Institut für Technische Informatik und Kommunikationsnetze
Computer Engineering and Networks Laboratory

TIK-SCHRIFTENREIHE NR. XXX

Philipp W. Kutter

Montages
—
Engineering of Computer Languages

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

A dissertation submitted to the
Swiss Federal Institute of Technology Zurich
for the degree of Doctor of Technical Sciences

Diss. ETH No. 13XXX

Prof. Dr. Lothar Thiele, examiner
Prof. Dr. Martin Odersky, co-examiner

Examination date: xxxxx xx, 2003

I would like to thank to my father,
Gerhard Rüdiger Kutter,
whose answers to my questions about his work with fourth
generation languages in banking software have been a seed
from my childhood, which has grown to this thesis.

To my marvelous wife, Enza, to my children,
Nora-Manon Alma Stella and Anaël Gerhard Victor-Maria,
to my mother and my sister.

To the Montages team, Alfonso Pierantonio and Matthias Anlauff,
to Florian Haussman, who was there when everything started, and
to my scientific advisors, Lothar Thiele, Yuri Gurevich, and
Martin Odersky.

For various comments on content and form of the thesis
I am thanksfull mainly to Chuck Wallace, but as well to
Marjan Mernik, Welf Löwe, Asuman Sünbül, Arnd Poetzsch-Heffter,
and Dimidios Spinellis. For inspiring discussions on related areas, thanks
to Craig Cleaveland, David Weiss, Grady Campbell, Erich Gamma, Peter
Mosses, Dusko Pavlovic. For helpfull insight in the business aspects of
Montages, I would like to thank Denis McQuade and Hans-Peter Dieterich.

.

Contents

1

Introduction

In this thesis we elaborate a *language description formalism* called *Montages*. The Montages formalism can be used to engineer *domain specific languages (DSLs)*, which are computer languages specially tailored and typically restricted to solve problems of specific domains. We focus on DSLs which have some algorithmic flavor and are intended to be used in corporate environments where main-stream state-based programming and modeling formalisms[1] prevail.

For engineering such DSLs it is important that the designs of the existing, well known *general purpose languages (GPLs)* can be described as well, and that this descriptions are easily reused as basic building blocks to design new DSLs. Using the Montages tool support *Gem-Mex*, such a new designs can be composed in an integrated semantics environment, and from the descriptions an interpreter and a specialized visual debugger is generated for the new language.

We restrict our research to sequential languages and the technical part of the thesis tries to contribute to the improvement of the DSL design process by focusing on ease of specification and ease of reuse for programming language constructs from well known GPL designs. For the sake of shortness we do not present detailed case studies for DSLs and refer the reader to the literature. Finally, we mainly look at exact reuse of specification modules, and we have not elaborated the means for incremental design by reusing specifications in the sense of object oriented programming. Of course these means are needed as well and we assume the existence of such reuse features without formalizing them. The technical part of the thesis provides the basic specification patterns for introducing all features of an object oriented style of reuse, and applying these patterns to Montages in order to make it an object-oriented specification

[1] Examples are state-machines, as found in UML or State-Charts, flow-charts, and imperative as well as most object-oriented and scripting languages.

formalism is left for future work.

The focus and contribution of this thesis is the design and elaboration of a language engineering discipline based on widely-spread state-based intuition of algorithms and programming. This approach opens the possibility to apply DSL technology in typical corporate environments, where the beneficial properties of smaller, and therefore by nature more secure and more focused computer languages are most leveraged. The thesis does not cover the equally important topic how to formalize these beneficial properties by means of declarative formalisms and how to apply mechanized reasoning and formal software engineering to DSLs.

The thesis is structured in three parts. In the first part the requirements for a language engineering approach are analyzed and the language definition formalism Montages is introduced. In the second part the formal semantics and system architecture of Montages is given. The third part consists of a number of small example languages, each of them designed to show the Montages solution for specifying a well-known feature of main-stream object oriented programming languages such as Java. The single description modules of these example languages can be used to assemble a full object-oriented language, or a small subset of them can be combined with some high-level domain-specific features into a DSL.

In the following we summarize for each part and its chapters their content and relation to each other.

Part I: Engineering of Computer Languages

The first part of this thesis describes the problems we try to solve (Chapter 2), and gives a tutorial introduction to Montages (Chapter 3).

Chapter 2: Requirements for Language Engineering

In this chapter we analyze the problem in the area of language engineering in general, with a special focus on DSLs. A typical application scenario for a DSL with algorithmic flavor is described, and the issue of designing DSLs is discussed. We motivate why the possibility to reuse existing language designs is important even for simple language designs, and show how introducing DSLs and especially language description formalisms allows one to split the development cycle. After these discussions the resulting requirements for a language requirement formalism are summarized and finally related work in the area of language design, domain engineering, and domain specific languages is discussed.

Chapter 3: Montages

The purpose of this chapter is to introduce Montages, a language description formalism we proposed with Pierantonio in 1996 and which has since then be used for descriptions and implementations of GPLs and DSLs in academic and industrial contexts. While a complete formal definition is delegated to Chapter 8, we give here a tutorial introduction. Since for the static semantics we use the well known technique *attribute grammars (AGs)*, we focus on our novel approach for describing dynamic semantics.

In short Montages define dynamic semantics by a mapping from programs to *tree finite state machines (TFSMs)*, a simple tree based state machine model we designed for streamlining the semantics of Montages. The states of such a machine are elements of the Cartesian product of syntax tree nodes, and states in *finite state machines (FSMs)*. The states of the FSMs are in turn associated with action rules. If the TFSM reaches some state, the corresponding action is executed in the environment given by the corresponding node in the syntax tree. The tree structure is defined by traditional EBNF grammars, producing a syntax tree, and the transitions from one node to another in the tree are specified by representing the structure of the tree as nested boxes within the FSMs. The nodes in the tree, whose number can be infinitely large, is associated with a finite number of different FSMs by defining one FSM per production rule in the EBNF, and then associating each node with the FSM corresponding to the production rule which generated the node. The TFSM definition is thus structured along the EBNF rules.

The algorithms for constructing a global FSM from the TFSM, for the simplification of TFSMs, and for the execution of TFSMs are given in an informal way, then special features for the processing of lists, and for the specification of non-local transitions are described. Finally previous results with Montages are summarized and related work in the areas of formal semantics, abstract state machines, and language description environments is discussed.

Part II: Montages Semantics and System Architecture

In the second part the formal semantics and system architecture of Montages are given. The XASM formalism, being used for both giving the formal semantics and implementing the system architecture, is introduced (Chapter 4), then the extension of XASM with parameterizable signature is motivated (Chapter 5), the details of attribute grammars in Montages are given (Chapter 7), and finally, using the previous definition and examples, the formal semantics of Montages is presented in the form of a meta-interpreter, a XASM program which reads both the specification of a language, and a program written in the specified language, and then executes the program according to the language's semantics (Chapter 8). The meta-interpreter can then be partially evaluated to specialized interpreters of the language, and even into compiled code, a process which is sketched in Chapter 5. In this context the parameterization of signatures is used to control the form of the resulting code in order to meet developers requirements on simplicity and transparency of the generated code.

Chapter 4: eXtensible Abstract State Machines (XASM)
The content of this chapter is a motivation and definition of the imperative fast-prototyping formalism *eXtensible Abstract State Machines (*XASM*)*. The XASM language has been devised by Anlauff as implementation basis for Montages. Since XASM have not been defined formally up to now, we contribute here a detailed *denotational semantics* of XASM. XASM is a generalization of Gurevich's ASMs, a state based formalism for specifying algorithms. The basic semantic idea of both ASMs and XASM is that each step of a computation is given by *set of state-changes*. The state itself is given by an algebra. While ASMs propose a fixed update language, the XASM formalism generalized the idea by allowing to introduce *extension functions* whose semantics can be freely calculated by an other ASM or externally implemented functions. In addition XASM feature a group of features building a pure functional sublanguage: constructor terms, pattern matching, and derived functions. If these features are used together with the imperative features an interesting mix of the imperative and the functional paradigm is achieved. Another built in feature of XASM are EBNF grammars. Such a grammar can be decorated with mappings into constructor terms, which are then processed with pattern matching. At the end of the chapter ASM related work is discussed, and a possible challenge of the so called "ASM thesis" is drafted.

Chapter 5: Parameterized XASM
The XASM extension *Parameterized* XASM *(PXasm)* is the topic of this chapter. We designed the novel concept of PXasm in order to allow for freely parameterizing the signature of XASM declarations and rules.

We motivate the necessity of PXasm by showing that it is not possible to generate the kind of syntax trees defined in Chapter 3 with traditional ASMs, since the signature of the trees depends on the symbols in the EBNF. After introducing the new features, we show how the tree generation problem can be solved; we introduce first techniques to navigate in the syntax tree, including examples for specifying abrupt control flow and variable scoping of a simple programming language. Then PXasm are used to define a self interpreter and the use of this self interpreter for formalizing execution of the earlier introduced TFSM is shown. Since the TFSM interpreter is the nucleus of the complete Montages semantics, this formalization is the basic building block of the formal semantics of Montages given later. On the example of the TFSM formalization we show how partial evaluation can be used to implement Montages by specializing its semantics. The given partial evaluator for ASMs is a further example for the use of PXasm.

Chapter 7: Attributed XASM

As mentioned, *attribute grammars (AGs)* are used in Montages for the specification of static semantics. In this chapter we propose a new variant of AGs, which combines features of object-oriented programming and traditional AGs to a new AG variant which features reference-values, attributes with parameters, and more liberal control-flow, e.g. no classification in synthesized and inherited attributes. The new variant is based on a further extension of XASM, called *attribute* XASM *(AXasm)*.

After motivating the design and initial examples for AXasm we give formal semantics to them in three different ways. First we show that AXasm can be translated easily into derived functions of XASM, then we extend the denotational semantics of XASM to the new features, and finally we give a self interpreter for AXasm. This self interpreter will be used in the Montages semantics to evaluate terms and rules.

Using AXasm, the complete specification of the object-oriented type system of the Java programming language is given in Appendix D. Although the example is relatively long, it shows that the approach scales to real-world languages. Finally we discuss related work in the field of AGs and specifications of Java.

Chapter 8: Semantics of Montages

Based on the previously proposed extensions of XASM, this chapter gives a formal semantics of Montages. We shortly discuss the choices for defining semantics of a meta-formalism like Montages. A parameterized, attributed XASM is then given, which processes, validates, and executes programs and Montages in 5 steps. First the abstract-syntax tree is generated, then the static semantics conditions are checked for each node. If all conditions are fulfilled, the states and transitions of the Montages are used to construct a TFSM which gives the dynamic semantics. Finally the TFSM is simplified, and then executed.

Part III: Programming Language Concepts

In this part we use Montages to specify programming language concepts. We try to isolate each concept in a minimal example language. The executability of each of these languages is tested carefully using the Gem-Mex tool, and we invite the reader to use the prepared examples and the tool to get familiar with the methodology. The standard Gem-Mex distribution contains the examples and is available at `www.xasm.org`.

The language *ExpV1* (Chapter 9, Models of Expressions) is a simple expression language. The remaining example languages are extensions of *ExpV1*. The first imperative language *ImpV1* extends *ExpV1* by introducing the concept of statements, blocks of sequential statements and conditional control flow. The concept of global variables is introduced in example language *ImpV2*.

The purpose of languages *ImpV1* and *ImpV2* is to introduced features of a simple imperative language. In a series of refinements, the primitive variable model of *ImpV2* is now further developed into *ImpV3*, and finally *ObjV1*. Language *ObjV2* is an extension of *ObjV1* with classes and dynamically bound instance fields, and *ObjV3* is an extension of *ObjV1* with recursive procedure calls. The languages *FraV1* , *FraV2* , and *FraV3* feature iterative constructs, exception handling, and a refined model of procedure calls, respectively.

The presented example languages are an extract from a specification of sequential Java. The Java specification mainly differs from the here presented languages by a complex object-oriented type system, many exceptions and special cases, and a number of syntax problems. We have given the specification of the complete Java type system as example in Appendix D. Unfortunately the scope of this thesis does not allow the inclusion of a full description of Java and we refer the reader to the description given by Schulte, Börger, Schmidt, and Stärk. In Appendix C we show how their model can be directly mapped into Montages. Other complete descriptions of object-oriented programming languages which can be mapped into Montages without major modifications are the specification of Oberon by the author and Pierantonio, and the specification of Smalltalk by Mlotkowski.

Part I

Engineering of Computer Languages

2

Requirements for Language Engineering

Information hiding (175) is a root principle motivating most of the mechanisms and patterns in programming and design that provide flexibility and protection from variations (142)[1]. One of the most advanced tools for information hiding is a *programming language*. A *general purpose language (GPL)* hides the details of how machine code is generated from more abstract descriptions of general algorithms. More hiding can be achieved by a *domain-specific language (DSL)* (217), which allows one to use a domain's specialized terminology to describe domain problems, and which allows one to hide the general programming techniques used to implement these problems efficiently.

The process of designing, implementing, and using a new, specialized computer language is often considered as part of the history of computer science. In contrast, the *DSL approach* aims at creating a repeatable software engineering process supporting information hiding by means of creating new languages. Most existing techniques supporting this process are too complex to be applied for people outside the software and hardware area. An important part of the DSL approach is therefore *computer language engineering*, the discipline of designing and implementing computer languages as tools for the software, hardware, and - most importantly - business engineers. The purpose of this thesis is to propose a simple, integrated approach, especially suitable for business related problem domains, such as finance, commerce, and consulting.

In this chapter we analyze the requirements for a *language description formalism* which can be used to reengineer the designs of existing, well-known GPLs, and to reuse those designs as basis for engineering new DSLs. The

[1] Data encapsulation, which is often used as synonym of information hiding, is only one of many mechanisms to support information hiding, other well known mechanisms are interfaces, polymorphism, indirection, and standards.

main issues with design and use of specialized computer languages are analyzed in Section 2.1. A typical application scenario for a DSL is presented in Section 2.2. The design process of DSLs is further analyzed in Section 2.3. Later in Section 2.4 we discuss the impact of reusing existing language designs, in Section 2.5 we sketch how part of the safety, progress, and security requirements of a system can be guaranteed on the language level, and in Section 2.6 is is shown how language description frameworks can be used to simplify language implementation. Finally in Section 2.7 the requirements for a language description formalism are formulated.

2.1 Problem Statement

The DSL approach exploits the design, implementation, and use of a new language, which is tailored for the needs of the domain at hand. Restriction to fewer, specialized features is considered as an advantage, since it allows one to hide more internal information. DSLs increase productivity not only through information hiding, but also by providing better scope for software reuse, possibilities for automatic verification, and their ability to support programming by a broader range of domain-experts. Often all these advantages are however shadowed by the cost incurred in designing and implementing a new language. Additionally, DSLs also involve relatively high maintenance costs since knowledge about the underlying domain usually grows with experience, and changing requirements lead to frequent revisions in the language. Resolving these issues is critical in making the DSL approach feasible, since otherwise it amounts to shifting the entire complexity of program development into the implementation and maintenance of the DSL. The situation is further aggravated by the fact that many small DSLs have an extremely limited number of potential users, sometimes also a brief life-span, and therefore do not justify too much effort from outside the group using the language. Designing a DSL is an important problem in itself and is a topic of research. However, it is not difficult to imagine a scenario where this problem is subsumed by the complexity in its implementation, and even more by maintenance, which involves specialized skills (in compiler technology, for example) usually not available with the domain experts who use the language.

From this situation we identify two main problems, which hinder the wide use of the DSL approach despite a long list of successful examples in the literature. The problems which faces every new DSL can be formulated as follows:

1. Users of the DSL are not familiar with the design of the new language.
2. Designers of the new language are often not experienced with techniques for implementing a new language.

The first problem hinders the use of a new language, while the second prevents successful implementation of the new language. A systematic approach for the solution of these two problems could be provided by a language engineering method which allows for

- a library of major existing language designs,
- the definition of new languages by reusing the design library, and
- the generation of language implementations directly from the language definitions.

Providing a *library of existing language designs* contributes to the solution of both problems, users can see existing language designs, which they already know and understand the language description style used, and designers can start with working descriptions and learn the description style by example. *Reusing the design of existing languages* not only simplifies the job of the designer but also helps the user to quickly understand the new language based on the reused existing one. Finally, if *implementations can be generated from the descriptions*, the problem of implementing the language can be reduced to the problem of defining the language in the given language definition style.

In order to follow this approach, the form of language definitions is of utmost importance. Most existing language definition formalisms are based on declarative techniques and are most suited to define languages with a declarative flavor. Since we understand the use of a DSL as a mechanism for information hiding of complex imperative and object oriented software systems, we need a language description formalism that allows one to map DSL programs directly into imperative, state-based algorithms. Those domain experts which are currently solving successfully domain-problems using main stream imperative and object oriented languages should be able to transfer their programming knowledge and experience directly into the design of a DSL. The DSL is thus a means to reuse their experience in a way where low level details about the implementation are hidden from the user and where implementation knowledge is moved into the language definition.

Our view on DSLs is in stark contrast to most of the existing DSL literature, which focuses on static, declarative DSLs. The most interesting paper which compares a declarative with an algorithmic DSL for the same application domain is the paper of Ladd and Ramming (137). They show how in an industrial context the development of software for telecommunication switches has been moved from C to an algorithmic, imperative DSL, and then further to a declarative DSL. Their case study shows clearly the advantages of the later declarative solution over the imperative one. One possible objection to their argumentation is that it may have been possible to define a more abstract imperative DSL, which would have shared most of the properties of the declarative language. Further at several places they are assuming that imperative, algorithmic languages are automatically ”general purpose” or ”Turing-complete” and further they take for granted, that an algorithmic, imperative DSL cannot be used as starting point to generate different software artifacts or to do analysis. Typically, algorithmic, imperative DSLs have reduced expressivity with respect to general-purpose languages, they have often an elaborated declarative static semantics, and besides the intuitive execution there are typically other things one can generate from them.

Even for clearly declarative DSLs, an additional dynamic semantics can be useful. If the declarative DSL specifies some sort of computation, it may be useful to add a dynamic execution semantics which is only used as an intuitive example of a possible execution behavior. Such a dynamic semantics would be given just for the purpose of delivering to the DSL user a state-based intuition. Another situation where adding imperative or object oriented features to a declarative DSL may make sense is *scripting*. If scripting is needed, it may be useful to extend a declarative core language with algorithmic features for scripting. This integrated scripting will certainly lead to simpler semantics than combining the declarative language with some general purpose scripting language. Since there are not many algorithmic DSLs described in the literature, we sketch in the next section a typical application scenario.

2.2 Typical Application Scenario

The most beneficial applications for DSLs exist, when two different groups of people must influence the behavior of a system. In such cases there is no clear separation between developers and users of a system. For instance, in the finance industry it is very common that both IT-experts and domain experts code part of the application. More complex IT tasks are solved by a high-level team of computer scientists, providing for instance a sophisticated data-base architecture and methods how to manipulate the data-base in a consistent way. The financial domain experts apply so called “office tools” like “Excel” or “Access” to “program” their own small applications on top of that infrastructure. This process is called “end user programming” (105).

The problem with using “Excel” or “Access”‘ for end user programming is their unrestricted expressiveness. The user is for instance not prevented from doing domain specific errors like calculating the sum of revenue and earning of a company in her/his spreadsheet calculations. A small DSL, allowing only for programming with domain specific, restricted expressiveness could make the process less error prone. We are convinced that a large part of the knowledge built into complex financial application suites could be leveraged into the semantics of a financial DSL.

As a concrete example we look at *trading strategies*. In today’s financial markets it is more and more common to use systematic trading strategies rather than buying and selling financial products in a non systematic, intuitive way. Because of their algorithmic nature, trading strategies are good candidates for automatization. The presented case study is based on an actual need of brokerage departments in large Swiss banks to automatize trading strategies.

In Section 2.2.1 we describe why automating trading strategies is important in the brokerage department of a bank, in Section 2.2.2 we analyze the problems using traditional GPLs or office tools. In Section 2.2.3 we show why using a DSL for their automatization is better than using a traditional PL and in Section 2.2.4 we conclude that DSLs are especially appropriate for the financial sector, since the requirements are changing very fast in this industry (53; 215; 109).

2.2.1 Situation

In a large bank, almost all transactions are finally executed by the brokerage department. The traders try to optimize their actions using systematic *trading strategies*. Three examples are given here.

- The traders must execute large amounts of orders generated by various other departments. Certain techniques can be applied to predict the development of the price of a financial product for the next few minutes and based on this assumptions the brokerage department may optimize its role as a buffer between the orders flowing in from other departments and the real market.

- For certain financial instruments, the bank is a *market-maker*, constantly offering to buy at a certain price, the *bid price*, and to sell at a slightly higher price, the *ask price*. If there are more sellers than buyers, the market-maker is lowering the prices until the market balance is reestablished. In the case of more buyers than sellers, the prices are increased. This process is called *spread trading*.

- It may be possible that a large client wants to execute a systematic, repetitive pattern of trades. This may serve, for instance, to hedge the client's risks resulting from other, non liquid investments.

A number of systems are supporting the traders in this activities, but because of the volatile requirements many tasks have to be executed by hand. The factors which constantly change the requirements are regulations coming from outside, internal management decisions, competition, and specific requirements from clients. If in the current situation some repetitive tasks are identified, the brokerage department may specify an application which helps them automating those tasks. The IT department is subsequently trying to implement the software according to the specification. In a large bank, the production cycle from the specification to the working software takes typically about three months. In this time, both security and usefulness of the new application are tested, and possible technical problems are identified and solved. After the production cycle, the software can be used by the traders.

This process may be too slow for the problem to be solved. Thus in many cases, the brokerage department will prefer to continue executing the tasks without automation. Since the costs for brokerage work-force are very high, and since even highly-trained experts tend to make more errors if they do repetitive tasks, the bank may lose money. Alternatively, the domain experts develop their own application using an office tool like “Excel”. Experience shows that such an ad-hoc solution is creating often more problems than it solves (39; 23).

2.2.2 Problem

It is relatively easy to write a program implementing the described trading strategies. The problem is not the coding of the algorithm, but the fact that the production cycle of three months makes the strategy to be implemented often obsolete. If we analyze what happens in those three months to a trading strategy software, we find a number of necessary activities which cannot be skipped.

- It must be tested whether the software correctly implements the strategy defined by the trader. An informal specification is always a source of misunderstandings. Often some information is lost between the know-how of the trader, and the implementation done by IT specialists.
- The software must be checked to behave always in a friendly way, not trying to use too many system resources, or to enter trades which would result in non-controllable situations.
- The risk-monitoring system of the bank must be used in a proper way. If a certain situation leads to an exposure which is pulling the trigger of the risk

measures, the software must stop executing the trading strategy and a rescue scenario must be triggered.

- The internal regulations determine which authorizations are needed for certain trades. In some situations, the software must thus interact with the traders to get digital signatures for the authorization.

If several trading strategies are implemented, many problems have to be solved repeatedly. Each resulting application has to pass the production cycle. If a general problem in the trading-strategy domain is detected, this problem can only be solved for the currently developed application. Older trading-strategy applications which may have the same error cannot be easily adapted and often the faulty behavior will show up in several applications.

Since the initial requirements are often ambiguous, and since problems with the application are most often fixed on the code level, the applications are often no longer consistent with their documentation at the end of the process. In a competitive environment, there will as well be no time to document the application properly. There is thus a danger that the resulting applications are not well specified, and cannot be maintained over a longer time period.

2.2.3 DSL Solution

A possible solution to this problem is to design a DSL for the specification of trading strategies. We call this DSL *TradeLan*. The elements of TradeLan are actions to enter, buy, and sell orders in the system, to “hit” orders being listed in the system, and to evaluate various indicators (including bid and ask price of the financial instrument to be traded as well as responses from the risk monitoring tool) as basis to decide when and how to execute certain actions.

Using the DSL approach, it is possible to tailor TradeLan such that

- only well-behaving trading strategies can be specified,
- the risk-monitor-system is automatically used in an intelligent way for any specified strategy; strategies which are not implementing the risk regulations cannot be defined,
- authorization checks are executed where necessary; there is no way to turn this feature on or off.

The specific problems for trading strategies are thus solved generically for all strategies written with TradeLan. The TradeLan programmer does not need to think how to solve these problems; she/he may concentrate on what the trading strategy is intended to do. The implementation of TradeLan adds all other necessary actions.

The implementation of this DSL will go through the three month production cycle. Probably it will even take some time longer since a DSL application is more complex than a simple trading strategy application. After the implementation went through the production cycle, the traders are faced with a completely new IT situation.

- A trader can now specify her/his trading strategy using TradeLan. For a programmer, writing a TradeLan specification will not look much simpler, but for the trader, a TradeLan program looks like an informal description of his ideas using *trader terminology*.

- From such a specification the implementation is generated, and the trader can immediately see whether the application is doing what she/he wants. Most importantly, additional trading strategies do not have to pass the production cycle any more. They can be implemented using TradeLan, and TradeLan specifications are just input to the TradeLan implementation which passed the production cycle already.

- Another advantage is that the people who defined the trading strategies can maintain them on their own. The TradeLan specifications look like informal specification documents, and they can be managed like other documents. Since they are understandable by the traders, they serve as documentation of the trading knowledge built up in the bank.

