \ \ldots \ l_n := x_n$, whose measure is $(\epsilon, 0, 0)$.

Clearly, $\mathcal{E}(R)$ is an ASM_0 program, as it consists exclusively of a decision tree built out of nested conditional rules with guards of the form *location* = *value*, where the leaves of the tree are elementary update blocks.

Finally, the correctness of the transformation follows from Lemma D.6, which states that, if $\mathcal{E}$ applied to R terminates (which is in fact always the case, as shown above), then $\Delta_S(\mathcal{E}(R)) = \Delta_S(R)$, i.e., the unfolding transformation preserves the semantics of rules.

[2] We rely on termination of the term- and rule-simplifying transformation $[\![\,\cdot\,]\!]$, which has not been proved, but follows immediately from the fact that—as opposite to the unfolding transformation $\mathcal{E}$—$[\![\,\cdot\,]\!]$ is defined by induction on the structure of terms and rules, respectively.

Bibliography

[1] J.-R. Abrial, E. Börger, and H. Langmaack, editors. *Formal Methods for Industrial Applications: Specifying and Programming the Steam Boiler Control*, volume 1165 of *LNCS (State–of–the–Art Survey)*. Springer, 1996.

[2] M. Anlauff. Aslan: Programming in Abstract State Machines (draft of the language specification). Available electronically in WWW under `http://www.first.gmd.de/~ma/projects.htm`, 1998.

[3] M. Anlauff, P. Kutter, and A. Pierantonio. Formal Aspects of and Development Environments for Montages. In M. Sellink, editor, *2nd International Workshop on the Theory and Practice of Algebraic Specifications*, Workshops in Computing, Amsterdam, 1997. Springer.

[4] A. Bartoloni et al. A hardware implementation of the APE100 architecture. *International Journal of Modern Physics*, C 4, 1993.

[5] A. Bartoloni et al. The software of the APE100 processor. *International Journal of Modern Physics*, C 4, 1993.

[6] B. Beckert and J. Posegga. leanEA: A Lean Evolving Algebra Compiler. In H. Kleine Büning, editor, Proceedings of the Annual Conference of the European Association for Computer Science Logic (CSL'95), volume 1092 of *LNCS*, pages 64–85. Springer, 1996.

[7] C. Beierle, E. Börger, I. Durdanovic, U. Glässer, and E. Riccobene. Refining Abstract Machine Specifications of the Steam Boiler Control to Well Documented Executable Code. In J.-R. Abrial, E. Börger, and H. Langmaack, editors, *Formal Methods for Industrial Applications. Specifying and Programming the Steam-Boiler Control*, number 1165 in LNCS, pages 62–78. Springer, 1996.

[8] J.A. Bergstra, J. Heering, and P. Klint. *Algebraic Specification*. ACM Press and Addison-Wesley, 1989.

[9] E. Best et al. A class of composable high level Petri nets. In G. De Michelis and M. Diaz, editors, *Proc. of ATPN'95 (Application and Theory of Petri Nets), Torino*, volume 935 of *LNCS*, pages 103–118. Springer, 1995.

[10] A. Blass, Y. Gurevich, and S. Shelah. Choiceless Polynomial Time. Technical Report CSE-TR-338-97, EECS Dept., University of Michigan, 1997.

[11] A. Bo, R. Eschbach, U. Glässer, R. Gotzhein, A. Prinz, W. Ying, Z. Yuhong, and Z. Weilei. The Formal Semantics of SDL (draft, available electronically at `http://tseg.bupt.edu.cn/`). Technical report, Beijing University of Posts and Telecommunication, 1999.

[12] G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide*. Addison-Wesley, 1999.

[13] E. Börger. Annotated Bibliography on Evolving Algebras. In E. Börger, editor, *Specification and Validation Methods*, pages 37–51. Oxford University Press, 1995.

[14] E. Börger. High level system design and analysis using Abstract State Machines. In D. Hutter et al., editors, *Current Trends in Applied Formal Methods (FM-Trends 98)*, volume 1641 of *LNCS*, pages 1–43. Springer, 1999.

[15] E. Börger and G. Del Castillo. A formal method for provably correct composition of a real-life processor out of basic components (The APE100 Reverse Engineering Study). In B. Werner, editor, *Proceedings of the First IEEE International Conference on Engineering of Complex Computer Systems (ICECCS'95)*, pages 145–148, November 1995.