These advantages are offset by the typically high costs of designing, implementing, maintaining, and introducing a new DSL, if a suitable approach for engineering such languages is not available.

2.2.4 Conclusions and Related Applications

Time to market is the most important factor in the financial industry (63). If a new business opportunity is found, a quick implementation of the corresponding IT solution decides over the commercial success. However, the financial risks with each transaction imply that software must be deployed carefully (195). The above described solution shows that it is a good idea to generate the applications from explicit descriptions of the business rules, rather than implementing each repetitive problem by hand. Main reasons are the long production cycles and the problem that a lot of domain-knowledge is lost at banks, since the traditional applications do not force the user to keep specifications consistent. Knowledge flows into application source code, from where it can only be retrieved with difficulty. For these reasons we expect that DSL techniques will establish themselves faster in the finance industry than in other more static business domains.

An application area related to trading strategies is the specification of *financial instruments* or contracts. The problem of defining contracts is becoming increasingly acute as the number and complexity of instruments grows (118). Probably the first publicly known implementation of a financial product specification has been created by JP Morgan in the context of their Kapital system (179), which was the first environment where the DeAddio's and Kramer's *Bomium* architecture (53) for specifying complex financial instruments has been applied. During his research Van Deursen has introduced *Rislan* (214; 215), a formal and exact language for specifying financial contracts. This language has subsequently been used by CapGemini in their Financial Product System software (218). Later the company LexiFi Technologies has introduced *mlFi* (109;

62), a similar language which has been initially formulated as DSL. Such languages not only enable traders to be more precise in constructing deals, but such a contract definition can provide the basis for valuing contracts, as well as automating and managing their processing trough the transaction live-cycle.

The trading strategy application described in this chapter would become even more interesting, if trading strategies could be defined not only over a fixed set of existing financial contracts, but over freely defined types of contracts, using a language such as Rislan or mlFi for the contract specification. The static information of a financial contract specification could then be used as parameter for the dynamic semantics of a trading strategy language like TradeLan.

Another promising area for applying DSLs in finance is the tailoring of research articles to specific market and client situation. The company A4M (135) has used Montages to develop for a small financial service provider a technology where three specially tailored DSLs are used to generate research reports for complex structured financial instruments. The first DSL, called *InstruLan* is used to describe the structure and semantics of the analyzed financial instruments, the second one, called *IndiLan* is used to define the calculation and naming of financial indicators derived from the available data, and the third language, called *FinTex* is used to give text fragments as well as the logic how to compose them to full-blown, natural-language financial-analysis texts, which may be personalized for specific clients, interest groups, risk profiles, e.t.c.

In contrast to Rislan and and especially mlFi, A4M's *InstruLan* is not a fixed, full blown language for specifying all kind of contracts, but *InstruLan* is a minimal language adapted to the clients existing set of products and terminology. Experience with using InstruLan for a large international bank in Zürich shows that in practice a family of minimal DSLs for specifying financial products, adapted to the needs of different clients may serve them better than a one-fits-all solution. On the other hand, an industry-proven product specification approach such as Bomium is a perfect basis to explore new types of financial instruments.

2.3 Designing Domain Specific Languages

Early in the history of programming language design, the idea arose that small languages, tailored towards the specific needs of a particular domain, can significantly ease building software systems for that domain (24). If a domain is rich enough and program tasks within the domain are common enough, a language supporting the primitive concepts of the domain is called for, and such a language may allow a description of few lines to replace many thousand lines of code in other languages (94). A good starting point for designing a domain-specific language (DSL) is a *program-family* (176). This idea is elaborated in the FAST (227) [2] process for designing and implementing DSLs.

Central to FAST is the process to identify a suitable family of problems, and to find abstractions common to all family members. Traditional software development methods would use the knowledge about a family of problems and common abstractions as well, but in a more informal way. In FAST, as well as other DSL processes, one tries to use these abstractions to produce implementations of family members in an orthogonal way. Rather than crafting an implementation for each problem at hand, one designs an implementation pattern for each abstraction, in such a way that implementations of single problems can be obtained by composing the patterns. Typically such implementation patterns are therefore developed with a GPL supporting generic programming in some way. To this point, FAST is very similar to most reuse methodologies.

We visualize the situation as follows. In Figure 1 the problem family contains members *m1*, *m2*, and *m3*. The common abstractions *a1*, *a2*, and *a3* are depicted as shapes, which occur repeatedly in the family members. The implementation patterns *i1*, *i2*, and *i3* are then developed for for each abstraction. In Figure 2 the process to construct an implementation is represented by the triangle. The input to the process is a member of the problem family and the implementation patterns of the abstractions. The output is an implementation solving the problem.

In the next step of the FAST process a language is designed for specifying family members. The syntax of the language is based on the terminology already used by the domain experts, and the semantics is developed in tight collaboration with them. The goals are to bring the domain experts into the production loop, to respond rapidly to changes in the requirements, to separate the concerns of requirement determination from design and coding, and finally to rapidly generate deliverable code and documentation.

The design process of the language consists of introducing syntax for denoting the abstractions we identified in the first step and defining the allowed constructions of complete sentences in the new language. This definition should capture the knowledge gained from the implementation patterns and exclude all non-correct combination of the abstractions. The possibility to define exactly in which way the syntax and the semantics of the language allow us to combine the basic abstraction is the big advantage over traditional ways of reusing

[2] Family oriented Astraction Specification and Translation

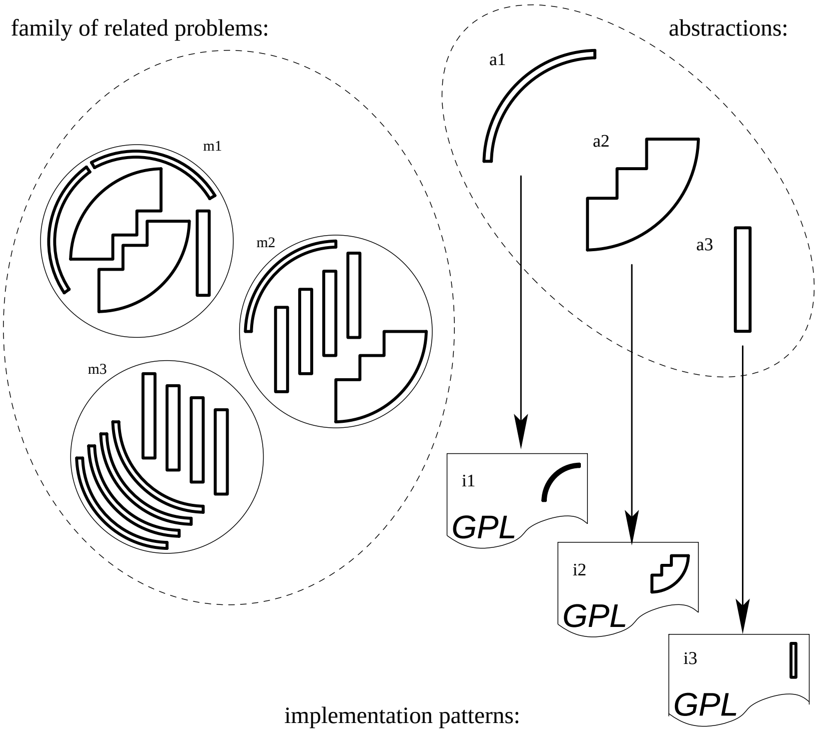


Fig. 1: Identification of problem domain and abstractions

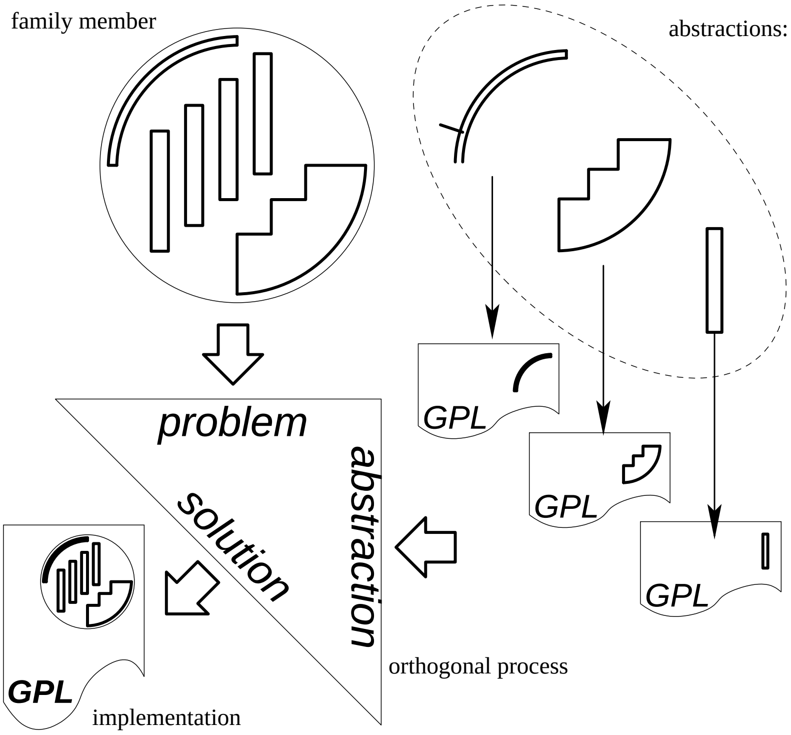


Fig. 2: Orthogonal process for implementing family members

abstractions, such as libraries or component frameworks. In none of the later two the user can be forced to use an abstraction in the right way: either the user is allowed to use the function or component, or not. By means of a language, the complete context of applying an abstraction is known, and the use of an abstraction can be allowed for certain contexts only.

As visualization, in Figure 3 we schematize a DSL definition and the relation of its syntactical productions *feature 1* …*feature 3* to the corresponding abstractions. The bottom left corner contains a number of DSL programs specifying the problem family members in the bottom right corner. The arrow from the problems to the abstractions and the one from the DSL definition to the DSL

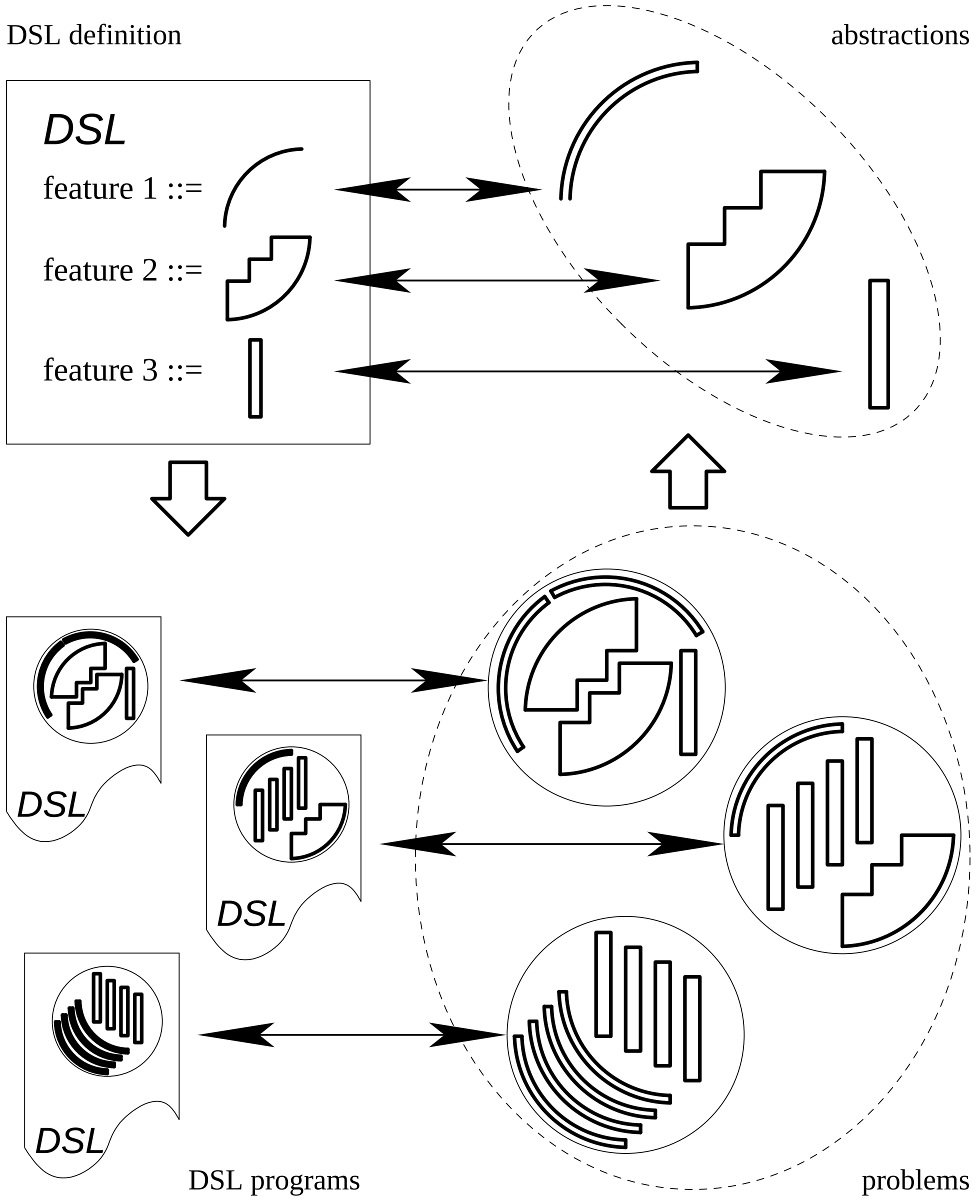


Fig. 3: Design of DSL for family member specification

programs depict the engineering process as it is described up to now: deriving abstractions from the problem domain, defining a DSL for specifying over such abstractions, and using the DSL to specify the problems in the domain.

We would like to note the difference between the GPL program resulting from the orthogonal process in Figure 2 and the DSL program in Figure 3. While both are related to the same problem, the GPL program is directly executable, while the compiler or interpreter of the newly designed DSL has to be implemented. In fact the implementation costs for a new DSL can be very high, if no specialized language implementation method is available.

This leads to the last step in the FAST process, the implementation of the DSL. One possibility is to use a meta-formalism to formally define syntax and semantics of the introduced DSL, and to generate the implementation from this definition. Alternatively traditional compiler or interpreter construction tools can be used.

The DSL implementation process is shown in Figure 4. This figure corresponds directly to Figure 2 but the informal description of the family members has been replaced by the formal DSL descriptions, and the implementation patterns have been combined with the specification of the DSL (production rules feature 1, feature 2, feature 3 on the right), resulting in a full specification of the DSL.

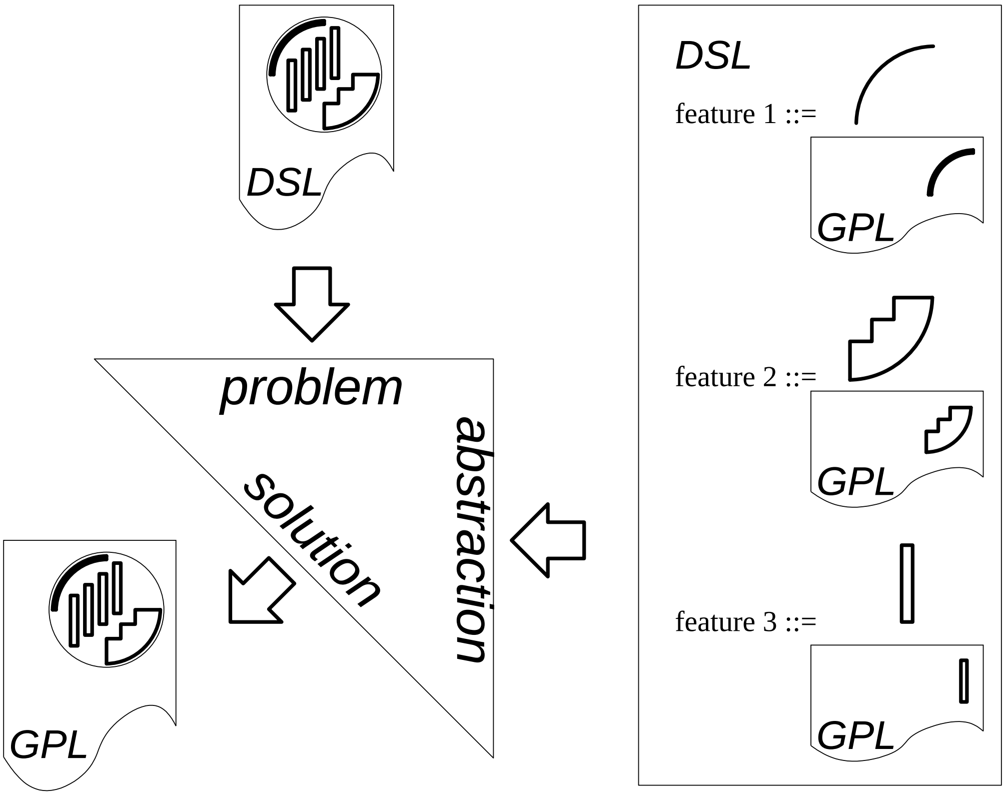


Fig. 4: Implementation of a DSL

2.4 Reusing Existing Language Designs

It is often a concern that the broad use of technologies for the introduction of DSLs would lead to a confusing number of different languages. The worst situation would be the coexistence of languages, where

- slightly different kind of syntax and semantics are used for features being functionally identical, and
- the same syntax is used for features being completely unrelated.

Most confusing gets the situation, when the exactly same task needs to be done in different languages, but the languages solve the task in different ways. For instance in the Centaur tool-set (35) the two DSLs for specifying pretty-printing and dynamic semantics processing of parse tree are providing different syntax for accessing the leaves of the tree, although both DSLs work on the same tree-representation in the Centaur-engine. Our experience with using the system (124) shows that such a situation has a negative impact on productivity.

Instead of designing new languages from scratch, as done in many existing DSL methodologies, we propose reusing designs of existing languages. This approach allows us to engineer the set of languages being used, rather than considering them as unrelated, incompatible entities. Our approach is to start with a library of existing, well-known language designs and to create new languages by applying the following four *language-design reuse patterns*:

- **restriction** Take an existing language and restrict its expressiveness. This can be done by removing features, or by fixing the possible choices for some features in a context dependent way.
- **extension** Add a new feature to an existing language by combining existing features under a new name, or by adding a new kind of semantics[3].
- **composition** The synthesis of a larger languages as a combination of small sublanguages. This pattern allows the designer to describe, test, and teach small subsets of language features, and combine them later to real-live languages.
- **refinement** Change the semantics of an existing construct. This is the most dangerous pattern. Typically it is applied in such a way, that the intuitive semantics remains the same for the user, but some details are adapted to a special situation.

If a language is designed based on existing well know languages there are more users which are familiar with part of the design, and a language description methodology which supports synthesis of new languages trough the actions of *restricting*, *extending*, *composing*, and *refining* existing descriptions simplifies

[3] Technically, the extension-pattern can be considered as a special case of the combination pattern. From the language user's point, they are very different, since the extension pattern involves only one existing language, while the combination pattern combines at least two different languages.

the task of the language designer to implement the language easily. Further, some of the advantages of DSLs as listed in Table 1, can be combined with the advantages of GPLs with respect to DSLs are listed in Table 2.

Tab. 1: Advantages of DSLs

Compactness	Features are focused on problems to be solved. Fewer concepts have to be learned to master the language. A larger group of people can use the language.
Abstractness	Since the specific application domain is known in advance, abstractions can be found, and many details can be hidden in those abstractions.
Self Documentation	Systematic use of the established terminology in the problem domain results in good self documentation.
Safety	Absence of a feature in a DSL guarantees its absence in all programs written with that DSL.
Progress	Transactions consisting of a number of actions can be encapsulated in the semantics of specific constructs.
Security	Correct authorization of each action can be guaranteed by the language definition.

Tab. 2: Advantages of GPLs

Stability	The language design has proven its consistency and will not change too much over time.
Existing Solutions	Many problems have been solved with the language. Not everything has to be done from scratch, and many examples of how to use the language exist.
Education	Many programmers know how to use the language and it is easy to find experienced developers.
Available Tools	Typically GPLs GPLs are supported by compilers, interpreters, debuggers, and other tools which are integrated in one, versatile development environment.

2.5 Safety, Progress, and Security

The systematic introduction of new languages as extensions, restrictions, compositions, or refinements of existing languages can be used to guarantee some of the *safety, progress,* and *security* requirements of a system. Following Szyperski and Gough (206) these properties can be defined as follows:

Safety Nothing bad happens.

Progress The right things do happen.

Security Things happen under proper authorization.

Using language design for guaranteeing some of these properties is a common technique (206). For GPLs only general properties like strong typing can be embedded. In a relatively narrow domain, many more requirements are known. Restricting, extending, and refining existing languages can be used to guarantee safety, progress, and security on the language level, rather than on the code level. The pattern to using language restriction for safety is already described (200) , but the idea to use language extension for progress, and language refinement for security have not been discussed earlier. As a disclaimer for the following discussion we would like to note that all of this problems can and are solved with traditional programming means as well. We try to highlight some advantages if the problems are solved on the language level rather than on the implementation level.

The first idea is to achieve *safety* by reducing expressivity of the programming language used for the critical components of a system. Reducing expressivity can be done by removing language-features, or by fixing the possible choices for some features in a context dependent way: for instance one could remove features to interact with external computers from pieces of code that serve for internal calculations only. In this way it is possible to guarantee safety conditions on the language level, allowing source code developers to concentrate on non-security-critical details. We call this technique *safety through reduced expressiveness*. An example is a safer subset of C presented in (64). Although reducing expressivity of languages is not a general solution to safety problems, a framework in which language features could be turned off individually would allow the developers to solve some safety problems. For instances computer viruses relying on certain language features could be stopped by allowing those features only in parts of the system which are completely write-protected from the network.

Security may be achieved by refining the semantics of an existing language feature such that correct authorization is guaranteed. As an example, consider a situation where a central security server has to be informed before each security critical call to a given library. This problem can be solved with a standard application programming interface (API) for the library. The problem with the API approach is, that changes in the library must be correctly reflected in the API, and each time a new function is added, there is the danger someone forgets to implement all API rules, such as the above mentioned rule that a security server has to be informed.

Our approach would be to change the programming language which is used such that the central security server is informed automatically, whenever the critical library is called. Like this it is guarantees that all authorization is done correctly, independent of how the application and the library are developed.

A typical example related to *progress* is a requirement that after opening a transaction, either all parts of the transaction are executed successfully, leading to a commit of the transaction, or a roll-back is triggered. Our idea is to guarantee this requirement by encapsulating the complete process in one new language construct. Of course such a construct has to be added to a language that has been restricted such that the transaction cannot be started otherwise. Another issue applying this approach could be performance problems.

Reuse of existing language designs and the subsequent restriction, extension, composition, and refinement of their definitions, both syntactically and semantically, are basic building blocks for a realistic application scenario for engineering of computer languages. An example for defining a DSL by first restricting to a subset and then extending with domain-specific features can be found in (20) where a protocol construction language is defined as extensions on top of a subset of C. We illustrated that achieving *safety*, *progress*, and *security* on the language level may be the conceptual motivations for introducing a DSL.

2.6 Splitting Development Cycles

From a high-level viewpoint every software development cycle can be presented as in Figure 5. A system is specified, a suitable architecture is designed, the software is implemented, tested against the specification, and finally brought into a form suitable for deployment. The platform for such a cycle is typically a GPL with its support tools, visualized in the figure as the innermost box labeled "platform". The result of going through the cycle is the creation of an application, which serves as the "platform" for the user to solve her or his daily problems. The user provides positive and negative feedback on the correctness, efficiency, general usefulness of the application. This feedback, along with additional requirements, triggers a new development cycle, resulting in a new version of the application.

Using a process like FAST the development of a system is split into two independent development cycles, as shown in Figure 6. In a first development cycle, a DSL is designed and implemented. The "application" resulting from this cycle is the DSL being used in the second cycle to specify and implement end-user applications. Users of the application provide feedback for the application-developers, and application-developers, who are also DSL-users, provide feedback to the DSL developers. This situation allows for an interesting split of maintenance tasks. Fine tuning and solution space exploration of the problem is done in the application development cycle working with the DSL, while improving performance and porting to other software and hardware architectures is typically done by refining the DSL definition. Similarly, reuse of algorithms happens on the level of DSL programs, while reuse of interfaces to underlying hardware and software architectures happens on the DSL-definition level.

The crucial software development problem in such projects is often the implementation of the DSL. This stems from the fact that in many cases the identified problem family is intricately structured, but each single family member is quite a simple problem. The implementation of such a family member can thus be relatively simple, compared to the costs for implementing the DSL. For a successful application of a DSL, the additional implementation costs for the DSL must be offset by the reduced costs of repeatedly using the DSL to solve problems of the problem family.

Methods that minimize the costs for design and implementation of DSLs increase considerably the number of useful and feasible DSL applications. Recently a lot of research was dealing with the problem how to minimize the costs for implementing a DSL (90; 21; 68; 163; 209; 200). The main idea behind most approaches is to define a *language definition formalism* which can be used to define the DSL, and to generate an implementation from such a definition. Having such a formalism and tool at hand, it is possible to split the development process into three development cycles, as shown in Figure 7.

While in the above described two cycle model a GPL is used in the development cycle of the DSL-definition, in the three cycle model, a language definition formalism (LDF) is used for the DSL-definition. The third development cycle,

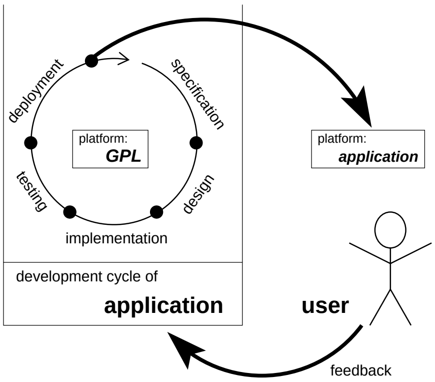


Fig. 5: Classic development cycle of applications

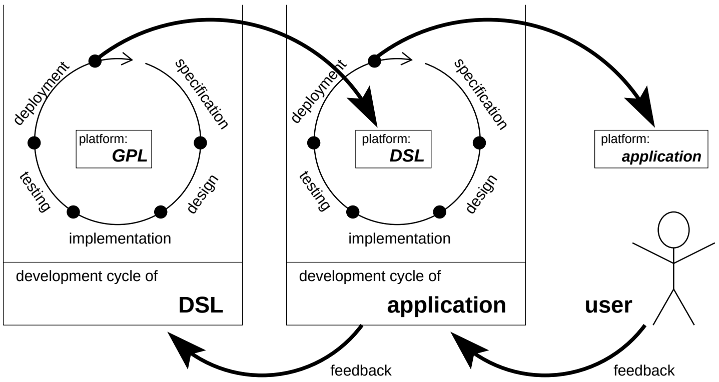


Fig. 6: Development cycles of DSL and application

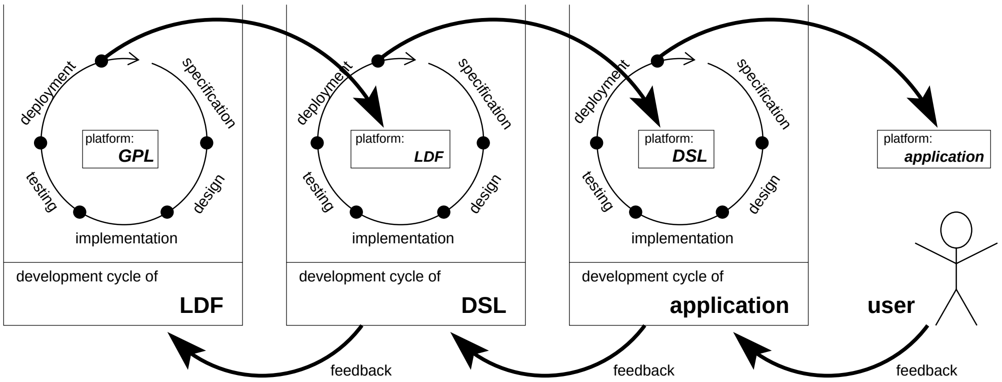


Fig. 7: Development cycles of Language Definition Formalism(LDF), DSL, and applications

shown on the left side of the graphic, is concerned with the development of the language definition formalism . The “application” generated by this cycle is the language development tool. The second cycle now uses the LDF as platform for the development of the DSL. Interfaces to existing hardware and software architectures as well as program generators for parser and other language technologies like attribute grammars are provided by the language definition formalism, allowing the DSL-designer to concentrate on efficiency, integration, and extensibility issues in the problem domain.

We hope that in this way the costs for DSL implementations can be split over many domains. However in the three-cycle model one has to consider the costs of learning the LDF as well as the costs for defining the DSL with the LDF. The sum of learning an LDF and implementing a DSL for the first domain may be larger than the costs to implement a DSL from scratch. Once the LDF method is learned, its application to new domains can be done with little costs. Restriction of the LDF to well known techniques such as EBNF, Attribute Grammars and Flow Charts avoids creating a new problem of understanding language definitions.

2.7 Requirements for a Language Description Formalism

In order to solve the stated problems, a language description formalism and the corresponding language design method should fulfill the following requirements.

- The techniques used for defining languages should be well known. The typical background of a programmer should be sufficient to understand the descriptions. EBNF and flow charts are typically the “specification tools” of a programmer.
- Languages should be described in a “compact” form. This is important since many users deal with large software projects and do not have the additional resources to create and maintain huge language descriptions. The size of a language specification should evolve linearly with the number of production rules in the grammar.
- A language description should be built with small, independent building blocks. Reusing the features of a language should involve a minimal interface with other components of the language. A mechanism for the modularization of language specifications is therefore needed.
- A library of specifications of major programming language concepts should be available. This library should cover both concepts for programming in the small, which can be reused to synthesize efficiently a DSL without reinventing details such as expressions, as well as concepts for programming in the large, which can be used to extend a DSL with state of the art modularization concepts, such as object orientedness. Most important, the modules of the library should have a high level of decoupling.
- Tool support should provide a comfortable development environment for the specified languages. Not only an interpreter or compiler should be generated from the specifications, but as well a number of support tools, such as debuggers, program animators, and source analysis tools.

2.8 Related Work

It may be correct to say that the concept of DSLs has not been invented but observed. One of the earliest references to DSLs is Landin (141). The large problem space to which software systems may be applied has caused a proliferation of such specialized languages. There has never been agreement on whether a multitude of different languages should be supported and managed by appropriate tools, or whether one should try to define languages like Ada or C++ which can be used to cover all problems.

One solution to combine advantages of specialized languages and general purpose languages is to provide programming languages which are extensible with domain-specific features. Research on extensible programming languages, as summarized by Standish (202), has led to insight both in techniques to allow for extensibility and problems related with extensibility. Extensibility as language feature has often led to more maintenance problems than it has solved. Altering the semantics of existing languages has been identified as especially harmful. Examples of successful extensible programming languages are *CLOSE* (119), an object-oriented Lisp language, and *Galaxy* (22), an efficient imperative language. In both cases, the extension features have been used to bootstrap the implementation of the languages.

The general problem of tailoring a programming language to the application domain forms part of language design research (230; 228; 96). With respect to the design of DSLs, the discussions about how to decide on feature inclusions are interesting. Knuth (121) argues that the inclusion and exclusion of features should be based upon observed usage in addition to theoretical principles. This idea has led to research on feature set usage analysis; a good summary can be found in the text of Weicker (226). The large amount of available material has even led to statistical investigations (196). The use of different DSLs with comparable definitions may lead to new applications of such work.

An interesting paper looking at the use of DSLs for software engineering is the work of Spinellis and Guruprasad (201). The paper investigates typical software engineering problems, which can be nicely solved by introducing a DSL and shows a list of representative examples. The most interesting example deals with the use of about 10 DSLs for the development of a CAD system in civil engineering (199). A software engineering discipline for which DSLs are especially well suited is *rapid application development.* Boehm notes that portions of certain application domains are sufficiently bounded and mature so that you can simply use a specialized language to define the information processing capability you want (26). He further highlights that individual users with relatively little programming expertise can, in hours or days, generate an application that once took several months to produce.

Looking at DSLs from a broader perspective, they are most naturally considered as part of *domain engineering* (165; 14; 166). The FAST process discussed earlier is an example for a domain engineering process focusing on DSL design. The method is based on previous work about program families (176).

FAST has been used by Weiss's group at Lucent and now at Avaya for over twenty different projects in software production. Experience reports and a detailed description of the approach can be found in (16; 227; 15; 50). A related approach being developed before FAST is the *Reuse-driven Software Process* (or Synthesis) approach by Campbell (37; 36). This approach has been adopted by many companies such as Rockwell International, Boeing, Lockheed-Martin, and Thomson-CSF.

The programming language C++ has turned out to be a good platform for the development of sophisticated domain-specific frameworks. Very often these frameworks are of generic nature. Recent work (48; 51) shows how DSLs can be used to make such frameworks accessible to domain experts, and how to combine DSL based processes like FAST with generic frameworks.

Another very promising approach is the *Sprint method* (210; 47). It follows the view that a DSL is a good parameterization for a domain-specific framework. Having efficient C++ frameworks at hand, using denotational-semantics for the language definition, one achieves both efficient implementations and nice formal semantics.

Combining generic frameworks with DSLs is further pursued in the *Jts* approach (21). This approach provides a set of tools which allow mainstream languages to be extended with domain-specific constructs. The implementation of existing language designs is directly reused and not generated from a language definition. The DSL technique is used only for new constructs. This approach is very realistic, since the description of existing languages and the generation of tools for this languages is very hard.

Methods based on established compiler construction tools like Coctail (77) and Eli (76) include full descriptions of existing languages and the generation of a state-of-the-art compiler. Since construction of an efficient compiler is a complex task, some of this complexity cannot be fully hidden, and the use of such tools is not very easy. In (180) the complexity of Eli is managed by allowing typical language features to be turned on and off, but this approach hides those details which would be needed to access the definitions of the existing languages. In general, all approaches for DSL implementation show that one has to make a trade off between ease of use and quality of the generated code.

Focusing on the support tools, rather than the actual language compiler or interpreter, the mid and the late-eighties saw a proliferation of different *programming environment generators*, some of the best known among them being the Synthesizer Generator (189), Centaur (35), Pan (19), Mentor (61), PSG (18), IPSEN (66), Pecan (188), Mjolner (147), Yggdrasil (38), GIPE (91) and ASDL (123). The current work on DSLs has renewed the interest in these frameworks. For example, the ASF+SDF Meta-Environment (120; 213) has been used to successfully implement several DSLs being used in the industry (214; 216). Other work is concerned with generation of tools from attribute grammar description of languages (93).

The flexibility associated with generating a language implementation from its specification results in significantly improving the ease in maintenance, which is important in the DSL context (216). In contrast to previous work on programming environment generators where the main focus was on the generation of a language-based editing system, current interests, however, are more related to issues like generating efficient compilers, interpreters, debuggers, and above all, ease in specification. Some of these tools can be generated only if the runtime behavior of a program is contained in the language description. As a result of this the specification of dynamic semantics has gained more importance than in the past.

While most existing applications in industry focus on small, declarative languages without dynamic semantics (44; 45), the abstract specification of dynamic semantics is an important topic of *formal programming languages semantics*, such as Denotational Semantics (192), Structural Operational Semantics (182), or Natural Semantics (110). Applying programming language semantics tools allows for high level specification of languages. A discussion on existing approaches for language definition formalisms tailored towards DSLs is presented by Heering and Klint (92). The main problem with applying programming language semantics approaches for DSLs is that they take advantage of a number of mathematical techniques like rewriting systems, algebraic specifications, or category theory which are not known to a typical computer-science engineer, let alone to the different kinds of domain engineers. Schmidt calls for a “popular semantics” (191) combining the formality of existing approaches with ease of use. Unfortunately many practical approaches cannot satisfy Schmidt’s requirements for a “popular semantics” since they are not based on a calculus allowing directly for correctness proofs. Among the classical programming language semantics approaches the Action Semantics (158) approach has been specially tailored for combining a traditional language semantics style with ease of use. The problem of modularity with respect to language descriptions has been investigated by Mosses and Doh (159; 160; 60).

Besides the use of many mathematical concepts, another source of complexity in classical programming language semantics approaches is their common property to consider each parse tree as a syntactic entity. Two equivalent subtrees are represented as the same entity, and it is not possible to decorate the parse tree with attributes or intermediate results, and control/data-flow graphs must be encoded with tables or continuations. In newer approaches like (70; 80; 167; 183) each parse tree is formalized as a tree of objects, which can be decorated with attribute values, intermediate results, and direct links to other objects, representing the control/data flow edges. Poetzsch-Heffter defines *occurrence algebras* (186) which allow to combine the newer approaches with traditional techniques.

Since one of the main problems with DSLs is language implementation costs, different implementation patterns have been investigated by Spinellis (200). He discusses both the language extension and the language restriction, or specialization pattern. The importance of language specialization for safety has been recognized clearly by him, but the relation of progress to language extensions is not discussed, since the focus of the paper is on language implementation rather than language design. We also propose to add a language refinement pattern for security. The language composition pattern, which we use repeatedly, is not mentioned in (200) since language combination is not possible with most existing language implementation techniques. At this point it is important to note that our composition notion is only informal and based on empiric results from a certain class of applications. An example for a state-based framework providing formal compositionality are *Especs* (177).

In this text we are not focusing on the problem of how to describe the syntax of a language, but in practical applications of DSL design, the definition of syntax is the first, and thus most critical task. Many successful DSL applications show very simple, sometimes line based syntax styles. Another approach for avoiding syntax problems is to use XML for the representation of programs. Cleaveland discusses different DSL scenarios with XML-syntax and explains them carefully (45). An earlier, related approach are Lucent's *Jargons* (163; 107; 161), and their support tool *InfoWiz*. InfoWiz is the major language implementation tool used in the FAST approach. Jargons build a family of DSLs with similar syntax on top of a host language called *FIT* (162). The variable part of a jargon is declared with *WizTalk*, a meta-language similar to XML.

For the reuse of existing GPL designs including the original syntax, a full scale parser generator such as Lex/Yacc (143; 104) is needed. Already in 1988 the parser generator TXL (49) was proposed for the definition of dialects of existing languages. In general the syntax problem is much harder if existing languages should be reused. According to Jones at least 500 programming languages and dialects are available in commercial form or in the public domain (106). Lämmel and Verhoef propose a sophisticated methodology to efficiently derive parsers by reusing existing grammars (138; 139). The syntax problem is very hard, and at the same time very well investigated. We are therefore referring to the literature and concentrate mostly on semantics.

Our treatment of characteristic and synonym productions allows an automatic generation of an abstract syntax tree (AST) from the concrete EBNF-syntax, as defined by Odersky (167). This choice is on one hand restricting the application of the current implementation to real-live programming languages with simplified syntax only, but on the other hand it simplified both the implementation of the tool, and the specification work with the tool. If we would have chosen a full fledged solution with completely independent treatment of concrete and abstract syntax, as featured by most of the mentioned attribute grammar and formal semantics systems, we would not have been able to design, implement, test, and validate a new programming language prototyping environment from scratch.

One of the most successful language specification technique, Attribute Grammars (122) is not discussed in detail here, but later in the related work Sections 3.5 and 7.3. At this point we would like to mention only the work of Mernik et al. on reusable and extendable language specifications (153; 154). The authors discuss how to use object-oriented programming features to allow for incremental programming language development. Adding such features to a specification environment is a very useful step, and the usability of many approaches, including the later introduced Montages approach, would benefit from such features.

3

Montages

In the Chapter 2 we analyzed specific requirements for a language description formalism. These requirements have been used as design principles for Montages, a meta-formalism for the specification of syntax, static analysis, static semantics, and dynamic semantics of programming languages.

- An introduction to Montages is given in Section 3.1.
- After a short description of syntax related aspects in Section 3.2,
- in Section 3.3 it is shown how Montages define dynamic semantics by making the syntax trees directly executable. To formalize executable trees, we introduce the concept of *Tree Finite State Machines* (TFSM).
- The details of Montages related to lists, and non-local control flow are explained in Section 3.4.
- Finally in Section 3.5 related approaches are discussed and the results of Montages related work are reviewed.

3.1 Introduction

New languages are defined passing through a number of stages, from initial design to routine use by programmers, forming the so–called *programming language life cycle*. During this process, designers need to keep track of already taken decisions and the design intentions must be conveyed to the implementors, and in turn to the users. Therefore, as for other software artifacts, accurate, consistent and intellectually manageable descriptions are needed. So far, the most comprehensive description of a programming language is likely its reference manual, which is mainly informal and open to misinterpretation. Formal approaches are therefore sought.

Montages is a new proposal for such a formal approach, which can be seen as a combination of EBNF, Attribute Grammars, Finite State Machines and a simple imperative prototyping language called XASM. All of these techniques except XASM are in some form part of the typical university curriculum of a programmer and we hope that the resulting descriptions are thus easy to understand by language designers, compiler constructors, programmers, as well as domain engineers.

One of the main achievements of Montages is a new way to modularize the design of languages. Our library of existing language designs contains small specification modules, each of them capturing a language feature, such as scoping, sub-typing, or recursive method calls. In the current state, the library contains all features needed to assemble a modern object-oriented language such as Java. Most interestingly we managed to achieve a high level of decoupling among the modules. For instance we can treat exception handling independently from method calls or break/continue semantics. The library of language features is shown in part II of this thesis.

Figure 8 illustrates the relationships between language specification and language instances, e.g. programs. On the left-hand side the syntax and semantics related components of a language specification are shown, and on the right-hand-side, the corresponding process on language instances is shown.

Syntax

Syntax of a programming language is specified by means of EBNF productions. The EBNF productions define a context free grammar (42), and can be used to generate a parser. In Section 3.2 we specify the exact kind of syntax rules, as well as a canonical construction of compact abstract syntax trees (AST). The corresponding phase 1 of Figure 8 refers to the transformation of programs into ASTs.

Static Semantics

Static Semantics of programming languages is described by means of attribute grammars (122) and predicate logic. All static information, such as static typing, constant propagation, or scope resolution can be specified with attribution rules. The resulting attribute values of the AST are both used during dynamic semantics, and for the evaluation of the static semantics condition of each construct. In phase 2 the attribution rules are evaluated transforming the AST into

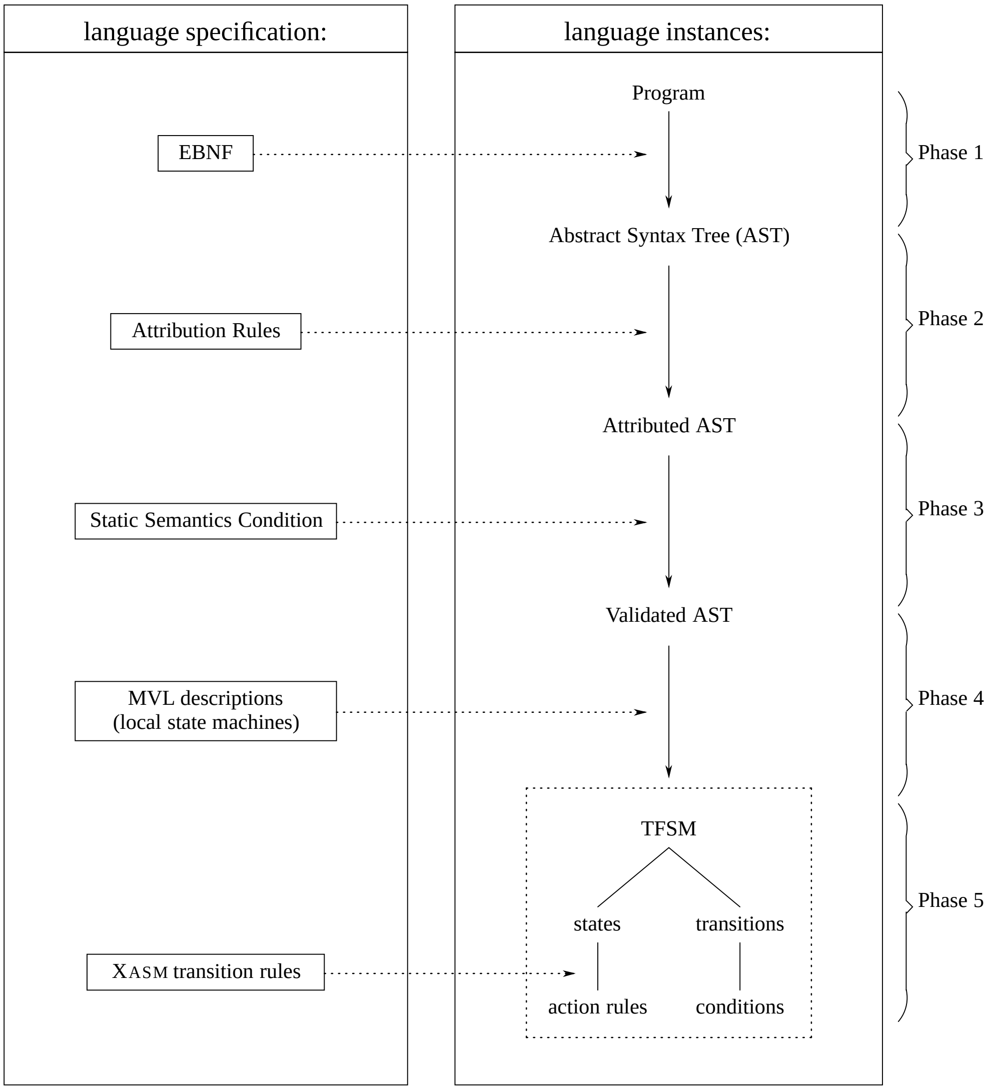


Fig. 8: Relationship between language specification and instances.

an attributed AST. The static semantics is given by means of predicates associated with the EBNF productions, so called *static semantics conditions*. Only if the static semantics condition of each node in the AST evaluates to true, the program is considered valid, otherwise it is rejected and not considered as a valid program of the specified language. In phase 3 the static semantics conditions are checked in order to validate the AST. Since attribute grammars and predicate logic are well-known formalisms, we do not explain them further in this chapter. The exact type of attribute grammars used by Montages is described formally in Section 7 and the formal description of static semantics definitions are deferred to Section 8.3.

Dynamic Semantics

Dynamic semantics defines the *execution behavior* of a program. Montages gives dynamic semantics by mapping each program of a described language into a finite state machine, whose states are decorated with actions which are fired, each time a state is visited. With other words, during execution control flows along transitions whose firing conditions evaluate to true, and at every state visited, the corresponding action rule is executed.

Instead of giving a transformation from programs into state machines, we introduce a novel kind of state machines, called *Tree Finite State Machines*(TFSMs) (phase 4 of Figure 8). TFSMs are derived from an XML based DSL formalism developed by the author (126). By means of TFSM we can directly execute an AST, without transforming it into another structure. The execution behavior of the program is then given by executing the TFSM (phase 5 of Figure 8). In short, the TFSM semantics of an AST is defined by giving a local state machine for each EBNF production rule. The local state machines and their embedding into the TFSM are given by means of *Montages Visual Language* (MVL). MVL allows to define control flow both inside a local state machine, and between machines associated with different productions, both those of the symbols denoting siblings in the AST[1] and those of arbitrary symbols[2]. Entry and exit points of a MVL machine are marked by the special states ”I” (initial) and ”T” (terminal). Execution of a program starts by visiting the ”I”-state of the AST’s root, and stops either by reaching the ”T”-state of the AST’s root or by being terminated by the action rules. Many interesting programs are not terminating at all. The introduction to TFSMs and their specification by means of MVL are given in Section 3.3.

Vertical Structuring

Unlike most other language description formalisms, in Montages the phases are not used to structure the specification horizontally in modules. Instead, for each production rule of the grammar a specification module, called a “Montage”[3] is given, containing The EBNF-definition, the attributions, the static semantics

[1] This corresponds to so called “structural” control flow into the sub-components of a language construct.

[2] This corresponds to more liberal ways of control flow such as goto-constructs.

[3] Montage: The process or technique of producing a composite whole by combining several different pictures, pieces of music, or other elements, so that they blend with or into one another.

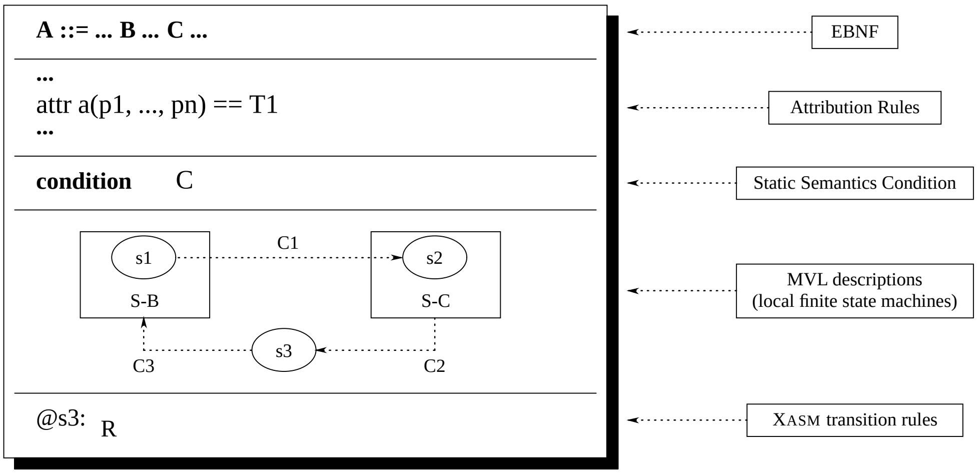


Fig. 9: An abstract Montages example

conditions, and the MVL-machine. Each Montage describes like this the semantics of a production rule, and can be considered in some sense a “BNF extension to semantics”(192; 191). A language definition consists of a set of Montages.

Examples

As an abstract example of a Montage containing all five parts take Figure 9. The first part contains an EBNF rule defining the context-free syntax, here a syntactic component A contains among others components B and C. The second part is the attribution rules. Here an attribute a with parameters $p1, \ldots, pn$ is defined by term $T1$. The third part, the static semantics condition is the predicate C. In the fourth part we see a first example for MVL. It is an abstract example, containing references to the B and C components, states $s1$ of the B-component, state $s2$ of the C-component, and state $s3$ of the A-Montage itself, as well as transitions with firing conditions $C1$, $C2$, and $C3$. It is missing the specification of the entry point ”I” and the exit point ”T”. The fifth part is the action rule R associated with state $s3$.