[16] E. Börger, U. Glässer, and W. Müller. The Semantics of Behavioral VHDL'93 Descriptions. In *EURO-DAC'94. European Design Automation Conference with EURO-VHDL'94*, pages 500–505, Los Alamitos, California, 1994. IEEE CS Press.

[17] E. Börger, U. Glässer, and W. Müller. Formal Definition of an Abstract VHDL'93 Simulator by EA-Machines. In C. Delgado Kloos and P. T. Breuer, editors, *Formal Semantics for VHDL*, pages 107–139. Kluwer Academic Publishers, 1995.

[18] E. Börger and J. Huggins. Abstract State Machines 1988-1998: Commented ASM Bibliography. *Bulletin of EATCS*, 64, February 1998. Available in electronic form at `http://www.eecs.umich.edu/gasm` and `http://www.uni-paderborn.de/cs/asm.html`.

[19] E. Börger and L. Mearelli. Integrating ASMs into the Software Development Life Cycle. *Journal of Universal Computer Science*, 3(5):603–665, 1997.

[20] E. Börger and W. Schulte. A Modular Design for the Java VM architecture. In E. Börger, editor, *Architecture Design and Validation Methods*. Springer, 1999.

[21] 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*, volume 1523 of *LNCS*. Springer, 1999.

[22] J. Buck, S. Ha, A. Lee, and D.G. Messerschmidt. Ptolemy: a framework for simulation and prototyping heterogeneous systems. *International Journal of Computer Simulation, Special issue on Simulation Software Development*, 4:155–182, April 1994.

[23] L. Cardelli. Type systems. In A.B. Tucker, editor, *The Computer Science and Engineering Handbook*, chapter 103, pages 2208–2236. CRC Press, 1997.

[24] W. Chan, R.J. Anderson, P. Beame, S. Burns, F. Modugno, D. Notkin, and J.D. Reese. Model checking large software specifications. *IEEE Transactions on Software Engineering*, 24(7):498–520, July 1998.

[25] E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999.

[26] E. Clarke and J. Wing. Formal methods: State of the art and future directions — Report by the Working Group on Formal Methods for the ACM Workshop on Strategic Directions in Computing Research. *ACM Computing Surveys*, 28(4):626–643, December 1996.

[27] L. Damas and R. Milner. Principal type schemes for functional programs. In *Proceedings of the 9th ACM Symposium on Principles of Programming Languages*, pages 207–212, 1982.

[28] W. Damm, H. Hungar, P. Kelb, and R. Schlör. Using graphical specification languages and symbolic model checking in the verification of a production cell. In *[63]*.

[29] N. Day. A model checker for Statecharts (Linking CASE tools with formal methods). Technical Report 93–35, Dept. of Computer Science, Univ. of British Columbia, Vancouver, B.C., Canada, October 1993.

[30] G. Del Castillo. Towards comprehensive tool support for Abstract State Machines: The ASM Workbench tool environment and architecture. In D. Hutter et al., editors, *Current Trends in Applied Formal Methods (FM-Trends 98)*, volume 1641 of *LNCS*, pages 311–325. Springer, 1999.

[31] G. Del Castillo and U. Glässer. Computer-aided analysis and validation of heterogeneous system specification. In *Computer Aided Systems Theory (EUROCAST'99)*, LNCS. Springer, 2000 (to appear).

[32] G. Del Castillo and W. Hardt. Fast dynamic analysis of complex HW/SW-systems based on Abstract State Machine models. In *Proc. of the 6th Int. Workshop on Hardware/Software Codesign (CODES/CASHE'98)*, 1998.

[33] G. Del Castillo and K. Winter. Model checking support for the ASM high-level language. In S. Graf and M. Schwartzbach, editors, *Tools and Algorithms for the Construction and Analysis of Systems, TACAS'2000*, volume 1785 of *LNCS*. Springer, 2000.

[34] D. Diesen. *Specifying Algorithms Using Evolving Algebra. Implementation of Functional Programming Languages.* Dr. scient. degree thesis, Dept. of Informatics, University of Oslo, Norway, March 1995.

[35] E.W. Dijkstra. Hierarchical ordering of sequential processes. *Acta Informatica*, 1:115–138, 1971.

[36] I. Đurđanović. ASM Virtual Architecture (ASM/VA). Available in WWW at `http://www.uni-paderborn.de/cs/ag-klbue/staff/igor/ASM/`.

[37] FALKO—Fahrplanvalidierung und Konstruktion in Nahverkehrssystemen. Siemens München, ZT SE 4.