A more intuitive example of a Montage containing ”I” and ”T” states is given in Figure 10. A while statement is specified, being different from a typical while by having a special action rule *profile* which is used to count how often a program loops. In fact, it is a global counter that counts iterations of all loops. The example is chosen since the state and action for *profile* makes the example more interesting, but also to show how a well known language construct can be slightly altered, for instance in order to support program profiling. The syntax of the while-construct is well known from typical imperative programming languages, such as Algol (164) or Pascal (231). The syntactic components are an expression, and a list of statements. The attribute *staticType* is used to guarantee that the expression component is of type *BooleanType*. The well known intention of the while-construct is to evaluate the expression, and then, if and only if it evaluates to true, to execute the statement list. After the execution of the statement list, the whole process is repeated. In our special version of

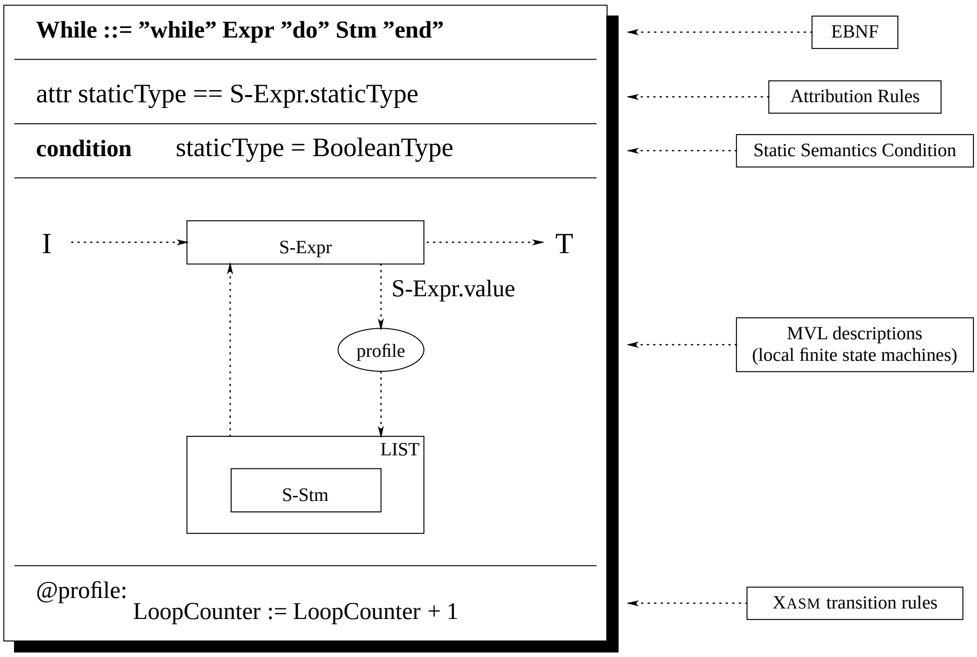

While ::= ”while” Expr ”do” Stm ”end”
attr staticType == S-Expr.staticType
condition staticType = BooleanType
I
S-Expr
T
S-Expr.value
profile
LIST
S-Stm
@profile:
LoopCounter := LoopCounter + 1
EBNF
Attribution Rules
Static Semantics Condition
MVL descriptions
(local finite state machines)
XASM transition rules

Fig. 10: The while example

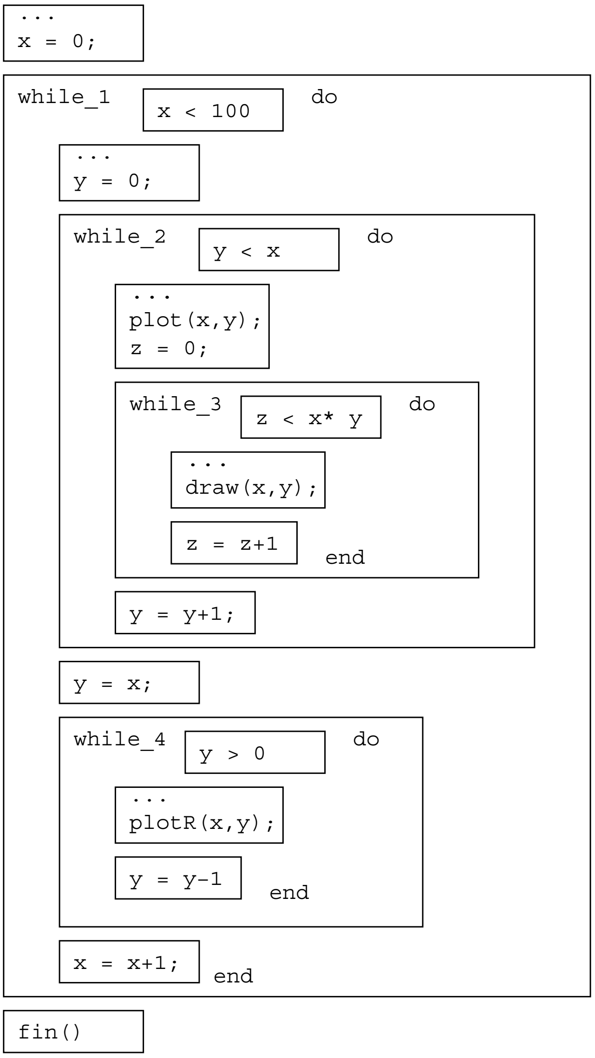
the while-statement, a counter *LoopCounter* is increased each time before the statement-list is executed.

The local finite state machine specifies exactly this behavior. The control enters the machine at the special, initial ”I” state. The ”I”-state leads immediately into the expression. We assume that the visit of the expression results in its evaluation, and that the result of the evaluation can be accessed as attribute *value* of the expression. After the evaluation of the expression, there are two possibilities. Either the expression evaluated to true and therefore transition with the firing condition *S-Expr.value* to the profile-state is chosen, or otherwise the transition is to the special state ”T” is chosen. This second special state marks the terminal or final state of the local machine.

Transitions like the one going to ”T”, having no firing condition are considered to fire in the *default case*. The default case is defined to happen, if no other transition exists whose firing condition evaluates to true. The Montages state machines first try to choose a transition with firing condition evaluating to true, else they choose a default transition. If there are several transitions, one is chosen nondeterministically. In our example, there are two transitions from the expression, one with firing condition going to the *profile* state, and one with default condition, going to the *T* state.

If the transition to *profile* is chosen, the profile state is visited next. The corresponding action rule increases the value of LoopCounter by one. Afterwards the statement-list is visited. List elements are visited by default sequentially. After the execution of the last statement in the list, the transition from the list to the expression is chosen, and the expression is reevaluated.

In a program a language construct is typically used several times. For in-



```
...
x = 0;
while_1  x < 100  do
    ...
    y = 0;
    while_2  y < x  do
        ...
        plot(x,y);
        z = 0;
        while_3  z < x* y  do
            ...
            draw(x,y);
            z = z+1  end
        y = y+1;
    y = x;
    while_4  y > 0  do
        ...
        plotR(x,y);
        y = y-1  end
    x = x+1;  end
fin()
```

Fig. 11: Program

stance in the program shown in Figure 11 we see four instances of while, which are numbered. The instances two and four are part of the statement-list of the first instance, and instance three is part of the statement-list of the second instance. This nesting is depicted as nested boxes.

An alternative, more traditional representation of the programs structure is the syntax tree shown in Figure 12. In order to keep the representation compact, we represent lists as dotted boxes, and show only the parent-child relation from while-instances to their expression and statement siblings. The *selectors* S-Expr and S-Sum are used to label these relations.

While the transitions in the While-Montage form an intuitive circle, representing loop behavior, it is less trivial to understand how this loop is applied to a complete program. Therefore we show how each transition in the Montages is instantiated in the syntax tree. The first transition in the While-Montage goes from the ”I”-state to the expression. In the program it connects the last state-

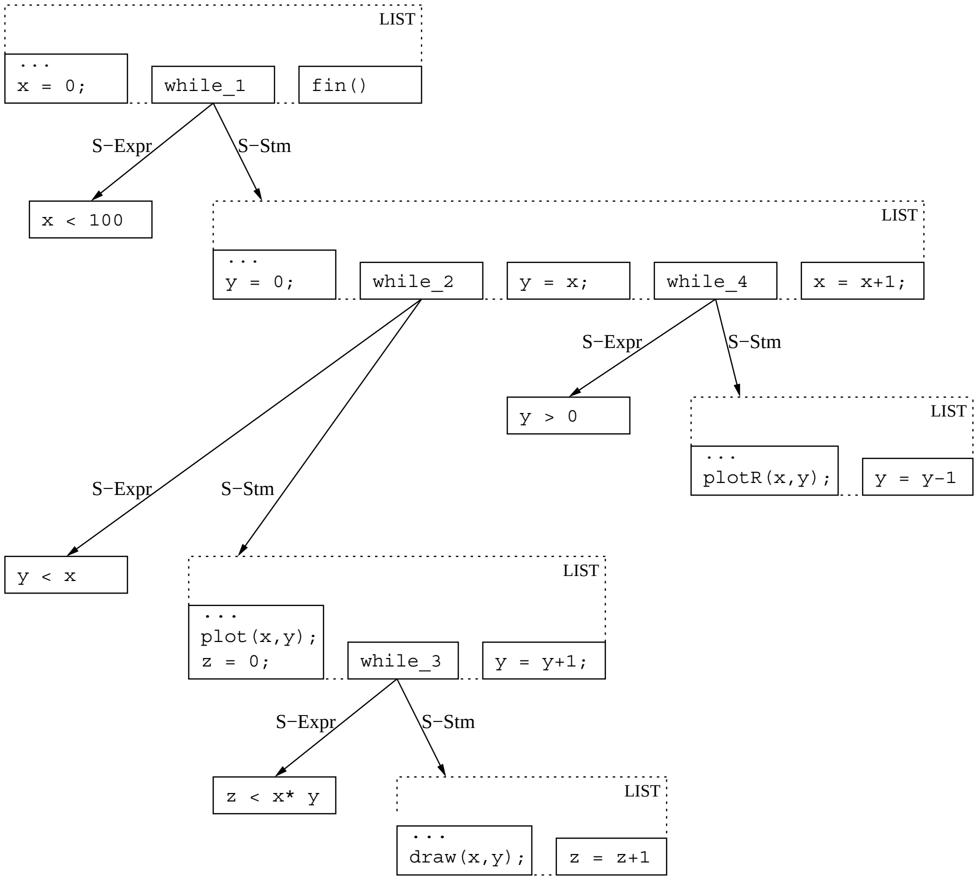


Fig. 12: Parse tree

ment before a while loop with the expression-component of a while loop. In Figure 13 the corresponding transitions are shown for all four instances of the while, being numbered accordingly. Correspondingly the transition from the expression-component to the profile state connects the expression of a while-statement with the first following statement, as depicted in Figure 14. The ”I” and ”T” states are thus used to plug the state machine of each while-loop into the state machine of the program.

Inside a while-statement, a transition with firing condition *src.value* goes from the expression to the profile state and a default transition links the profile-state to the statement-list. For each instance of a while the profile-state and the connecting transitions are drawn in Figure 15. Finally in Figure 16 the transition from the statement-list back to the expression is visualized. The complete transition graph is shown in Figure 17. The presented state machine is executed starting with the first statement in the topmost list, following lists sequentially if there are now explicit transitions, otherwise following the given transitions. In this way the program has been transformed in a state machine structure over the parse tree which is directly executable. Starting with the first statement, the variable x is set to 0. Then the transition leads us to the evaluation of $x < 100$. From this program fragment, two possible transitions can be chosen. One, assuming

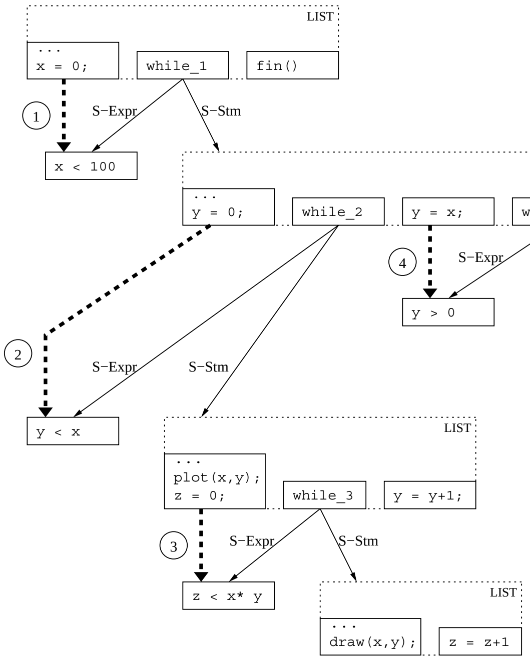


Fig. 13: Parse tree with I-arrows

that the value of the expression evaluates to true, leads to the first profile-state, the second leads back to the topmost list of statements. Since $x = 0$, the fist transition to profile is chosen, and the counter *LoopCounter* is increased by one. Then the list of statements within the first while instance is visited. After the update of y to 0, a transition leads us to the expression-component $y < x$ of the second while component. Like this, the complete program can be executed.

The main part of this chapter contains a more detailed overview of how Montages specify execution behavior of programs by making the parse tree an executable state machine. In Section 3.3 we give an intuitive definition of the execution behavior related aspects of Montages. It is shown how the MVL descriptions given for each language construct and the nodes of the AST define together the state-space and transitions of a special kind of state machines, called *Tree Finite State Machines* (TFSMs). In these machines, the states are pairs of MVL-states and AST-nodes. Each MVL-transition specifies TFSM-transitions for each AST-node associated with the Montage it is contained in. The definition of dynamic semantics by means of TFSMs is given in Section 3.3. In Section 3.4 the TFSM model is used to give the definitions of list processing and to explain how non-local transitions are defined in Montages. In order to make these descriptions more precise than the previous while-example, we start with a closer look on syntax definitions and the construction of the AST.

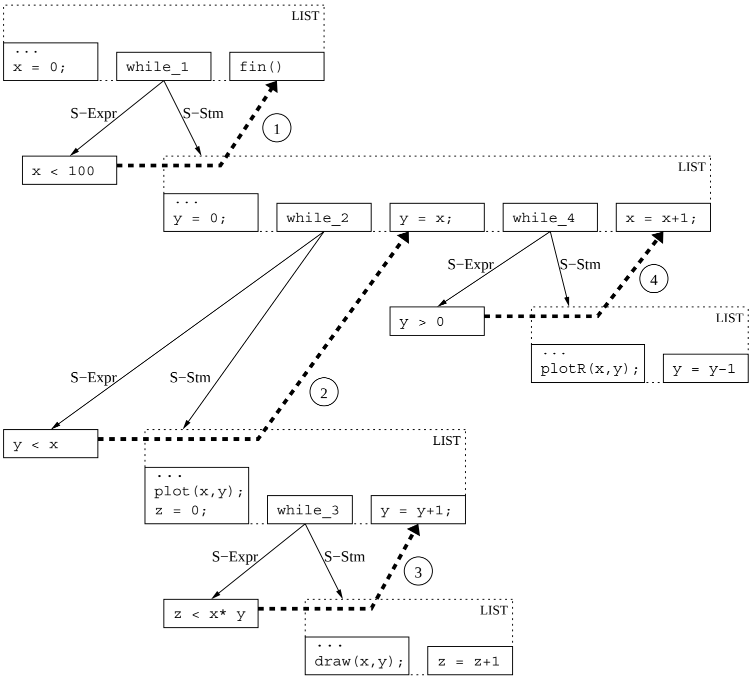


Fig. 14: Parse tree with T-arrows

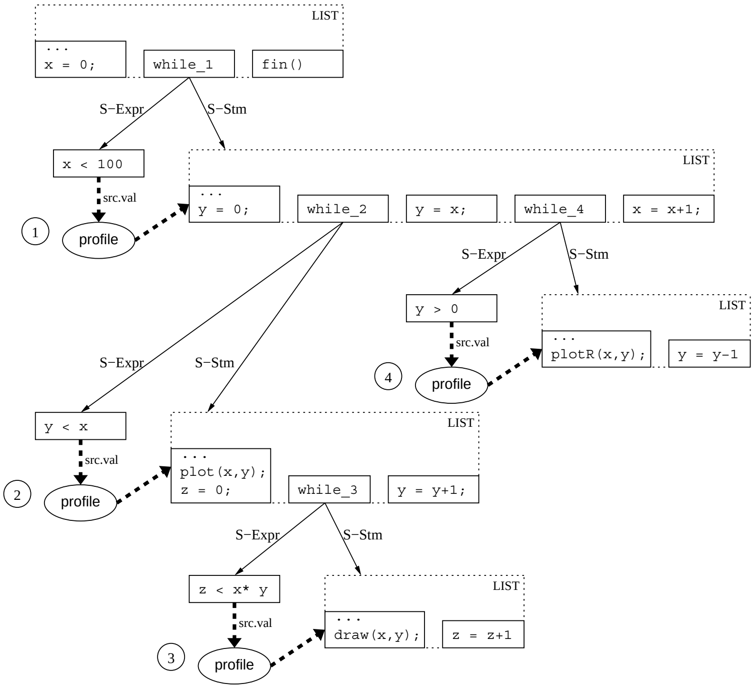


Fig. 15: Parse tree with profile action and arrows

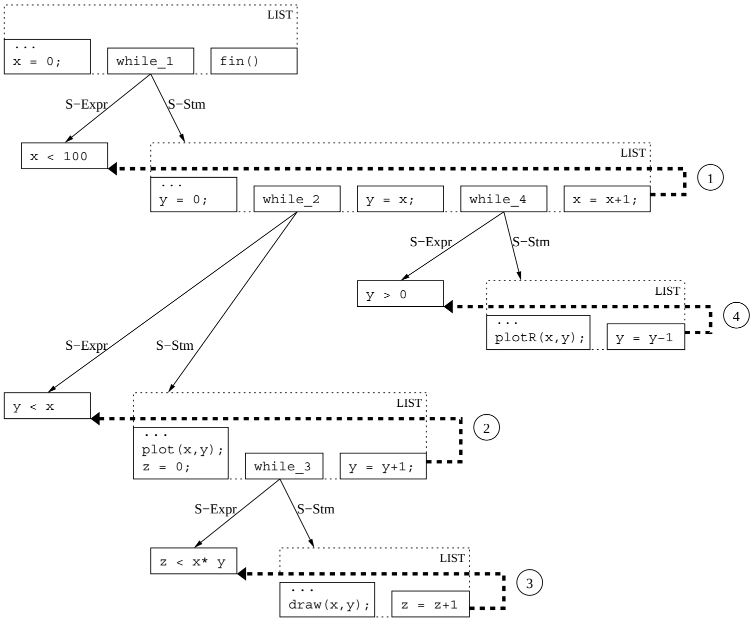


Fig. 16: Parse tree with the back arrow

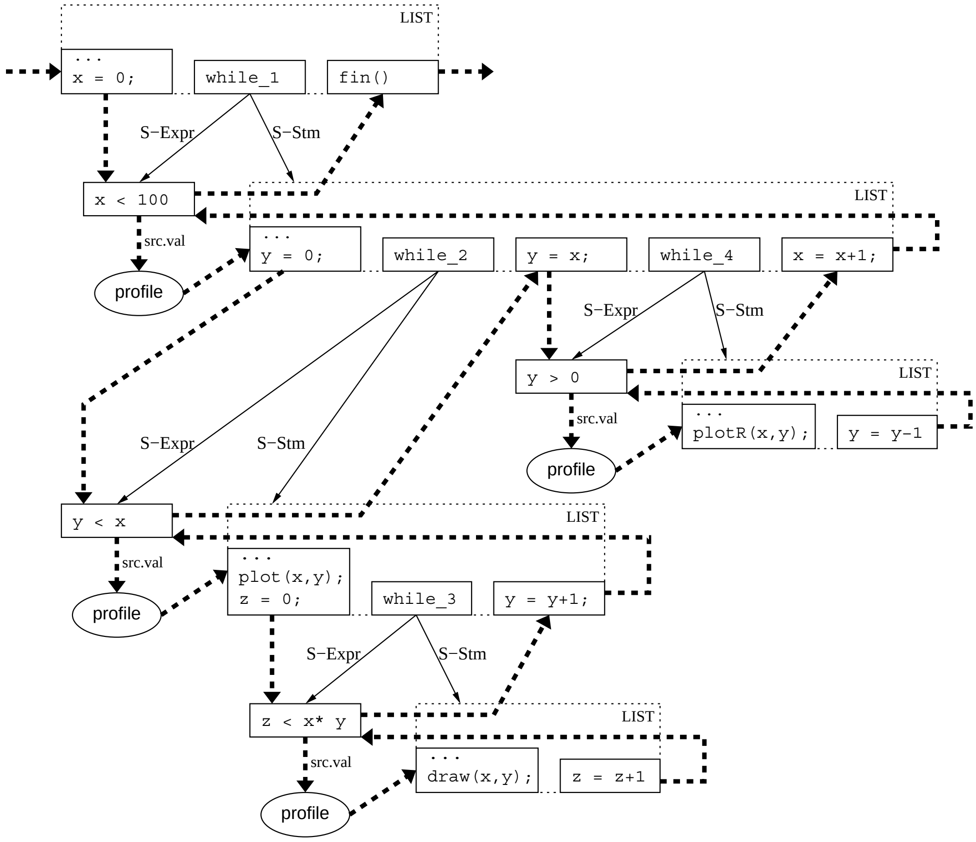


Fig. 17: Parse tree with all arrows

3.2 From Syntax to Abstract Syntax Trees (ASTs)

In this section, the transformation from a program into an AST is described. This also forms the basis for classifying the nodes with characteristic and synonym universes and for navigating trough the AST using selector functions.

3.2.1 EBNF rules

The syntax of the specified language is given by the collection of all the EBNF rules defined in the different Montages. Following the approach of Uhl (212), we assume that the rules are given in one of the two following forms:

$$\begin{array}{rcl} A & ::= & B\ C\ D\ D \\ E & = & F \mid G \mid H \end{array}$$

The first form declares that A contains the components B, C, D, and again D in that order whereas the second form defines that E has exactly one of the alternative components F, G, or H. Rules of the first form are called *characteristic productions*[4] and rules of the second form are called *synonym productions*. It is then possible to guarantee that each non-terminal symbol appears in exactly one rule as the left-hand-side. Non-terminal symbols appearing on the left of the first form of rules are called *characteristic symbols* and those appearing on the left of synonym productions are called *synonym symbols*. EBNF also features lists and options which may be used in right-hand-sides of productions and are going to be introduced in Section 3.4.

3.2.2 Abstract syntax trees

The treatment of characteristic and synonym productions described above allows an automatic generation of an abstract syntax tree (AST) from the concrete EBNF-syntax, as defined by Odersky (167). The resulting ASTs are relatively compact. The idea for making the tree compact is to create nodes only for parsed characteristic symbols, and to represent synonym symbols by adding additional labels. Each node is thus labeled by exactly one characteristic symbol and zero or more synonym symbols. Labeling of nodes is done by declaring a set or universe for each symbol. Adding a label l to a node n is done by putting n into universe l. As a consequence, the characteristic universes partition the universe of AST nodes. For each characteristic universe U a Montage is given, specifying syntax and semantics of U's elements. Given a node, the associated Montage is referred to as ”its Montage”, and given a Montage, the elements of the corresponding characteristic universe are called the ”instances of the Montage”.

[4] In the original publications (212; 167) the name of ”characteristic production” is ”generator production”, since only these productions generate a new node in the AST. We have chosen the name characteristic production, because they can be used to characterize the nodes as described above.

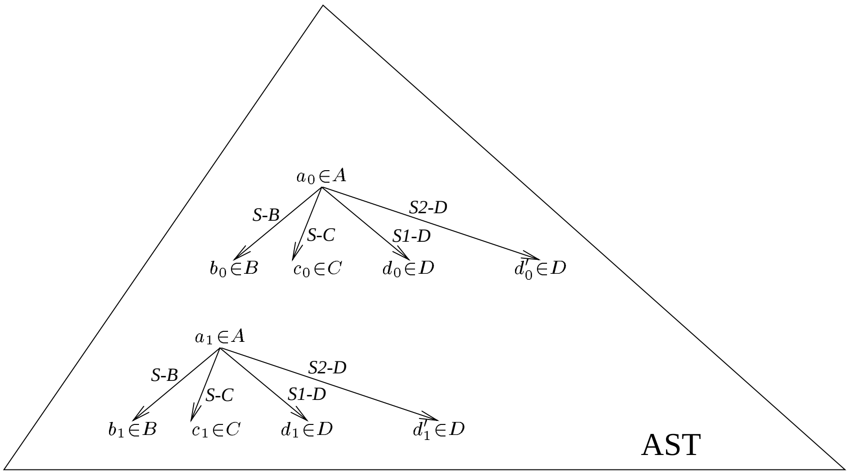


Fig. 18: Instances of universe A, definitions of selectors *S-B, S-C, S1-D, S2-D*

The so called *selector functions* can be used to navigate through the AST. Selector functions are defined as follows. Each node n in the AST has been generated by some characteristic rule

$$A ::= Z_1 Z_2 \ldots Z_n$$

For each symbol Z_i appearing only once on the right-hand-side of the rule, the selector function

$$\textit{S-Z}_i : Node \rightarrow Node$$

maps n to its unique Z_i-sibling. For each symbol Z_j appearing more then once, the selector functions

$$\begin{array}{rl} \textit{S1-Z}_j : & Node \rightarrow Node \\ \textit{S2-Z}_j : & Node \rightarrow Node \\ \ldots & \ldots \\ \textit{Sm-Z}_j : & Node \rightarrow Node \end{array}$$

map n to its first, second, ..., m-th Z_j-sibling. Given for instance the rule *A := B C D D,* Figure 18 visualizes the situation for two A instances a_0 and a_1.

In order to allow to traverse a tree in arbitrary ways we define in addition the function *Parent* which links each node with its parent-node in the tree.

Example

As a running example we give a small language $\mathcal{S}$. For the moment, we can abstract from the meaning of $\mathcal{S}$ programs and consider them as examples for the construction of ASTs. The start symbol of the grammar is Expr, and the production rules are

Gram. 1:

$$\begin{array}{lcl} \textit{Expr} & = & \textit{Sum} \mid \textit{Factor} \\ \textit{Sum} & ::= & \textit{Factor} \text{ “+” } \textit{Expr} \\ \textit{Factor} & = & \textit{Variable} \mid \textit{Constant} \end{array}$$

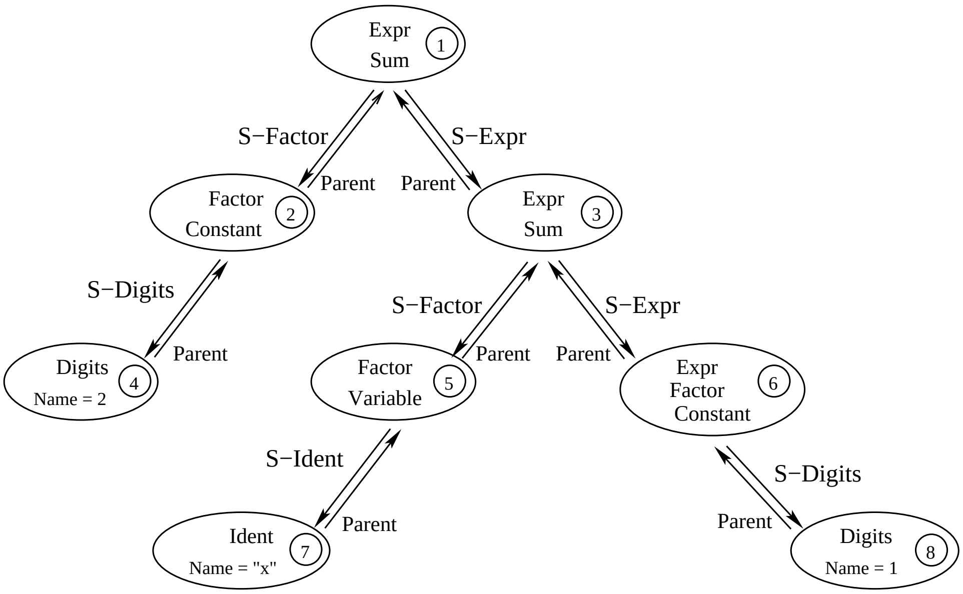


Fig. 19: The abstract syntax tree for `2 + x + 1`

Variable	::=	*Ident*
Constant	::=	*Digits*

The following term is an $\mathcal{S}$-program:

```
2 + x + 1
```

As a result of the generation of the AST we obtain the structure represented in Figure 19. The labels indicate to which universes a node belongs, and the definitions of the selector functions are visualized as edges. The leaf nodes contain the definition of the attribute *Name*, which in turn contains the micro-syntax of the parsed Digits- and Ident-values. The function *Parent* is visualized with the edges going from the leaves towards the root of the tree.

3.3 Dynamic Semantics with Tree Finite State Machines (TFSMs)

In Montages, dynamic semantics is given by *Tree Finite State Machines* (TFSMs), a special kind of state machines which we deviced for allowing AST's being executed without transforming them. The states of a TFSM are tuples consisting of an AST-node, and a state of the local state machine given for each node by means of its Montage. Execution of programs can be understood and visualized by highlighting the current node *CNode* in the AST and the current state *CState* in the corresponding Montage. If the state *(CNode, CState)* is visited, the action rule associated with *CState* is executed, using attributes and fields of *CNode* to store and retrieve intermediate results.

Notational Conventions

As mentioned, a language definition consists of a set of Montages, which defines a mapping from EBNF productions to local state machines, and indirectly from AST nodes to local state machines. Given these mappings, the states of a TFSM are tuples consisting of an AST-node and a state of its associated local state-machine. Throughout this text we are saying that a TFSM is "*in state S of node N*", rather than the more precise formulation *in the state being the tuple formed by state S, node N*. Further we use the notion "state of a node's Montage", rather than the more precise, but lengthy formulation "state of the local state machine associate with a node via the Montage associate with the EBNF production which created the node. The local state machines and their embedding into the TFSM are given by means of *Montages Visual Language* (MVL). in the descriptions we will use the terms "local (finite) state machine" and "MVL-machine" to denote the machines associated with AST nodes, and we will use the terms "(finite) state machine" and "TFSM" for the global machine representing the dynamic semantics of an AST.

TFSM transitions

Transitions in TFSMs change both the current node *CNode* and the current state *CState*. A TFSM-transition t is defined to have five components, the source node *sn*, the source state *ss*, the condition c, the target node *tn*, and the target state *ts*.

$$t = (sn, ss, c, tn, ts)$$

In the condition expression c, the source node *sn* can be referred to as bound variable *src*, and the target node *tn* as bound variable *trg*. Typically conditions depend on attributes of the source and/or target node. The source state and target state cannot be referred to in the condition. A transition can be activated if its source node *sn* is equal to the current node *CNode*, its source state *ss* is equal to the current state *CState*, and if its condition c evaluates to true; if a transition is activated, in the next state the current node *CNode* is equals the target node *tn* and the current state *CState* equals the target state *ts*.

Montages Visual Language (MVL)

The state machine of a Montages is given in *Montages Visual Language*(MVL).

Transitions in MVL are specifications for one or many TFSM-transitions. MVL defines how MVL-transitions of the Montages are instantiated with TFSM transitions. In Section 3.3.2 we give the corresponding definitions in form of the algorithm *InstantiateTransition*. Later in Section 3.3.3 this algorithm is used to construct a TFSM, in Section 3.3.4 the simplification of TFSMs is discussed, and finally in Section 3.3.5 their execution is described. More advanced features, allowing to specify families of transitions by means of references to lists and sets of nodes are introduced later in Section 3.4.

Isomorphism between "flat" view and TFSM view
In the following examples, as already in the while-example (Figures 11, 12, 13, 14, 15, 16, and 17), the MVL-machines are drawn repeatedly for each AST-node and therefore the states of these figures corresponds directly to TFSM-states. This visualization is called the "flat" view on TFSM, and is mathematically isomorphic with the TFSM model. In Figure 20 the isomorphism between the "flat" view and the TFSM view is illustrated. On both sides of the figure, the same AST with three nodes is shown, a parent node, and two sibblings. We assume that both sibblings are produced by the same EBNF rules, and consequently they are associated with the same MVL-machine. In the given example, this machine consists of exactly one MVL-state labeled *a* and a transition sourcing in *a*. The target of the transition is not specified in the current context. On the left-hand-side the "flat" intuition is shown, where the MVL machine is instanciated for each corresponding AST node. As a consequence, there are two instances of the same state *a*, and the transitions sourcing in *a* are departing from these instances. On the right hand side, the corresponding TFSM view is shown. The MVL machine is existing only once, and not instanciated. The states of the TFSM are not the states of the MVL machine, but tuples consisting of an AST node, and an MVL state of the corresponding machine. In our figure there are two such tuples, visualized as dotted double-headed arrows, labeled (_, _). The MVL transitions sourcing in the MVL-state a correspond now to the two TFSM transitions sourcing in the TFSM tuple-states.

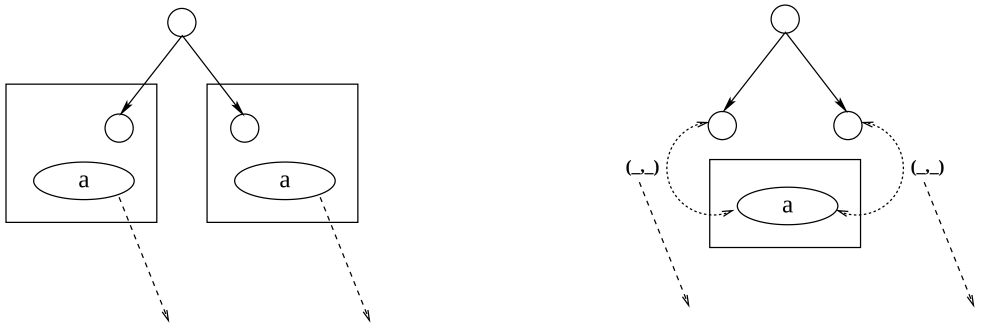


Fig. 20: Isomorphism between "flat" view and TFSM view

3.3.1 Example Language $\mathcal{S}$

Throughout this Section we use the previously introduced examples *A*, *While*, and the Montages presented here for the language $\mathcal{S}$ whose grammar has been introduced in Section 3.2. We show now informally how the MVL-state machines of the Montages together with the AST can be used to execute a program by intrepreting it as a TFSM. The same example will be used in the following sections as examples for the formal TFSM definitions.

The programs of language $\mathcal{S}$ are arithmetic expressions which may have side effects and are specified to be evaluated from left to right. The atomic factors are constants and variables of type integer.

The Montage for `Sum` is shown in Figure 21. The topmost part of this Montage is the production rule defining the context-free syntax consisting of a *Factor* and an *Expr* right-hand-side symbol. The second part defines the states and transitions of this construct by means of a MVL description. All transitions are labeled with the empty firing condition. The control enters the state machine at the ”I”-state, visits the state machine corresponding to the Factor-sibling, then the state machine corresponding to the Expr-sibling and finally the ”add”-state is visited, resulting in the execution of its action rule. The XASM action rule, which is given in the third part accesses the *value*-attributes of the siblings of a Sum-instance, and assigns their sum to the *value*-attribute of the Sum-instance. Finally, the ”T”-state is visited being the final state of the Sum state machine.

The Montages *Variable* and *Constant* are shown in Figure 22. Both of them contain exactly one state, the *Variable*-Montage's state triggers a rule reading the value of the referenced variable from the *CurrentStore*, and the constant Montage's state triggers a rule reading the constant value. Both actions set the value-attribute to the corresponding result.

In Figure 23 we represents the MVL sections of these Montages as they are associated with the corresponding nodes of the AST we showed already in Figure 19. Visiting a state s in Figure 23, the current state *CState* is state s in the corresponding Montage, and the current node *CNode* is the node associated by the dotted line.

Based on this “flat” representation, the boxes in the state machines can be replaced with the state machine corresponding to the sibling referenced by the box

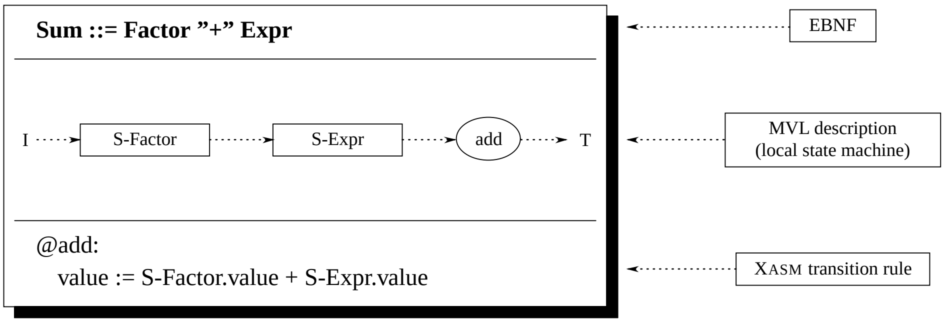


Fig. 21: Montage components.

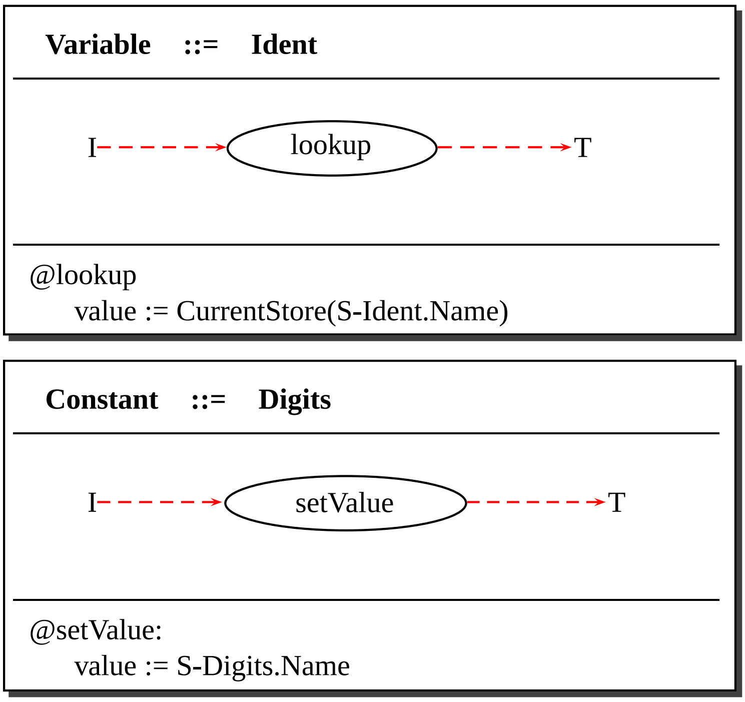


Fig. 22: The Montages for the language $\mathcal{S}$.

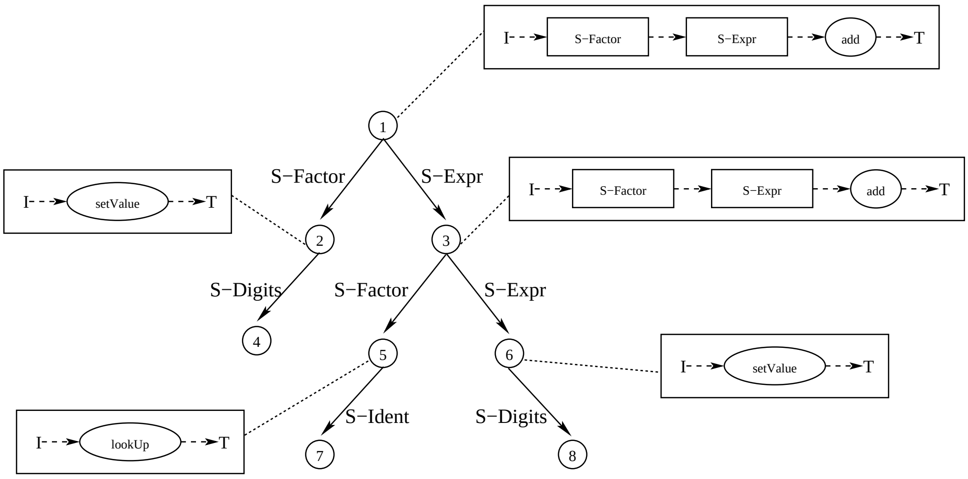


Fig. 23: The finite state machines belonging to the nodes.

label. The S-Expr box of the state machine associated with node 1 in Figure 23 is for instance replaced by the state machine associated with node 3, being the S-Expr sibling of node 1. In Figure 24 the resulting hierarchical state machine is represented. The AST-nodes associated with the states are here directly surrounding the states. In Figure 24 the hierarchy of the AST is visualized as nested boxes, labeled by the selector functions. This visualization corresponds to a MVL-description of the complete program.

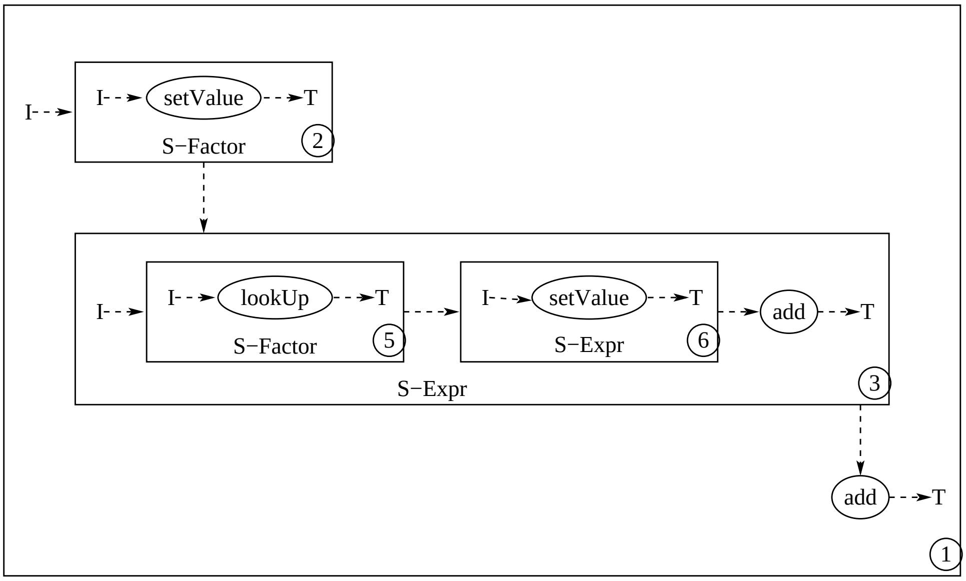


Fig. 24: The constructed hierarchical finite state machine.

We can even go one step further, transforming the hierarchical state machine into a flat one. Since we know that execution entry and exit points for each language construct are marked by the special states ”I” and ”T”, we replace each transition whose target is a box representing an AST node n, by a transition whose target is *(n, ”I”)*, and correspondingly we replace each transition whose source is a box representing an AST node n, by a transition whose source is *(n, ”T”)*. The resulting visualization is given in Figure 25. Each oval, I, and T represents directly a state in the TFSM, whose node component is given by the dotted arrow into the AST, and whose state component is given by the label.

Since the ”I” and ”T” states are not associated with action rules, and since all transitions are labeled by the empty condition, the state machine of Figure 25 can be simplified into the one shown in Figure 26.

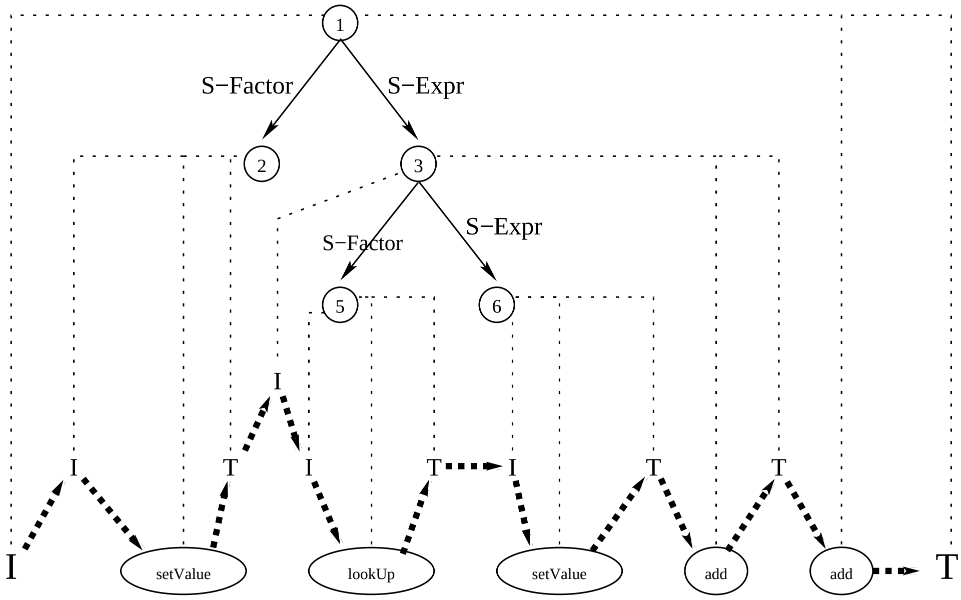


Fig. 25: The flat finite state machine and its relation to the AST.

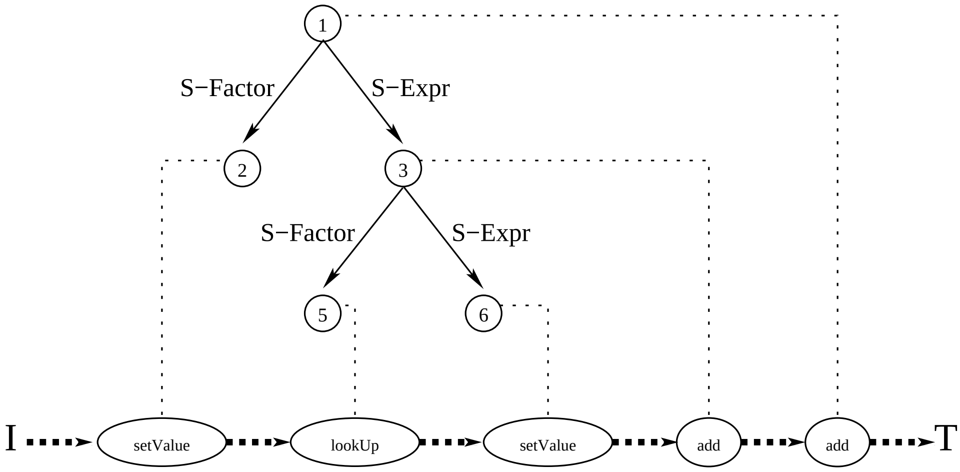


Fig. 26: The simplified finite state machine and its relation to the AST.

At this point, we can understand the dynamic semantics of the program by executing the state machine. First, the initial state of the root node is visited. Then the following steps are repeated.

1. The action rule associated with the visited state is executed.

2. A control arrow whose firing condition evaluates to true is chosen, and the state it points to is visited next. If there is more than one possible next state, one of them is chosen nondeterministically. If there is no arrow with a predicate evaluating to true, an arrow with the *default*-condition is chosen. If there is no arrow with the *default*-condition either, the same state is visited again.

3. Goto step 1.

Coming back to our example, assuming that *CurrentStore* maps x to 4, the execution of the state machine in Figure 26 sets the value of node two to the constant 2, sets the value of node five to 4, sets the value of node six to 1, sets the value of node three to the sum of 4 and 1, and finally sets the value of node one to the sum of 2 and 5.

3.3.2 Transition Specifications and Paths

Montages define a TFSM for each program of the specified language by giving the context-free grammar and a local state machine for each characteristic symbol in the grammar. The local state machine, given by means of MVL, consists of a set of states, associated with action rules, and a set of MVL-transitions.

As mentioned, the states of the TFSM range over the Cartesian product of AST-nodes and MVL-states, and transitions have five components, the source, consisting of a source AST-node and a source MVL-state, the condition, and the target, consisting of a target node, and a target state. The MVL-transitions are considered to be *transition specifications* which are instantiated as TFSM-transitions. In this refined view an MVL-transition specification has three components, the source path, the condition, and the target path. The MVL visualization of a transition specification is an arrow from the visualization of the source path to the visualization of the target path. The condition of the transition specification is used as the label of the arrow.

The MVL-elements for visualizing paths are boxes and ovals. A state of the MVL-machine is a special case of a path. With respect to an instance n of the Montages containing the MVL-elements, their semantics can be described as follows:

- The *oval nodes* are the states. The states are labeled with an attribute. It serves to identify the state, for example if it is the target of a state transition or if it is associated with an action rule. If a state is visited, the associated action rule is executed, such that intermediate results are saved and retrieved as attributes of n and its siblings.

- There are two special kind of states denoting the entry and exit points of the MVL state machine. The initial state I, represented by the letter ”I”, denotes the first state visited, if the machine is entered. The terminal state ”T” denotes the last state visited.

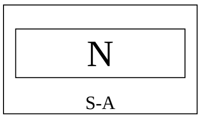
- The *rectangular nodes* or boxes represent siblings of n. They are labeled with the corresponding selector function. Boxes may contain other boxes and ovals. Boxes contained in other boxes represent siblings of siblings. Ovals in boxes represent the corresponding state of the node represented by the surrounding box.

  Later in Section 3.4 we will introduce special boxes referencing all elements in a lists of siblings as well as boxes referencing all elements of characteristic and synonym universes.

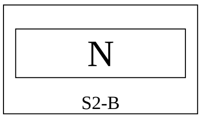
A path can be represented visually by means of nested boxes and ovals, as discribed above, or textually. The textual representation of a path is a term which is recursively built up by the following operators *siblingPath* and *statePath*.

- *siblingPath(Ident, Int, Path)*

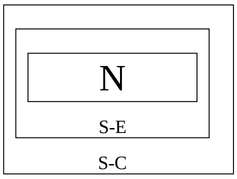
  The arguments of a siblingPath are *Ident*, the symbol of the sibling, *Int*, its occurrence, and *Path*, the relative path from the denoted sibling to the target of the full path. The relative path is never empty, since the target of a full path needs to denote a state. Occurrence *undef* is used for unique symbols in the right-hand-side of a grammar rule. The paths siblingPath(”A”, undef, N), siblingPath(”B”, 2, N), siblingPath(”C”, undef, siblingPath(”D”, undef, N)) are visualized as follows. The box N stands for an arbitrary relative path.



siblingPath(”A”, undef, N)



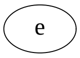
siblingPath(”B”, 2, N)



siblingPath(”C”, undef, siblingPath(”D”, undef, N))

- *statePath(Ident)*

  The argument of a state path is the name *Ident* of the state. The paths statePath(”e”), statePath(”I”), statePath(”T”), siblingPath(”A”, undef, statePath(”f”)), siblingPath(”B”, 2, statePath(”g”)), siblingPath(”C”, undef, siblingPath(”D”, undef, statePath(”h”))) are visualized as follows.