[38] U. Glässer. Modelling of concurrent and embedded systems. In F. Pichler and R. Moreno-Díaz, editors, *Computer Aided Systems Theory–EUROCAST'97 (Proc. of the 6th International Workshop on Computer Aided Systems Theory, Las Palmas de Gran Canaria, Spain, Feb. 1997)*, volume 1333 of *LNCS*, pages 108–122. Springer, 1997.

[39] U. Glässer, R. Gotzhein, and A. Prinz. Towards a new formal SDL semantics based on Abstract State Machines. In G. v. Bochmann, R. Dssouli, and Y. Lahav, editors, *9th SDL Forum Proceedings*, pages 171–190. Elsevier Science B.V., 1999.

[40] B. Grahlmann. The PEP Tool. In O. Grumberg, editor, *Proc. of CAV'97 (Computer Aided Verification)*, volume 1254 of *LNCS*, pages 440–443. Springer, 1997.

[41] B. Grahlmann. Combining finite automata, parallel programs and SDL using Petri nets. In B. Steffen, editor, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS'98)*, volume 1384 of *LNCS*, pages 102–117. Springer, 1998.

[42] M. Große-Rhode. On a reference model for the formalization and integration of software specification languages. In *Bulletin of the EATCS n. 68, Formal Specification Column, Part 8, by H. Ehrig*, pages 81–89, June 1999.

[43] C.A. Gunter. *Semantics of programming languages: structures and techniques.* Foundations of Computing Series. MIT Press, 1992.

[44] Y. Gurevich. Evolving Algebras. A Tutorial Introduction. *Bulletin of EATCS*, 43:264–284, 1991.

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

[46] Y. Gurevich. ASM Guide 1997. CSE Technical Report CSE-TR-336-97, EECS Department, University of Michigan–Ann Arbor, 1997.

[47] 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.

[48] Y. Gurevich and J. Huggins. Evolving Algebras and Partial Evaluation. In B. Pehrson and I. Simon, editors, *IFIP 13th World Computer Congress*, volume I: Technology/Foundations, pages 587–592, Elsevier, Amsterdam, the Netherlands, 1994.

[49] N. Halbwachs. *Synchronous Programming of Reactive Systems.* Kluwer Academic Press, Netherlands, 1993.

[50] D. Harel. Statecharts: A visual formulation for complex systems. *Science of Computer Programming*, 8(3):231–274, June 1987.

[51] S. Heinkel and T. Lindner. The Specification and Description Language applied with the SDT support tool. In *[63]*.

[52] J. Hennessy and D. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann Publishers, 1990.

[53] G.J. Holzmann. The model checker Spin. *IEEE Trans. on Software Engineering*, 23(5):279–295, May 1997. Special Issue on Formal Methods in Software Practice.

[54] P. Hudak, S.L. Peyton Jones, and P. Wadler (editors). Report on the programming language Haskell, a non-strict purely functional language (Version 1.2). *SIGPLAN Notices*, Mar, 1992.

[55] J. Huggins and R. Mani. The Evolving Algebra Interpreter Version 2.0. Documentation of the Michigan Evolving Algebra Interpreter. `ftp://ftp.eecs.umich.edu/groups/Ealgebras/interp2.tar.Z`.

[56] J.K. Huggins. Abstract state machines home page. EECS Department, University of Michigan. http://www.eecs.umich.edu/gasm/.

[57] The Institute of Electrical and Electronics Engineers, New York, NY, USA. IEEE Standard VHDL Language Reference Manual—IEEE Std 1076-1993. Order Code SH16840, 1994.

[58] ITU-T Recommendation Z.100. Specification and Description Language (SDL), 1994 + Addendum 1996.

[59] ITU-T Recommendation Z.100 Annex F (Draft). Specification and Description Language (SDL), November 1999.

[60] C.B. Jones. *Systematic Software Development using VDM.* Prentice Hall, 1990.

[61] A.M. Kappel. Executable Specifications Based on Dynamic Algebras. In A. Voronkov, editor, *Logic Programming and Automated Reasoning*, volume 698 of *Lecture Notes in Artificial Intelligence*, pages 229–240. Springer, 1993.

[62] E.A. Lee and A. Sangiovanni-Vincentelli. A framework for comparing models of computation. *IEEE Transaction on Computer-Aided Design of Integrated Circuits and Systems*, 17(12):1217–1229, December 1998.

[63] C. Lewerentz and T. Lindner, editors. *Formal Development of Reactive Systems – Case Study Production Cell*, volume 891 of *LNCS*. Springer, 1995.