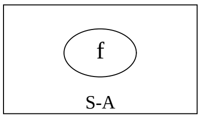
statePath(”e”)

I

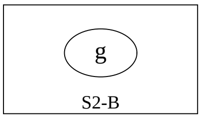
statePath(”I”)

T

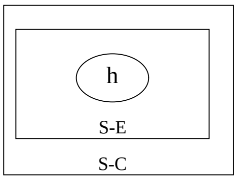
statePath(”T”)



siblingPath(”A”, undef, statePath(”f”))



siblingPath(”B”, 2, statePath(”g”))



siblingPath(”C”, undef, siblingPath(”D”, undef, statePath(”h”)))

A special short-hand notation is allowed in the visual notation. If the source of a transition is not a state, but a box referencing a node, the transition is assumed to source in the ”T”-state of the corresponding node. Correspondingly, if the target of a transition is a box, the transition is assumed to target the ”I” state of the referenced node. The short-hand notation is allowed, since the ”I”-state is considered as a collector of all transitions incoming to a node, and the ”T”-state is considered as a starting point of all transitions leaving a node.

According to the given definitions, we can now represent the MVL-transitions in the abstract A-Montage (Figure 9) as the following triples.

Term 1:

```
(siblingPath("B", undef, statePath("s1")),
 C1,
 siblingPath("C", undef, statePath("s2")))

(siblingPath("C", undef, statePath("T")),
 C2,
 statePath("s3"))

(statePath("s3"),
 C3,
 siblingPath("B", undef", statePath("I")))
```

The source of the C2 transition, being a box, has been completed in the textual representation with state ”T”, whereas the target of the C3 transition has been completed with state ”I”. Another example is given by the following textual representations of the transitions in the While Montage (Figure 10).

Term 2:

```
(statePath("I"),
 default,
 siblingPath("Expr", undef", statePath("I")))

(siblingPath("Expr", undef, statePath("T")),
 src.value,
 statePath("profile"))
```

```
(siblingPath("Expr", undef, statePath("T")),
 default,
 statePath("T"))

(statePath("profile"),
 default,
 siblingPath("Stm", undef, statePath("LIST")))

(siblingPath("Stm", undef, statePath("LIST")),
 default,
 siblingPath("Expr", undef", statePath("I")))
```

Please note, that the special treatment of lists, together with the state ”LIST” will be discussed later in Section 3.4.

3.3.3 Construction of the TFSM

The construction of a TFSM for a given AST is done by instantiating for each instance of a Montage all transition specifications given in its MVL state machine.

The instantiation of the MVL-transition specifications with TFSM transitions is done by the algorithm *InstantiateTransition*. Given a node n of the AST, and a transition specification t

$$t = (\mathit{SourcePath}, \mathit{Condition}, \mathit{TargetPath})$$

of the corresponding Montage, t is instantiated as a TFSM transition t' which is constructed as follows.

The four global variables *SourceNode0*, *SourcePath0*, *TargetNode0*, and *TargetPath0* are initialized such that SourceNode0 and TargetNode0 equal node n, SourcePath0 is initialized with the *SourcePath* parameter of t, and TargetPath0 is initialized with the *TargetPath* parameter of t.

$$\begin{aligned}
\mathit{SourceNode0} &= n\\
\mathit{SourcePath0} &= \mathit{SourcePath}\\
\mathit{TargetNode0} &= n\\
\mathit{TargetPath0} &= \mathit{TargetPath}
\end{aligned}$$

At each step, InstantiateTransition checks, whether *SourcePath0* (or *TargetPath0*) is matching a term like *siblingPath(Symbol, Occ, Path0)*. If so, the corresponding selector function for *Symbol* is applied to the *SourceNode0* (respectively *TargetNode0*) resulting in node n'; the corresponding global variable *SourceNode0* (respectively *TargetNode0*) is updated with the new node n' and the global variable *SourcePath0* (respectively *TargetPath0*) is updated with *Path0*. In the following pseudo-code `"=~"` is used to denote ”matches a term like”, corresponding to pattern matching in functional languages. The pattern variables are marked with a &-sign.

```
if SourcePath0 =~ siblingPath(&Symbol, &Occ, &Path0) then
```

```
   let n' = (selector function (&Symbol, &Occc)
             applied to SourceNode0) in
     SourceNode0 := n'
     SourcePath0 := &Path0
if TargetPath0 =~ siblingPath(&Symbol, &Occ, &Path0) then
   let n' = (selector function (&Symbol, &Occ)
             applied to TargetNode0) in
     TargetNode0 := n'
     TargetPath0 := &Path0
```

After a number of steps, SourcePath0 matches a term like *statePath(&srcS)* and TargetPath0 matches a term like *statePath(&trgS)*. At this point Instantiate-Transition generates the TFSM transition t' defined as follows.

$$t' = (\textit{SourceNode0}, \textit{\&srcS}, \textit{Condition}, \textit{TargetNode0}, \textit{\&trgS})$$

Coming back to our running example, the transition specifications of the Montages Sum can be textually represented as follows.

Term 3:

```
Montage Sum:
  (statePath("I"),
   true,
   siblingPath("Factor", undef, statePath("I")))

  (siblingPath("Factor", undef, statePath("T")),
   true,
   siblingPath("Expr", undef, statePath("I")))

  (siblingPath("Factor", undef, statePath("T")),
   true,
   statePath("add"))

  (statePath("add"),
   true,
   statePath("T"))
```

Transitions to and from boxes are directly represented as arrows to or from the corresponding I or T state. The corresponding textual representation of the transition specifications in Montages *Variable* and *Constant* is given below.

Term 4:

```
Montage Variable:
  (statePath("I"),
   true,
   statePath("lookup"))

  (statePath("lookup"),
   true,
   statePath("T"))

Montage Constant:
  (statePath("I"),
   true,
   statePath("setValue"))
```

```
(statePath("setValue"),
 true,
 statePath("T"))
```

The instantiation of the transition specifications for all nodes $n_1, n_2, \ldots, n_6$ in AST of the program example *2 + x + 1* results into the following list of TFSM transitions.

```
(n1, "I",        true, n2, "I")
(n2, "I",        true, n2, "setValue")
(n2, "setValue", true, n2, "T")
(n2, "T",        true, n3, "I")
(n3, "I",        true, n5, "I")
(n5, "I",        true, n5, "lookup")
(n5, "lookup",   true, n5, "T")
(n5, "T",        true, n6, "I")
(n6, "I",        true, n6, "setValue")
(n6, "T",        true, n3, "add")
(n3, "add",      true, n3, "T")
(n3, "T",        true, n1, "add")
(n1, "add",      true, n1, "T")
```

In fact, these transitions correspond exactly to the transitions in Figure 25, taking as source and target of a transition the combination of the states together with the nodes referenced by the dotted arrows.

3.3.4 Simplification of TFSM

The simplification resulting in Figure 26 can now be described as follows. If there exists two transitions

$$\begin{aligned} t_1 &= (n_1, s_1, c_1, n_2, s_2) \\ t_2 &= (n_2, s_2, c_2, n_3, s_3) \end{aligned}$$

such that s_2 equals ”I” or ”T”, then t_1 and t_2 can be replaced by transition

$$t_3 = (n_1, s_1, c_1 and c_2, n_3, s_3)$$

This simplification algorithm only works if there is exactly one ”I” and one ”T” arrow in a Montage and if ”I” and ”T” states are not associated with actions. Otherwise a more general simplification algorithm removes all states not having an action associate and combines incoming and outgoing transitions. In the upper part of Figure 27 we see a state/node pair *(s, n)* of a TFSM which is a candidate for removal from the TFSM transition graph. If the state s in the MVL-graph of the Montage associated with node n is not associated with an action rule, the (s, n) pair can be removed, and the incoming and outgoing transitions can be combined as visualized in the lower part of Figure 27.

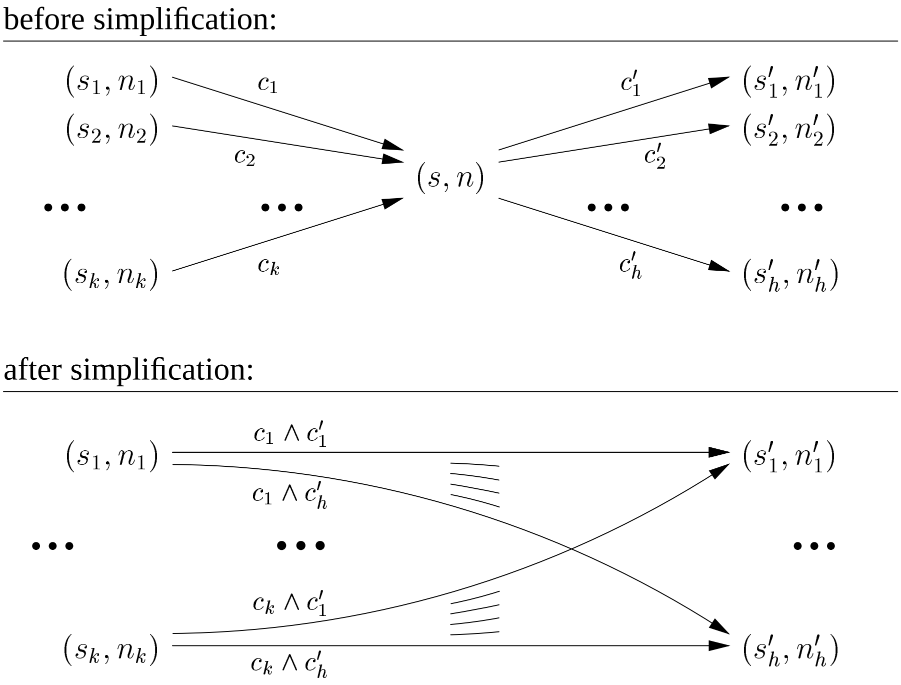


Fig. 27: A TFSM fragment before and after simplification

3.3.5 Execution of TFSMs

Execution of the program is now done by an algorithm Execute, which has two global variables, *CNode*, the current node, and *CState*, the current state. At the beginning, *CNode* is the root of the AST, and *CState* is ”I”.

$$\begin{aligned} \mathit{CNode} &= \mathit{root\ of\ AST} \\ \mathit{CState} &= \text{”}I\text{”} \end{aligned}$$

The core of Execute has two steps, which are repeated until the machine terminates. Termination criteria depend on the environment of the machine, e.g. whether the environment can change part of the machine’s state.

1. In the first step, the action rule of the state CState in the MVL state machine corresponding to CNode is executed.

2. In the second step, a TFSM transition

$$(\mathit{CNode}, \mathit{CState}, c, tn, ts)$$

is chosen, whose source node equals *CNode*, whose source state equals *CState*, and whose condition c evaluates to true. If such a transition exists, CNode is set to *tn* and CState is set to *ts*.

3. Then repeat the process, starting at step 1.

This general execution algorithm corresponds to the process described at the end of the example given in Section 3.3.1. This “core” algorithm is going to be formalized later in Section 6.1, in Section 6.4 it will serve as example for the new Montages tool architecture, and finally in Section 8.4.6 it is used as part of the formal semantics of the Montages formalism itself.

3.4 Lists, Options, and non-local Transitions

We have omitted up to now the treatment of lists and options in the EBNF rules, as well as non-local transition specifications in MVL. Both lists and non-local transition specifications can be used to specify a transition which corresponds to a set of TFSM transition instances, rather than a single instance. In the presence of lists and non-local transitions, the algorithm *InstantiateTransitions* generates from one transition specification in MVL a set of transitions in a TFSM.

In Section 3.4.1 we show the EBNF features to specify lists and options, as well as the way how the AST is constructed for such grammars, and how MVL-transitions from and to lists are instantiated in a TFSM with a family of transitions. The visual and textual representation of non-local transitions by means of so called *global paths* as well as the instantiation of transition specifications involving such paths is given in Section 3.4.3. In Section 3.4.2 we give the full specification of the algorithm instantiating the transitions by combining the definitions from Section 3.4.3 and Section 3.3.3. Finally in Section 3.4.5 we use a goto-language as example how a family of TFSM transition is generated for each transition specification in MVL.

3.4.1 List and Options

In characteristic rules, the right-hand-side symbols can be in curly *repetition brackets*, denoting a list of zero to many instances, or in square *option brackets*, denoting an optional instance. An optional B instance can be specified as follows:

```
 A ::= ... [B] ...
```

A possibly empty list of B instances has the following form

```
 A ::= ... {B} ...
```

A comma separated list of B instances with at least one member can be specified as follows.

```
 A ::= ... B  {"," B} ...
```

The same kind of list with zero or more members can be given using a combination of curly and square brackets.

```
 A ::=  ... [B {"," B}] ...
```

The mapping into ASTs is defined such, that each of the above right hand sides is mapped into a list of B instances. Further the EBNF list

```
{ C D }
```

parses sequences of C followed by D, but represents them as a list of C's and a list of D's, which are accessible with the corresponding selector functions. For instance a production

```
L ::= { C D }
```

parsing ”$C_1D_1C_2D_2C_3D_3$” results in two lists,

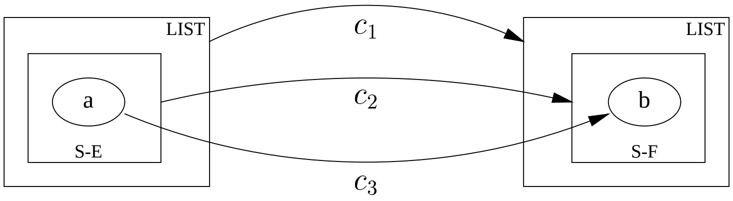


Fig. 28: Examples for MVL-Transitions connecting lists.

```
[C1, C2, C3], [D1, D2, D3]
```

which are accessible via selectors S-C and S-D[5].

The construction of the AST for lists and options works as follows. From the list or option operators the production creates an ordered sequence of zero, one, or more instances of the respective symbol enclosed in the operator is returned. This sequence is then transformed into an AST representation as follows. If it is

- of length 0, it is represented in the AST with a specially created node, which is an instance of universe *NoNode*. Consequently in the AST it cannot be seen whether an instance of NoNode has been generated by an option operator, or by a list operator.

- of length 1, it is represented in the AST as the node representing the unique member. In the AST we can therefore not see any difference between a list of length one, an instance produced from an optional symbol, or an instance produced from a normal symbol.

- of length 2 or longer, it is *not* transformed and represents itself in the AST.

There are two ways to refer to a list with a path. The first possibility is to refer to the elements of the list. In the first case, a transition specification from or to a path denoting a list of nodes is instantiated with a family of TFSM transitions, one for each element in the node.

Besides referring to elements of a list, it is possible to refer to the list itself, by using the LIST-box as source or target of a transition. In the textual representation the references to lists is represented by a special state *LIST*.

As example we show in Figure 28 MVL-transitions between lists. The visual representation of paths denoting lists is the visual representation of the denoted element, surrounded by a special box labeled with *LIST* which visualized the list itself. Such list boxes can only contain a single symbol, and represent a list of instances of that symbol, as described above. The visualization of the involved paths relate to two lists, one of E-instances, and one of F-instances. As mentioned above, they can occur on the right-hand-side of the characteristic

[5] A more flexible treatment of lists and options in Montages has been elaborated by Denzler (55)

production in any of the following forms, not changing anything in their visualization in MVL or representation in the AST. The list of possibilities is not complete.

- ... {E} ... {F} ...
- ... {F} ... {E} ...
- ... {E F} ...
- ... E ”,” {E} ... F ”,” {F} ...
- ... [E ”,” {E}] ... [F ”,” {F}] ...
- ... E F ”,” {E F} ...

The c_1-transition in the figure connects the LIST-boxes. It specifies one TFSM transition, from the ”T”-state of the last element in the E-List to the ”I”-state of the first element in the F-list. The c_2-transition connects the actual elements of the lists. It specifies a family of transitions, connecting the ”T”-state of each E-list element with the ”I”-state of each F-list element. Finally, the c_3-transition specification connects the ”a”-state of each E-list element with the ”b”-state of each F-list element.

In the textual representation the references to lists is represented by a special state *LIST* resulting in the following textual representation of the three MVL-transitions.

Term 5:

```
(siblingPath("E", undef, statePath("LIST")),
 c1,
 siblingPath("F", undef, statePath("LIST")))

(siblingPath("E", undef, statePath("T")),
 c1,
 siblingPath("F", undef, statePath("I")))

(siblingPath("E", undef, statePath("a")),
 c1,
 siblingPath("F", undef, statePath("b")))
```

3.4.2 Extension of InstantiateTransitions

The instantiation of transitions involving lists and options can be done by refining the algorithm *InstantiateTransition* of Section 3.3.3 with two cases, one for source nodes being lists and one for target nodes being lists. In both cases the algorithm *InstantiateTransition* is called recursively for each element in the list. In order to make the definition clearer, we assume that the initial values of the global variables are given as four parameters *SourceNode*, *SourceState*, *TargetNode*, and *TargetState*. The header of the algorithm is thus

```
algorithm InstantiateTransition(SourceNode,
                                SourcePath,
                                TargetNode,
                                TargetPath)

variables SourceNode0 <- SourceNode
          SourcePath0 <- SourcePath
          TargetNode0 <- TargetNode
          TargetPath0 <- TargetPath

loop
 ...
```

and in the *loop* part, the source and target paths are simplified as described at the end of Section 3.3. The new cases for list processing are given as follows.

```
if SourceNode0 = list L with more than 2 elements then
  for all elements l in list L
    call InstantiateTransition(l, SourcePath0,
                               TargetNode0, TargetPath0)
if TargetNode0 = list L with more than 2 elements then
  for all elements l in list L
    call InstantiateTransition(SourceNode0, SourcePath0,
                               l, TargetPath0)
```

The processing of the special LIST-states by the algorithm InstantiateTransition has to handle the special cases of NoNode-instances and normal nodes, since as we discussed, only lists with the minimal length two are represented in the AST as actual lists. If a MVL transition targets to a LIST-state of some path, there are thus two possibilities for the instantiation with a TFSM transition:

- If the target node is a list of nodes, the transition is instantiated with a transition going to the ”I” state of the first element in the list.

- Otherwise the transition is instantiated with a transition going to the ”I” state of the target node itself.

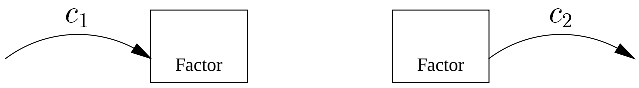
The instantiation of MVL-transition whose source path is a LIST-state is treated correspondingly.

- If the source node is a list of nodes, the transition is instantiated with a transition starting at the ”T” state of the last element in the list.

- Otherwise the transition is instantiated with a transition starting at the ”T” state of the source node itself.

The algorithm InstantiateTransition is now refined with two cases which are checked before the resulting TFSM-transition is generated.

```
if SourcePath0 =˜ statePath("LIST") then
  SourcePath0 := "T"
  if SourceNode0 = list L with more than 2 elements then
    SourceNode0 := last element of L
if TargetPath0 =˜ statePath("LIST") then
  TargetPath0 := "I"
  if TargetNode0 = list L with more than 2 elements then
    TargetNode0 := first element of L
```



(... , ..., c_1, ”I”, n_2) (n$_2$...

Fig. 29: Examples for MVL-Transitions involving global-paths.

Implicit Transitions

A last important aspect of lists and options are *implicit transitions* in the TFSM. Implicit transitions are TFSM-transitions with the default-conditions which are added in order to provide for sequential data-flow in lists, and in order to guarantee, that control flows through the NoNode-instances. For each element in a list, except the last one, an implicit transition with default-condition is added from the ”T”-state of the element, to the ”I”-state of the next element in the list. For each NoNode-instance, an implicit transition from its ”I” to its ”T” state is added.

3.4.3 Global Paths

For certain programming constructs like procedure calls, goto’s, and exceptions we need a way to specify a transition from or to nodes which are not siblings, but ancestors of the Montage. The nesting of boxes with selector functions allows us to access direct and indirect siblings. In order to allow for transitions from or to arbitrary nodes in the AST, we introduce the *global path*. The global path is visualized by a box labeled with a characteristic or synonym symbol. This box represents all instances of said symbol.

Besides the already introduced path operators *siblingPath* and *statePath* we introduce thus a third one called *globalPath*. The parameters of a global path are the name of a characteristic or synonym symbol and a path. Control arrow to or from a global path denote a family of arrows to or from all corresponding instances. As in the case of boxes labeled with selector functions, incoming arrows are connected with the ”I”-state and outgoing arrows are connected with the ”T”-state.

As an example consider again the AST from Fig. 19. A global path *Factor* would refer to nodes 2, 4, and 6 whereas a global path *Sum* would refer to nodes 1 and 3. In this constellation a MVL-transition into a global path *Factor* would denote 3 control arrows ending in the initial states of nodes 2, 4, and 6, a MVL-transition departing from the same global path would denote 3 control arrows departing from the terminal states of nodes 2, 4, and 6. The situation is depicted in Figure 29. A transition targeting and a transition sourcing in a global path Factor is shown, together with the instantiation as TFSM transitions.

In order to process global paths, the algorithm InstantiateTransitions has to be refined again, this time with two cases calling InstantiateTransitions for each instance of a universe. The new cases look as follows:

```
if SourceNode0 =~ globalPath(&Universe, &Path0) then
  for all elements n in universe &Universe
    call InstantiateTransition(n, &Path0,
                               TargetNode0, TargetPath0)
if TargetNode0 =~ globalPath(&Universe, &Path0) then
  for all elements n in universe &Universe
    call InstantiateTransition(SourceNode0, SourcePath0,
                               n, &Path0)
```

3.4.4 Algorithm InstantiateTransition

We have now covered all aspects of InstantiateTransitions and can collect the combine the initial definition and the refinements to the following final version. Since we have not introduced a formal algorithmic notation yet, the code is given in an informal way, referring to well known concepts like calling procedures, updating variables, or ranging over lists. Later in Section 8.4.3, the fully formalized algorithm is given as ASM 57. Interestingly the fully formalized algorithm is neither longer nor more complex.

```
algorithm InstantiateTransition(SourceNode,
                                SourcePath,
                                TargetNode,
                                TargetPath)

variables SourceNode0 <- SourceNode
          SourcePath0 <- SourcePath
          TargetNode0 <- TargetNode
          TargetPath0 <- TargetPath

loop
  if SourceNode0 = list L with more than 2 elements then
    for all elements l in list L
      call InstantiateTransition(l, SourcePath0,
                                 TargetNode0, TargetPath0)
    exit
  if TargetNode0 = list L with more than 2 elements then
    for all elements l in list L
      call InstantiateTransition(SourceNode0, SourcePath0,
                                 l, TargetPath0)
    exit
  if SourcePath0 =~ siblingPath(&Symbol, &Occ, &Path0) then
     let n’ = (selector function (&Symbol, &Occ)
               applied to SourceNode0) in
       SourceNode0 := n’
       SourcePath0 := &Path0
  if TargetPath0 =~ siblingPath(&Symbol, &Occ, &Path0) then
     let n’ = (selector function (&Symbol, &Occ)
               applied to TargetNode0) in
       TargetNode0 := n’
       TargetPath0 := &Path0
  if SourcePath0 =~ statePath("LIST") then
    SourcePath0 := "T"
    if SourceNode0 = list L with more than 2 elements then
      SourceNode0 := last element of L
  if TargetPath0 =~ statePath("LIST") then
    TargetPath0 := "I"
    if TargetNode0 = list L with more than 2 elements then
      TargetNode0 := first element of L
  if SourceNode0 =~ globalPath(&Universe, &Path0) then
    for all elements n in universe &Universe
      call InstantiateTransition(n, &Path0,
                                 TargetNode0, TargetPath0)
  if TargetNode0 =~ globalPath(&Universe, &Path0) then
    for all elements n in universe &Universe
      call InstantiateTransition(SourceNode0, SourcePath0,
                                 n, &Path0)
  else
    let SourcePath0 =~ statePath(&srcS),
        TargetPath0 =~ statePath(&trgS) in

       create TFSM transition
         (SourceNode0, &srcS, Condition, TargetNode0, &trgS)
       exit
```

3.4.5 The Goto Language

As an example language for transitions involving lists and global paths, we give a simple extension of the expression language (S) we introduced in the previous sections. In addition to expressions, the extended language features print, goto, and labeled statements. The new EBNF rules are given as follows.

Gram. 2:	*Prog*	::=	*Statement* “;” { *Statement* }
	Statement	=	*Print* | *Goto* | *Labeled*
	Print	::=	“*print*” *Expr*
	Goto	::=	“*goto*” *Ident*
	Labeled	::=	*Label* “:” *Statement*
	Label	=	*Ident*

In Figure 30 we show two alternative, but equivalent Montages for the Prog-construct. The first solution introduces a list of statement by using a recursive EBNF rule and the square brackets denoting an option. Alternatively the second solution uses the curly list brackets to express directly a list.