[64] P. Liggesmeyer and M. Rothfelder. Towards automated proof of fail-safe behavior. In W. Ehrenberger, editor, *Computer Safety, Reliablity and Security, SAFECOMP'98*, LNCS 1516, pages 169–184, 1998.

[65] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification.* Springer, 1992.

[66] K.L. McMillan. *Symbolic Model Checking.* Kluwer Academic Publishers, 1993.

[67] E. Meijer and J. Jeuring. Merging monads and folds for functional programming. In J. Jeuring and E. Meijer, editors, *Advanced Functional Programming*, volume 925 of *LNCS*, pages 228–266. Springer, 1995.

[68] A. Merceron, M. Müllerburg, and G.M. Pinna. Verifying a time-triggered protocol in a multi-language environment. In W. Ehrenberger, editor, *Computer Safety, Reliability and Security, SAFECOMP'98*, LNCS 1516, pages 185–195, 1998.

[69] E. Mikk, T. Lakhnech, M. Siegel, and G.J. Holzmann. Implementing Statecharts in Promela/Spin. In *Workshop on Industrial-Strength Formal Specification Techniques (WIFT '98), Boca Raton, FL, USA*. IEEE Computer Society Press, 1998.

[70] R. Milner. A theory of type polymorphism in programming. *J. Comput. System Sci.*, 17:348–375, 1978.

[71] R. Milner and M. Tofte. *Commentary on Standard ML.* MIT Press, 1991.

[72] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML.* MIT Press, 1990.

[73] J.C. Mitchell. Type systems for programming languages. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B: Formal Models and Semantics, chapter 8, pages 365–458. North-Holland, New York, N.Y., 1990.

[74] P.D. Mosses. Denotational semantics. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B: Formal Models and Semantics, chapter 11, pages 577–631. The MIT Press, New York, N.Y., 1990.

[75] S. Owre, J.M. Rushby, and N. Shankar. PVS: A Prototype Verification System. In Deepak Kapur, editor, *Proceedings of the 11th International Conference on Automated Deduction (CADE)*, volume 607 of *Lecture Notes in Computer Science*, pages 748–752. Springer-Verlag, 1992.

[76] L.C. Paulson. *ML for the working programmer*. Cambridge Univ. Press, 2nd edition, 1996.

[77] 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, 1998.

[78] W. Reisig. *Petri-Nets: an Introduction*. EATCS Monographs on Theoretical Computer Science. Springer Verlag, 1985.

[79] G. Schellhorn and W. Ahrendt. Reasoning about Abstract State Machines: The WAM Case Study. *Journal of Universal Computer Science*, 3(4):377–413, 1997.

[80] J. Schmid. AsmGofer 1.0. Available electronically in WWW under `http://www.uni-ulm.de/~s_jschmi/AsmGofer`, 1999.

[81] D.A. Schmidt. *The Structure of Typed Programming Languages*. Foundations of Computing Series. MIT Press, 1994.

[82] M. Spielmann. Automatic verification of Abstract State Machines. In N. Halbwachs and D. Peled, editors, *Computer Aided Verification, CAV '99*, number 1633 in LNCS, pages 431–442, Trento, Italy, 1999.

[83] T. Vullinghs, W. Schulte, and T. Schwinn. TkGofer: A functional GUI library. In M. Wirsing and M. Nievat, editors, *Proc. of AMAST'96*, volume 1101 of *LNCS*. Springer, 1996.

[84] P. Wadler and S. Blott. How to make *ad-hoc* polymorphism less *ad hoc*. In *16th Annual ACM Symposium on Principles of Programming Languages (POPL'89)*, pages 60–76, 1989.

[85] C. Wallace. The Semantics of the C++ Programming Language. In E. Börger, editor, *Specification and Validation Methods*, pages 131–164. Oxford University Press, 1995.

[86] K. Winter. Model checking for Abstract State Machines. *Journal of Universal Computer Science*, 3(5):689–701, 1997.

[87] M. Wirsing. Algebraic specifications. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, pages 675–788. Elsevier, 1990.

[88] W.H. Wolf. Hardware-software co-design of embedded systems. *Proceedings of the IEEE*, 82(7):967–989, 1994.

[89] T.Y. Yen and W.H. Wolf. *Hardware software co-synthesis of distributed embedded systems.* Kluwer Academic Publishers, 1996.

[90] W. Zimmerman and T. Gaul. On the Construction of Correct Compiler Back-Ends: An ASM Approach. *Journal of Universal Computer Science*, 3(5):504–567, 1997.