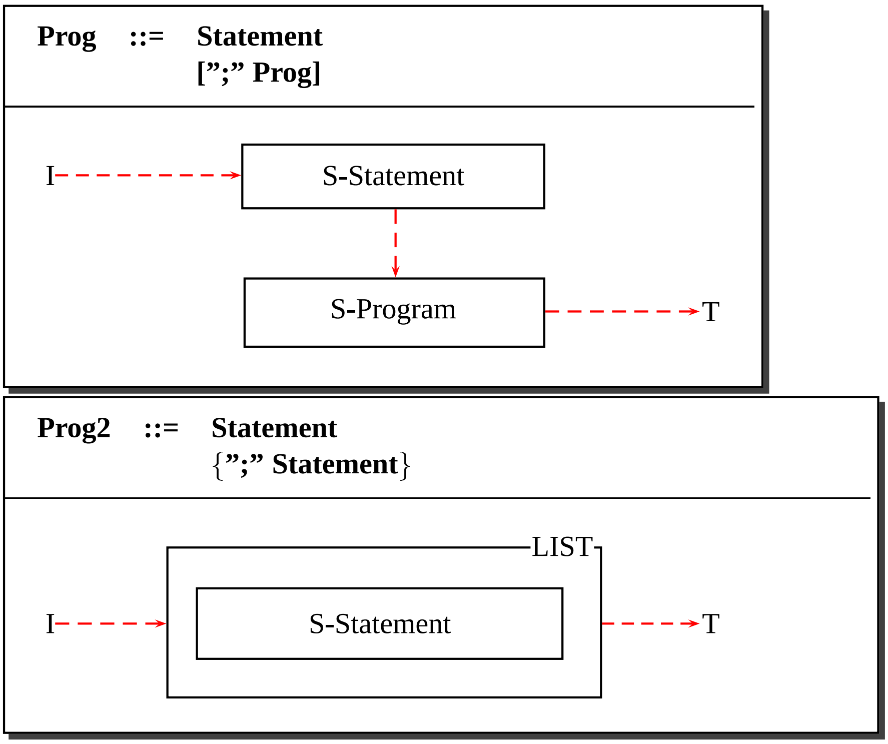


Fig. 30: The Montages Prog and Prog2.

In the first case the sequential control has to be given explicitly, in the second case we use a special box for lists. Such LIST-boxes define as default sequential control flow.

The print statement (Figure 31) fires an action using the XASM syntax for printing to the standard output. Its use is to test the behavior of the other statements. The *Labeled* statement (Figure 31) is composed by a label and a statement. It sends control directly to the statement-part, and has no further behavior attached. *Label* is a simple synonym for an identifier.

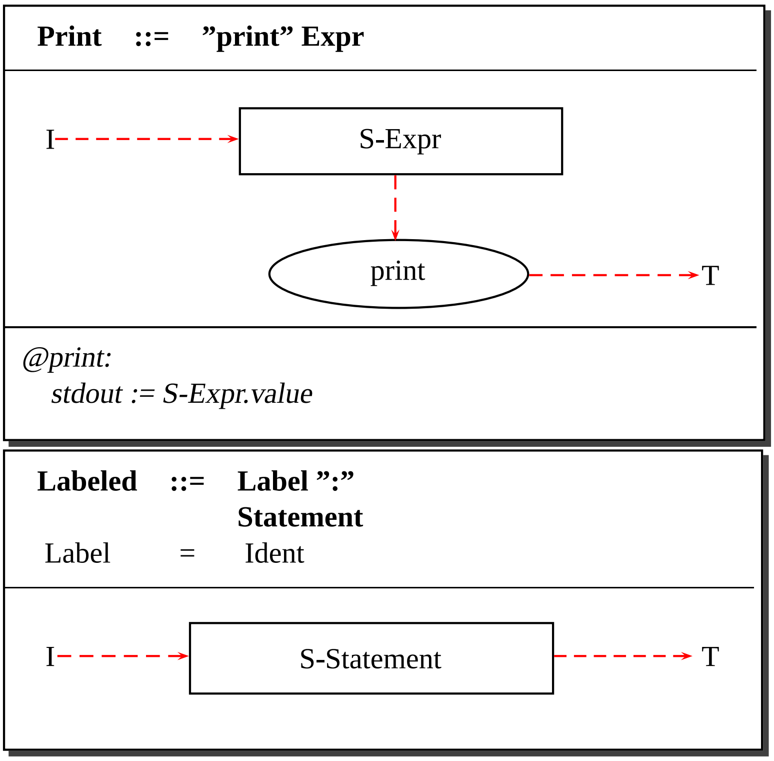


Fig. 31: The Montages Print and Labeled.

The interesting Montage is the *Goto* Montage which is shown in Figure 32. The box labeled with "Labeled" is a global-path referencing all instances of the EBNF-symbol Labeled. The MVL-transition from the go-state to the exit-state within the Labeled reference denotes a family of TFSM transitions from the "go" state going to the "I"-state of each Labeled-statement. The firing-condition

$$trg.\textit{S-Label}.Name = src.\textit{S-Ident}.Name$$

of these transitions depends from the source node *src* and the target node *trg*. The condition guarantees that the label of the target matches the identifier-component of the goto statement. If each label is used only once, this guarantees that the conditions are mutually exclusive for each Goto-instance.

An example program in our language is

```
A: print 1;
   goto  B;
C: goto  A;
B: print 2;
   goto  C;
```

the corresponding states and nodes of the TFSM are given in Figure 33. The result of executing the TFSM is the sequential printing of 1, 2, 1, 2, 1, 2, ...

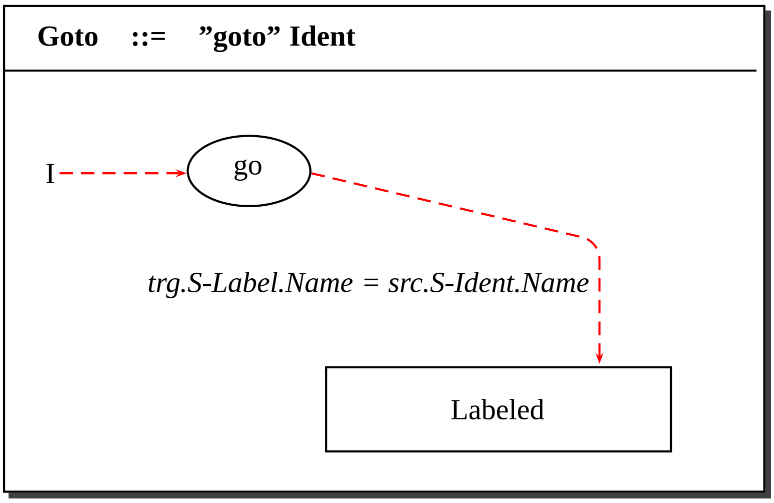


Fig. 32: The Goto Montage.

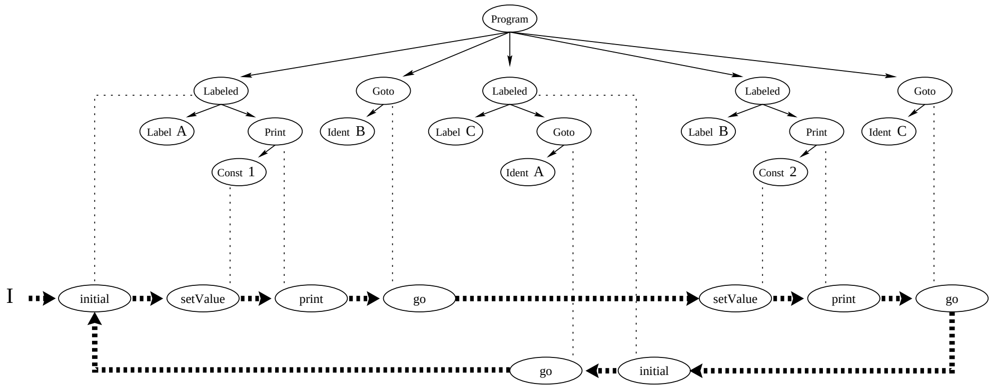


Fig. 33: The nodes and states of the TFSM.

3.5 Related Work and Results

The work on Montages was originally motivated by the formal specification of the C language (85)[6], which showed how the state-based Abstract State Machine formalism (ASMs) (80; 81; 97) is well-suited for the formal description of the dynamic behavior of a full-fledged main-stream programming language. At the risk of oversimplifying somewhat, we can describe some of these models (85; 224; 130) as follows. Program execution is modeled by the evolution of two variables[7] *CT* and *S*. *CT* points to the part of the program text currently in execution and may be seen as an abstract program counter. *S* represents the current value of the store. Formally one defines the *initial state* of the functions and specifies how they evolve by means of *transition rules*.

Some of the ASM models of programming languages assume that the representation of the program's control and data flow in the form of (static) functions between parts of the program text is given. Others like the Occam model described in (27) use ASMs for the construction of the control and data flow graph. All of them use informal pictures to explain the flow graph. These pictures have been refined and formalized as the Montages Visual Language.

3.5.1 Influence of Natural Semantics and Attribute Grammars

Another important experience before the definition of Montages was the use of Kahn's *Natural Semantics* (110) for the dynamic semantics of the programming language Oberon (124). Although we succeeded due to the tool support by Centaur (34), the result was less compact and more complex then the ASM counterpart given by Haussmann and the author in (130); one reason is that one to carry around all the state information in the case of Natural Semantics. An important empirical result of this experiment was the fact that treatment of lists produced a relatively large number of repetitive rules. Therefore the definition of Montages included from the beginning a special treatment of lists, being part of the Montages Visual Language.

The input from the Verifix project (73; 88) has helped to see the necessity of using *attribute grammars (AGs)* (122) for the definition of static semantics. Montages use AGs for the specification of static properties. Among the several mechanisms proposed for defining programming languages, AG systems have been one of the most successful ones. The main reason for this lies in the fact that they can be written in a declarative style and are highly modular. However, by itself they are unsuitable for the specification of dynamic semantics. The work of Kaiser on *action equations* (111; 112) addresses this problem by augmenting AGs with mechanisms taken from *action routines* proposed by Medina-Mora in (151) for use in language based environments. In Appendix A we give a detailed comparison of Montages with *action equations*. Later Poetzsch-Heffter designed the MAX system (184; 185; 186) being the first system taking advan-

[6] Historically the C case-study was preceded and paralleled by work on Pascal (80), Modula2 (157), Prolog (30), and Occam (28).

[7] These variables are called *dynamic functions* in ASM terminology.

tage of combining ASMs with AGs. Further references to MAX will be given in Section 7.3. Action Equations and MAX can be considered as direct predecessors of Montages. In contrast to them Montages is a graphical formalism.

3.5.2 Relation to subsequent work on semantics using ASM

While the Montages approach can be considered as a systematization of the existing ASM descriptions of programming languages (80; 157; 85; 224; 130; 156) a newer thread of ASM specifications is started by Schulte and Börger (33), braking among others with the tradition to using visual descriptions for control flow. This new thread uses a style similar to structural description methods such as Natural Semantics (110) and SOS (182), but the resulting ASM models are isomorphic to the kind of models defined by earlier ASM formulations of programming languages or by a Montages description. The combination of a declarative specification style and a formal model based on abstract syntax trees and control flow graphs can be unintuitive for the experts in structural semantics formalisms, which expect models where programs are formalized as terms, rather than trees, and where control flow is given over the term structure. At the same time the chosen mixture of two different styles make the resulting descriptions unfriendly for programmers, which have typically no background in structural description methods. A more promising approach in this direction is the MAX approach of Poetzsch-Heffter, where parse-trees are formalized as *occurrence algebras* (186), which allows to combine ASMs directly with a structural description method. The work of Poetzsch-Heffter contains as well a precise definition of upwards pattern-matching, which allows to access nodes further up in the tree. A similar technique is used by Schulte and Börger in the form of patterns with *program points* which are ”visualized” by tiny, prefixed Greek letters.

Nevertheless, the new style of language descriptions by Schulte and Börger which has been further elaborated by Stärk for teaching in a theoretical computer science lecture at ETH Zürich (203) has led to an interesting correctness proof of translation from Java to the Java Virtual Machine (204).

As an experiment we have reengineered with Montages a reproduction of the model of the imperative core of Java as given by Stärk. In our reproduction the textual rules are shortened from the original 85 lines to 29 lines, and the complete control flow is specified graphically. The given reproduction can be directly executed using the Gem-Mex tool and has been presented to the students of the ETH classes. Our reproduction of Stärks model is given in Appendix C. In Chapter 14 we show a corresponding state-of-the art Montages description of the same features and explain why our version is better with respect to compositionality.

3.5.3 The Verifix Project

A further systematization of the traditional thread of ASM and Montages descriptions of programming languages has been developed by Heberle et al. (88;

87) in the context of the Verifix project (73; 88) which aims at a systematic approach for provably correct compilers. The Verifix approach uses a variant of Montages for the specification of source languages, and allows to use state-of-the-art compiler technology. The Verifix variant of Montages is a combination of Montage's style for dynamic semantics with traditional well-proven variants of attribute grammars, while our definition of Montages uses a more experimental version of attribute grammars which is described in Chapter 7. Heberle describes a method for correct transformations in compiler-construction and uses the Verifix variant of Montages as formal semantics for the source languages. In order to make the resulting proofs modular and repeatable, he defines the domain-specific language *AL* for giving action rules. *AL* is a specialized version of ASM, resulting from his analysis of existing ASM and Montages specifications of imperative and object-oriented languages. As a result, two independently developed specifications for the same programming language will typically be equivalent, if Heberle's approach is followed, whereas Montages and traditional ASMs allow for many different specifications of the same set of constructs. On the other hand, if domain-specific languages are developed, the approach of Heberle can be more complex than the here presented approach.

The proposal of Heberle can as well be generalized to a new way of structuring language descriptions based on Montages. Instead of using a fixed language such as XASM for defining action rules, one could allow to plug in an arbitrary language. A DSL could then be developed by first defining an action DSL, such as AL, which is used to define action rules in the specification of the final DSL. The interface in order to use one language to define action rules of the specification of another language is relatively lean, in essence providing means to navigate the AST, and to read and write the attributes of the AST. A special case of this language specification structuring mechanism arises if some action rule executes recursively code of the specified language. This case has been implemented in the Gem-Mex tool and used by the author in some of the later referenced industrial case studies.

3.5.4 The mpC Project

Another compiler project using Montages is the *mpC parallel programming environment* (69). Montages in used in this project in two different ways: first, the most sophisticated part of the language, the sublanguage of expressions for parallelization, is modeled using the Gem-Mex environment, second, the obtained formal specification is used for test suite generation (115; 114; 113).

Modeling of mpC expressions in Montages framework helped to find several inconsistencies in the mpC language semantics and gave a lot of useful ideas for the code generation part of the compiler. The Montages specification of mpC expressions is used for three different purposes:

- **Test cases generation.** The static semantics part of the specification (syntax productions, constraints) is used to generate both a set of statically correct, and a set of statically incorrect programs, which constitute a positive and a negative test suite, respectively.

- **Test oracle generation.** The dynamic semantics part of the specification, e.g. the execution behavior, is used for generating trustable output of a test program. The test oracle compares actual and trustable outputs for a particular test case. If the results are not identical the verdict is failure.

- **Providing test coverage criteria.** The specification coverage analysis demonstrates whether all parts of the specification are exercised by the test suite. If the coverage criteria are satisfied then no more test cases are needed, otherwise additional test programs should be added to the test suite. Several coverage Montages-oriented coverage criteria were developed.

With help of the generated test suites the mpC team found more then 30 errors in the current compiler implementation, as a result the quality of the compiler was significantly improved (187). This case study demonstrated that Montages specification is a powerful tool for developing language test suites, which is an important part of the compiler development process.

3.5.5 Active Libraries, Components, and UML

Montages together with the support environment Gem-Mex (9) can as well be seen as an *active library* as defined by Czarnecki and Eisenecker (51). According to the given definition, active libraries extend traditional programming environments with means to customize code for program visualization, debugging, error diagnosis and reporting, optimization, code generation, versioning, and so on. Gem-Mex provides such a meta-environment based on Montages, covering program-visualization, debugging, code generation, and versioning. Another example of an active library is the *intentional programming* system (197; 198). While fixed programming languages (both GPLs and DSLs) force us to use a certain fixed set of language abstraction, active libraries, such as Montages or intentional programming allow us to use a set of abstractions optimally configured for the problem at hand. They enable us to provide truly multi-paradigm and domain-specific programming support.

Unfortunately Microsoft decided to keep details of the intentional programming system confidential, until they release it for commercial use. A direct comparison of Montages and intentional programming must thus be delayed to the official launch of intentional programming. From the existing publications we understand that intentional programming relies on pure transformation approaches for giving dynamic semantics, while Montages make the parse trees directly executable.

The practical experience with Gem-Mex opened early the discussion on the need for a component based implementation of Montages. XASM features a component system, which is used for this purpose. In Denzler's dissertation (55) the use of component technology for Montages is explored in detail, and led to an alternative implementation based on Java Beans.

The disadvantage of Denzler's approach is that it makes it more difficult to realize efficient implementations by means of partial evaluation. Further the low abstraction level of Java w.r.t. XASM may permit less reuse, and it is more difficult to apply formal transformations such as partial evaluation.

Nevertheless we belive that future industrial applications will follow the approach to use a main-stream host language and implement Montages as a pattern for language engineering on top of this language. Actions would be formulated directly in the host language, and the whole abstract syntax tree and tree finite-state machine would be provided as a framework for using the Montages pattern. At the moment we think the emerging executable *Action Language* for UML state machines (2; 229) is the best candidate, especially since it has many similarities with XASM, and since a harmonization of Montages with UML terminology for state machines and actions would allow us to reposition Montages as a tool for *Model Driven Architectures* (25; 170), the OMG group's variant of domain engineering and DSL technology (43; 148).

3.5.6 Summary of Main Results

The following list summarizes the main results of Montages related applications and research.

- The language definition formalism Montages has been defined and elaborated over the last six years. The first version, published by Pierantonio and Kutter in 1996 (131; 133) has been step-wise refined, and simplified since then. Shortly after these publications Anlauff joined the Montages core team.

  The original formulation of Montages was strongly influenced by a case study where the Oberon programming language was specified (130; 132). The earliest case studies outside the Montages team were a specification of SQL by di Franco (58) and a specification of Java by Wallace (225). Other more recent case studies include the use of Montages as a front-end for correct compiler construction in the Verifix project (73; 88), applications of Montages to component composition (13), and its use in the design and prototyping of a domain-specific language (134). These have led to several improvements in the formalism which have been reported in (12).

  The here presented final version of Montages and its semantics has been influenced by a pure XML based semantics description formalism (126), which has been developed by the author for the company *A4M applied formal methods AG* (135).

- Three general purpose programming languages, Oberon (132), Java (225) and C (98), have been specified using Montages. These case studies have led to constant improvements of the tool and methodology such that all three language can now be described easily, with exception of certain syntax-problems. For example we cannot solve the dangling if problem. Another example of syntax-problems is that we need to introduce more explicit naming conventions for classes and variable names in Java. It is fair to say, that Montages can and has been used to specify real-world programming languages, if the syntax (not semantics) is simplified.

The syntax problem can be solved by basing Montages on abstract syntax, as shown in the examples of Appendix A, or by using XML syntax (126).

- As final case study for this thesis, the Java language has been described again. The work of Wallace (225) has shown several deficiencies of Montages, if a language with the complexity of Java is described. Among other improvements, Wallace proposed to replace the original use of data-flow arrows with a much more general mechanism. Nevertheless we decided to replace data-flow arrows completely with AGs, which allows as well to solve the problems found by Wallace. The very detailed work of Wallace has then been partly adopted by Denzler, and later completed to a full Java description by the author.

  The most complex part of Java proved to be the specification of subtyping, name resolution, and dynamic binding. This part of the specification is shown in Appendix D as an example. It must be noted that the limited parsing capability of the current Montages implementation has forced us to introduce explicit syntax for resolving whether an identifier is a class, a method, or an attribute. Therefore one can argue that our specification does not completely cover name resolution.

  Although the length of the resulting Java specification has led to its exclusion from the text, it showed that such a description is feasible. All sequential features of Java have been specified such that they can be used in isolation, and reused in small sub-languages. The complete specification of Java has been split up in a total of fourteen sub-languages. Typically one language extends its predecessors. The extensions are very small, typically two to three new specification modules and half a dozen new definitions, and can often be reused in later stages without adaption.

- A library of reusable language concept descriptions has been elaborated from the new Java case study. This library is presented in Part III of this thesis. The semantic features of major object-oriented GPLs are covered in principle by these components and a full object-oriented language can be described by combining and adapting them. In fact, the library is structured again as a number of small languages, reusing each others specification modules.

  It will be difficult to model the exact syntax and semantics of other existing object-oriented GPL such as C++ without further adapting the library but for our purpose of having building blocks of GPL concepts reusable for DSL designs the library is very useful.

- Several DSLs have been developed with and applied by different industrial partners. The executed case studies are

  - The design and implementation of the data model mapping language *CML* for the bank UBS (134). This work has been done jointly with Lothar Thiele and Daniel Schweizer.

  - The specification and implementation of the hybrid system descriptions language *HYSDEL* for the Automatics Institute at ETH Zurich (6). This work has been done jointly with Samarjit Chakraborty.
  - The design and implementation of three DSLs for a financial analysis generation software system of a small financial service provider. These languages have been shortly described at the end of Section 2.2 and are currently in productive use at one of Switzerland's largest banks.
  - The specification and implementation of the SMS application language *Eclipse* for the company Distefora Mobile.

  The last two case studies have been executed by the author and Matthias Anlauff for A4M AG.

- Besides GPLs and DSLs the basic notation of another language description formalism called *Action Semantics* (158) has been described (7). This work has been done jointly with Lothar Thiele and Samarjit Chakraborty.

- The imperative prototyping language XASM (5) has been designed, implemented, and tested by Anlauff and the author for the company A4M applied formal methods AG which is supporting and further developing the language under an open source license (8). XASM is a generalization of the mathematical Abstract State Machine (ASM) formalism. XASM is used not only for the definition of semantic actions but for the formalization and implementation of the complete Montages approach.

  The initial, non-formal definition of XASM by Anlauff has now been formalized by the author, and a number of additional features and reusable techniques have been developed. The formalization and the newly designed features are presented in Chapter 4. Further a pure object oriented version of XASM has been developed and specified by the author and an executable Montages description of this new language can be downloaded (128).

- XASM has been used by Anlauff as DSL for the implementation of the Montages tool support *Gem-Mex*[8]. Gem-Mex allows the language designer to generate for each specified language an interpreter, a graphical debugger, and language documentation (10). The design of these tools has been driven by the case studies. The use of XASM for the implementation allowed a quick adoption of the environment to changes. Further the author has been able to influence the development of Gem-Mex on the XASM level, without knowing the details of the underlying C-code.

  By using the DSL XASM to implement the language description formalism Montages (respectively its tool set Gem-Mex), the development process of our team is a refined version of the three cycle process (Section 2.6, Figure 7). In

[8] The current Gem-Mex implementation has been preceeded by work of Sèmi (193) on using Centaur for the tool support of Montages, and by a first Montages implementation based on Sather.

fact our process is a four-cycle process, resulting as a combination of the three cycle process with the two-cycle process (Figure 6), which are both embedded in our actual development process.

  - The two-cycle process is built by the GPL *C* which we use to develop the DSL XASM, which in turn is used to develop the application Gem-Mex.
  - The three-cycle process is overlapping these cycles: XASM is considered the GPL used to develop the language description formalism Montages, which is then used to develop an arbitrary DSL, which is used to develop applications.

  With other words, in the two-cycle aspects of our development process, Gem-Mex is considered the resulting application, and in the three cycle aspect, the very same software, also know as Montages, is considered as the language description formalism, being the central building block of the three cycle process. Our four cycle process is visualized in Figure 34.

- Both the Montages meta-formalism, and the XASM formalism have been specified and tested using Gem-Mex. The Gem-Mex meta-formalism description of Montages has been partly derived from a description of an XML based meta-formalism developed by the author for A4M Applied Formal Methods AG (126). The Montages- and XASM-implementations generated by Gem-Mex from their Montages descriptions are fully functional, but cannot compete yet with hand written implementations. Their main purpose at the moment is the documentation of the design process of Montages and XASM.

  In this thesis, an alternative XASM definition of Montages is given in Chapter 8. This new semantics is specially designed to allow for a relatively efficient implementation by means of partial evaluation.

  In parallel we work on using Montages for bootstrapping XASM in the context of the XASM open-source project. The bootstrapping process for XASM is visualized in Figure 35.

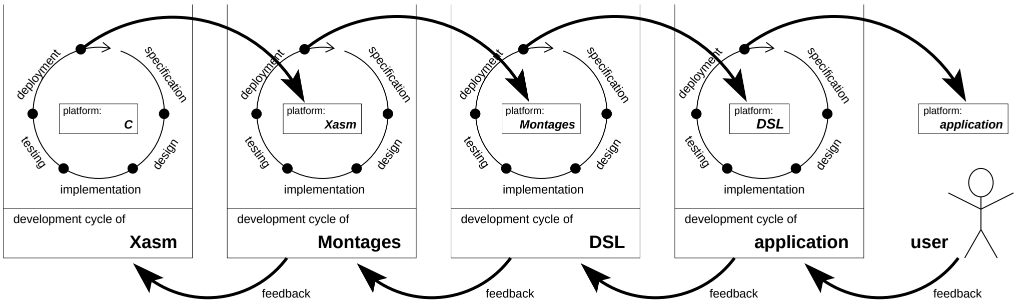


Fig. 34: The four development cycles of the Montages team

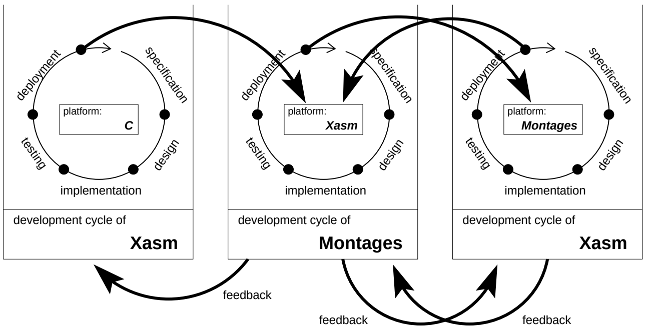


Fig. 35: The bootstrapping of XASM

Part II

Montages
Semantics and System Architecture

In the first part we discussed requirements for language definition formalisms, and introduced our language definition formalism Montages trying to fulfill the formulated requirements. The requirements discussed in Part I are all related to the needs of DSL designers, implementors, and users. As a consequence we have been able to report positive results about usability and expressivity of our approach.

On the other hand, discussions with software developers and system engineers in the financial industry and in networking companies showed that our approach needs to fulfill various requirements related to the form, transparency, and quality of the resulting code, if it ever should have a chance for serious industrial applications, let alone for entering main-stream technologies. In other words, it is not enough to deliver a DSL with a very simple design. The developers which are responsible to support the DSL for the domain experts expect that not only the DSL is easy to understand and maintain, but as well the generated code.

It is difficult to explicitly formulate these kinds of requirements, since they will largely depend on the environment in which the code is going to be used. In order to be able to meet as many as possible of the possible requirement which will show up in concrete situations, our approach should allow

- to influence the structure of the generated code,
- to influence the naming of identifiers in the generated code, and
- to clean the code from those parts which are only needed to make the approach general, but are not relevant or used in a concrete situation.

As example, assume a DSL which features global variables and updates, and where `x := x + 1` is an admissible program. The developers require that the system generates the code they are expecting: `x := x + 1` At least for simple examples they need this kind of ”validation”, indicating whether the system is doing what they expect. As indirect requirement simple language descriptions and simple programs, such as the above `x := x + 1` should result in simple generated code. The current implementation of Montages (10) generates for each specified DSL an interpreter, the complexity of the generated code is therefore independent of the complexity of the DSL programs.

In order to improve the current implementation we are going to develop in this part a formal, executable semantics of Montages which serves directly as building block for a new system architecture. For the formalization of the semantics as well as for the other parts of the system architecture we use the ASM-language XASM which is described in Chapter 4. For the sake of simplicity we abstract from the problem of implementing XASM and present everything on the level of XASM assuming that a transparent and relatively efficient implementation of XASM exists[9].

[9] The XASM Open Source project www.xasm.org is working on XASM implementations.

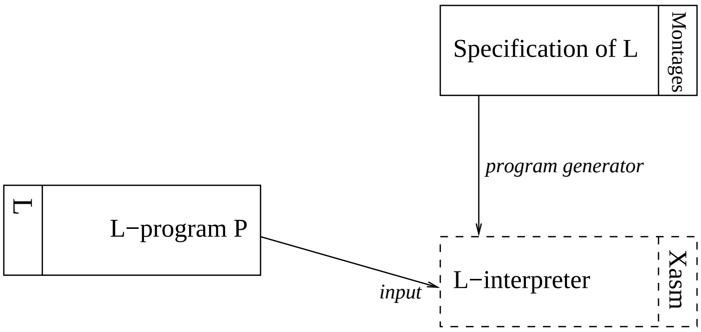


Fig. 36: Current Architecture of Montages System

The here presented system architecture replaces the current implementation, where the specification of a language L, written with Montages, from which a program generator creates an L-interpreter. In Figure 36 these components are visualized, the generated interpreter is represented with a dashed box, and the user supplied language specification and program are solid line boxes. The interpreter works as usual, taking as input an L-program P which is then executed. The program P does not influence the complexity of the generated interpreter. As stated above, the resulting problem is that we cannot expect simple code for simple programs.

The current program generator is further designed as a proof-of-concept for the feasibility of complex Montages description, such as the description of general purpose languages. The implementation has not been tuned towards simplification of the generated code, and the generated interpreters are relatively complex, independent of the complexity of the described language.

For our new architecture we developed with XASM a meta-interpreter of Montages, reading both a specification of a language L (syntax and semantics) and a program P written in the described language L, parsing the program according to the given syntax-description, and executing the program according to the given semantics description. By assuming that the language specification is fixed, we can partially evaluate (46) the meta-interpreter to a specialized interpreter of the specified language. Assuming in addition that the program is fixed, we can further specialize the interpreter into code implementing the program. In Figure 37 the specification of L, written in Montages and the program P, written in L, are shown as boxes on the left side. Both the L specification and P are input to the meta-interpreter which is written in XASM, and visualized on the right side. The box below the meta-interpreter is a L-interpreter, obtained by partially evaluating the meta-interpreter assuming that the specification of L does not change. The L-interpreter box is dashed, showing that it has been generated by the system, rather than provided by the user. As usual, the interpreter takes as input the program P and executes it. Finally, from the interpreter a specialized P-implementation is obtained by partially evaluating the interpreter, assuming that the program P is not changing. Again the box is

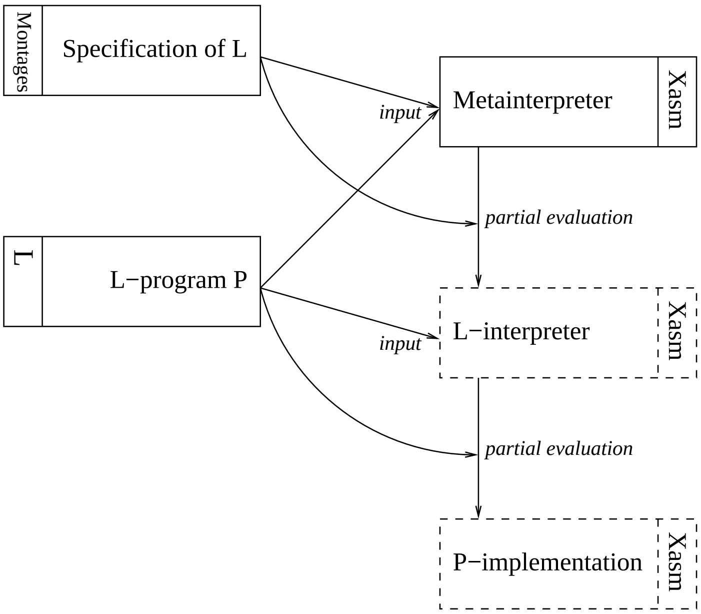


Fig. 37: New Architecture of Montages System

drawn with dashed lines, since it does not have to be provided by the user. The detailed definition of the meta-interpreter is given in Chapter 8. A more detailed sketch of the partial evaluation process is given in Chapter 5.

Using only partial evaluation would create the problem that the generated code inherits the more abstract signature of the language-specification level. As an example consider again a DSL with global variables and destructive updates. The syntax of an assignment may be given as:

```
Assignment ::= Ident ":=" Expression
```

and the semantics of the construct is given by an action in the XASM language. We refer to the micro-syntax of the global variable as `S-Ident.Name` and to the value of the previously evaluated expression as `S-Expression.value` Assuming a hash-table *Global(_)* which holds the values of global variables, the following XASM rule gives the semantics of the Assignment feature:

```
Global(S-Ident.Name) := S-Expression.value
```

Obviously, even if the variable and the expression partially evaluate to the values of the initial example, ”x” and ”x + 1”, the generated code will never be simpler than

```
Global("x") := Global("x") + 1
```

In order to achieve the desired outcome, we need to *parameterize the signature* of the semantics rule. We extended our formalism such, that the signature of variables and functions can be given by a string-value in $-signs. This variant of

XASM is called parameterized XASM (PXasm) and is introduced in Chapter 5. In our example, we can now use a global variable with parameterized signature, rather than the hash-table. The new semantics of the Assignment feature is now:

```
$S-Ident.Name$ := S-Expression.value
```

On the left hand side the $-signs are used to refer to a global variable whose signature is given by the expression `S-Ident.Name`. Once the value of `S-Ident.Name` is fixed to ”x” the left-hand side can be simply specialized to global variable ”x”, and the code generated for our initial example is now the desired

```
x := x+1
```

In Section 11.1 the detailed Montages semantics of an example language *ImpV2* having this semantics is presented, and we invite the reader to consult this section for further details about our above example showing why not only partial evaluation but as well parameterized signatures are needed for our new architecture.

Combining partial evaluation and parameterization of signature results in a techniques which works similar to template languages used for program generation (44; 45). In our case the actual “generation” of the program happens only if the partial evaluation results in a complete evaluation of the signature-parameters, whereas in traditional template languages the content of the templates can always be evaluated. Further our parameterization of signature is integrated with our development language XASM in such a way, that programs can be executed even if partial evaluation did not completely evaluate the parameterized signature. In contrast, unevaluated templates are typically not valid programs.

Another advantage of the new architecture is that the fixed meta-interpreter is much easier to test and maintain than the original interpreter generator. In the software development process of Gem-Mex, as visualized in Figure 34, the maintenance of the generator showed to be the most difficult part, since it was difficult to test whether the generator is really implementing the semantics of Montages. In contrast the meta-interpreter written in XASM is very compact and serves both as semantics and implementation, there is thus no problem of mismatch between semantics and implementation. Although execution the meta-interpreter is far too slow for real applications, it can still be used to test. Once a problem is solved successfully with the meta-interpreter, one has confidence on the functionality of the system. The result of the partial-evaluation can then be tested against the existing reference implementation given by the meta-interpreter.

Further we found that the partial evaluator gives us a lot of freedom to identify variable and static aspects of a system in a late stage, or even dynamically. We can choose freely which parts of the system should be interpreted, allowing them to be changed dynamically, and which parts are partially evaluated, resulting in specialized code. In Section 9.1.2 we show for instance how Montages can be specialized and transformed using partial evaluation. The traditional choices of DSL interpreter or DSL compiler are only special cases of the possible choices: they assume that the language specification is fixed. In some cases it is beneficial to leave part of the language specification interpreted, or to assume part of the program input to be fixed. Often the partial evaluator must be called at run-time, for instance after a number of configuration files are read.

The following chapters are building up the tools which are needed to define the new system architecture in a formal way. In Chapter 4 we introduce the specification language XASM, in Chapter 5 XASM is extended with features allowing for parameterized signature and partial evaluation, in Chapter 6 we apply the introduced techniques to simplify and compile TFSMs, in Chapter 7 the kind of attribute grammars used by Montages is formalized, and finally in Chapter 8 we give the Montages meta-interpreter serving in the new architecture both as semantics and implementation of Montages.

4

eXtensible Abstract State Machines (XASM)

*eXtensible Abstract State Machines (*XASM*)* (4; 11; 5) has been designed and implemented by Anlauff as formal development tool for the Montages project. Recently XASM has been put in the open source domain (8). Unfortunately a formal semantics of XASM has not been given up to now. We streamline Anlauff's original design and present a denotational semantics, complementing the existing informal description. In fact we found that XASM implement a semantic generalization of Gurevich's *Abstract State Machines (ASMs)* (79; 80; 81; 82). The initial idea for this generalization came from May's work (150) which is the first paper formalizing sequential composition, iteration and hierarchical structuring of ASMs. May notes that his approach complements

> .. the method of refining Evolving Algebras[1] by different abstraction levels (31). There, the behavior of rules performing complex changes on data structures in abstract terms is specified on a lower level in less abstract rules, and the finer specification is proven to be equivalent. For execution, the coarser rule system is *replaced* by the finer one. In contrast, in the hierarchical concept presented here, rules specifying a behavior on a lower abstraction level are encapsulated as a system which is then *called* by the rules on the above level. (150), Section 6, page 14, 29ff

XASM embeds this idea in the form of the "XASM call" into a realistic programming language design. The XASM call allows to model recursion in a very natural way, corresponding directly to recursive procedure calls in imperative programming languages. Arguments can be passed "by value", part of the state can be passed "by reference", the "result" of the call is returned as value allowing for functional composition, and finally the "effects" of the called machine

[1] Evolving Algebras is the previous name of ASMs.

are returned at once, maintaining the referential transparency property of non-hierarchical ASMs. Börger and Schmidt give a formal definition of a special case of the XASM call (32) where sequentiality, iteration, and parameterized, recursive ASM calls are supported.

In their framework a so called "submachine" is not executed repeatedly until it terminates, but only once. The XASM behaviour of repeated execution can be simulated by explicit sequentiality, but unfortunately they are excluding the essential feature of both Anlauff's and May's original call to allow returning from a call not only update sets, but as well a value. This restriction hinders the use of their call for the modeling of recursive algorithms. Of course one could argue again, that returning a result from their "submachine" call can be simulated by encoding the return value in some global function, but the essence of ASM-formulations is to give a "direct, essentially coding free model" (81).

The full XASM call leads to a design where every construct (including expressions and rules of Gurevich's ASMs) is denoted by both a value and an update set. This is a generalization of Gurevich's definition of ASMs, where the meaning of an expression is denoted by a value and the meaning of a rule is denoted by an update set (82).

In the context of this thesis, XASM are used for defining actions and firing conditions of the Montages formalism, and the XASM extensions defined in later chapters will be used to give formal semantics to Montages. In Section 4.1 ASMs are introduced from a programmer's point of view looking at them as an imperative language, which can be used to specify algorithms on various abstraction levels. The denotational semantics of ASMs, as defined by Gurevich (82), is given in Section 4.2[2]. Based on a unification and generalization of this semantics, the XASM extension of ASMs is motivated and formalized in Section 4.3. The complete XASM language is a full featured, component based programming language. The features of a pure functional sublanguage of XASM, including constructor terms, pattern matching, and derived functions are given in Section 4.4, and the support for parsing in XASM is described in Section 4.5. Finally, in Section 4.6 we discuss related work.

[2] The formalization of choose and extend chosen by Gurevich (82) are not standard denotational semantics and it may be argued that they are ambiguous. An inductive definition can solve this problem, but we wanted to build our definitions on Gurevich's original formulation.

4.1 Introduction to ASM

ASMs are an imperative programming language. An imperative program is built up by statements, changing the state of the system, given by the current values of data-structures. A data structure is an abstract view of a number of storage locations. Typical Examples of data-structures are variables, arrays, records, or objects. Execution of statements results in a number of read and write accesses to visible and hidden storage locations. The higher the abstraction level of an imperative programming language, the more happens behind the scene for each statement. Ousterhout analyzes the increase of work done per statement for imperative languages of different abstraction levels, starting from machine languages, over system programming languages, reaching up to scripting languages (173). On average, each line of code in a system programming language such as C or Java translates to about five machine instructions, which handles directly read and write accesses to the physical machines. Scripting languages, such as Perl (223; 222), Python (219; 146), Rexx (71; 169), Tcl (172; 171), Visual Basic (which was ”created” as a combination of Denman’s MacBasic and Atkinson’s HyperCard (67)), and the Unix shells (145) feature statements which execute hundreds or thousands of machine instructions.

4.1.1 Properties of ASMs

Unlike the statements of the mentioned programming languages, ASM statements are not executed sequentially, but in parallel. It is therefore difficult to compare ASMs with these formalisms, or to fit them in Ousterhout’s taxonomy. Rather than triggering a number of sequential steps of a given physical machine, the parallel rules define a new, tailored abstract machine. Therefore ASMs are very well suited to describe semantics of programming systems on various abstraction levels. The parallel execution of ASM statements allows to bundle an arbitrary amount of functionality into one state-transition of a system. In traditional imperative languages, regardless of whether they are machine, system, or scripting languages, the amount of work done in one step is fixed by the functionality of the statements featured by the language. In ASM it is therfore relatively easy to tailor a parallel block of statements, whose repeated execution results in a run of states corresponding exactly to the states of the algorithm to be modeled.

Another important difference of ASMs with respect to the mentioned imperative formalisms is the absence of specialized value-types, data-structures, and control-statements. In ASMs there exist no integer, real, or boolean as value-types; the usual variables, arrays, or record data-structures are missing; neither while, repeat, nor loop statements are available. Instead the following solutions are chosen in the ASM formalism:

value-types ASMs feature only one type of value, the *elements*. A typical implementation of ASMs provides a number of predefined elements, like numbers, strings, booleans, as well as elements which can be at runtime created using the *extend* construct of ASMs; examples for such dynamically created elements include objects, as well as abstract storage locations. All of them are considered being elements.

data-structures

ASM feature a unique, universal, n-dimensional data structure that corresponds to an n-dimensional hash table. This data-structure is called n-ary *dynamic function*. A dynamic function *f* can be evaluated like a normal function,

$$f(e_1, \dots e_n)$$

where $e_1 \dots, e_n$ are ASM elements. However it can also be updated,

$$f(e_1, \dots e_n) := e_0$$

where e_0 denotes the new value of *f* at the point $(e_1, \dots e_n)$. The resulting definitions of the dynamic functions represent the state of an ASM, similar to the way how values of variable, arrays, and records represent the state of an imperative program. The single locations, consisting of function name and argument tuple can as well be considered to be the storage locations of an underlying abstract machine.

0-ary functions are used to model variables, unary functions are used to model arrays and records. A set or *universe* is modeled by its characteristic function, mapping all members of the universe to *true*, and all other elements to *false*. Functions mapping all arguments either to *true* or *false* are called *relations*. Universe is a synonym of unary relation.

control-statements

Instead of explicit loop or iteration constructs an ASM program is automatically repeated until it terminates. Termination condition is a fix point of state changes, i.e. if a rule generates no more updates, it terminates.

To control the repeated execution of an ASM rule modeling an algorithm, ASMs feature an *if-then-else* statement, allowing to execute statements conditionally, and a number of statement-quantifiers, allowing to construct sets of statements depending on the current state.

While these features look exotic for most programmers, they have shown to be useful in our context. Programming an algorithm in ASM allows to concentrate on the conceptual structure of the state, and the evolution of that state in a granularity which is completely controllable. Gurevich proves, that every sequential algorithm can be modeled by an ASM which makes exactly the same steps as the modeled algorithm is intended to do (84). The last property has been formulated in the ASM-thesis (79), and a large number of case studies have been elaborated for giving evidence to the thesis, not only with respect to sequential, but as well with respect to distributed algorithms. A summary of all case studies has been published (29) and further discussion of related work is found in Section 4.6.

4.1.2 Programming Constructs of ASMs

ASM statements are built by six different rule constructors.

Update Rule
The basic *update rule* is used to redefine an n-ary function at one point. Given the rule

```
f(t1, ...., tn) := t0
```

first the terms *t0, ..., tn* are evaluated to elements $e0, ..., en$, and then the function *f* is redefined such that in the next state

$$f(e1, ..., en) = e0$$

holds. Please note that the equation $f(t1, ..., tn) = t0$ may never hold, since in parallel to the given redefinition of *f*, the functions used to build terms *t0, ..., tn* may be redefined as well, such that in the next state they evaluate to different elements. For instance the rule

```
x := x + 1
```

will never result in a situation where $x = x + 1$ holds. But if before the execution of the rule $x = n_0$, then after the execution $x = n_0 - 1$ holds.

Parallel Composition
ASM rules are composed in parallel. There are thus no intermediate states, if a block of ASM code is executed, and the order of ASM statements in the block does not influence the behavior. Further the same expression has the same value, independent where in the block it appears. This property is known as *referential transparency (RT)* from functional programming. If a language has RT, then the value of an expression depends only on the values of its sub-expressions (and not, for instance, on how many times or in which order the sub-expressions are evaluated). These properties influence considerably the style of the resulting descriptions.

The standard example showing the effect of parallel composition of rules is the following swap of two variables[3] *x* and *y*.

```
x := y
y := x
```

If this rule is executed, the values of *x* and *y* are exchanged. In contrast to sequential programming languages, there is no need to use a help variable, as done in the following minimal sequential version:

```
tmp := x;
x   := y;
y   := tmp;
```

Unlike the sequential version, the above parallel rule will never terminate, since it updates *x* and *y* in each step, and a state fix-point is thus never reached. The following example shows a terminating parallel rule.

[3] Variables are 0-ary functions in ASM terminology.

Consider the situation, where we have three variables, x_1, x_2, and x_3. All of them are initially set to the value *undef*. In each step of the algorithm, x_1 takes the value 1, x_2 takes x_1's value of the previous step, and x_3 takes x_2's value of the previous step. It will thus take three steps, until the value 1 is propagated to x_3. The ASM program *AP* corresponding to our algorithm is

ASM 1:

```
asm AP is
  functions x1, x2, x3

  x1 := 1
  x2 := x1
  x3 := x2
endasm
```

The variables are declared as dynamic functions with arity 0. By default, at the beginning all dynamic functions evaluate to *undef*. The requirements how values are propagated are directly expressed as the three parallel updates.

After the first step of *AP*, x_1 equals 1, but the remaining functions still equal *undef*. After the second step, both x_1 and x_2 equal 1, but x_3 is still *undef*. After the third and all following steps, all three functions evaluate to 1. The system terminates after the fourth step, since the state of the system is no more changing, e.g. a fixed point has been reached.

Consistency

At this point we would like to raise the issue of inconsistent rules. If the same variable is updated to different values in parallel, for instance by the rule `x := 1 x := 2`, then an inconsistent state is reached and the calculation is aborted. Throughout the thesis we assume consistent rules, although it has to be noted that in general consistency of a rule cannot be guaranteed statically.

Conditional Rules

The conditional rule allows execution to be guarded with predicates. One special application of the conditional rule is to model sequential execution with ASMs. Typically a 0-ary function is used to model an abstract program counter *mode*. For instance the following sequential algorithm

```
var   x = 1, length = 10
array a, f

1  x    := x + 1;
2  a(x) := f(x);
3  if x < length goto 1
4  end
```

can be modeled as the following ASM.

ASM 2:

```
asm ModeTest is
  functions mode <- 1,
            x, length a(_), f(_)

  if     mode = 1 then
    x := x + 1
    mode := 2
  elseif mode = 2 then
    a(x) := f(x)
    mode := 3
  elseif mode = 3 then
    if x < length then
      mode := 1
    else
      mode := 4
    endif
  endif
endasm
```

In fact most ASMs given in the literature follow more or less this pattern to model sequentiality. The advantage of ASMs is, that they allow us to abstract from low level intermediate steps. In typical ASM applications the number of sequential steps is relatively small and therefore the presented solution is acceptable.

Do-for-all Rules

The do-for-all rule allows to trigger an ASM rule for a number of elements contained in a universe and fulfilling a certain predicate. Given a universe *U* containing three elements *e1, e2, e3* and the predicate *Q* over the dynamic functions and the bound variable *u*, the rule

```
do forall u in U: Q(u)
  f(u) := 3
enddo
```

corresponds exactly to

```
if Q(e1) then
  f(e1) := 3
endif
if Q(e2) then
  f(e2) := 3
endif
if Q(e3) then
  f(e3) := 3
endif
```

where *Q(e)* is *Q(u)* with the bound variable *u* replaced by the element *e*.

As a further do-forall example, consider a generalization of algorithm *AP* (ASM 1) to n variables instead of three. We number the variables and use a unary dynamic function *x(_)* mapping the number of a variable to its value. This corresponds to an array of variables. To trigger the updates, we use a rule quantifier, triggering the update `x(i - 1) := x(i)` for each *i* ranging from 2 to *n*. The argument *n* is passed as parameter to the ASM that looks as follows.

ASM 3:

```
asm AP'(n) is
 function x(_)

 do forall i in Integer: i >= 2 and i <= n
   x(i-1) := x(i)
 enddo
```

This algorithm will terminate after n steps.

Choose Rules

The choose rule works similar to the do forall rule, but the rule is only instantiated once for an element of the universe fulfilling the predicate. The *ifnone* clause of the choose-rule allows to give an alternative rule, if there is no such element.

Given again a universe *U* containing three elements *e1*, *e2*, *e3* and the predicate *Q* over the dynamic functions and the bound variable *u*, the rule

```
choose u in U: Q(u)
  f(u) := 3
endchoose
```

corresponds to the empty rule, if neither *Q(e1)*, *Q(e2)*, nor *Q(e3)* holds, otherwise to the rule

```
  f(e) := 3
```

where *e* is nondeterministically chosen from those elements in *U* for which *Q* holds, e.g. from $\{u|u \in U \land Q(u)\}$.

As an example consider a situation where messages have been collected in a universe *MessageCollector*. A predicate *ReadyToProcess(_)* decides which of these messages can be processed. Processed messages are removed from universe *MessageCollector*. Please remember that universes are modeled by their characteristic function. An element *e* is therefore removed from the declared universe by the rule `MessageCollector(e) := false`. If there is no message remaining to be processed, the function *mode* is set from its initial value *undef* to ”ready”. For simplicity we give no details on *Process(_)* and predicate *ReadyToProcess(_)*.

ASM 4:

```
asm ProcessMessages
is
  universe MessageCollector
  function mode

  ...

  choose m in MessageCollector: ReadyToProcess(m)
    Process(m)
    MessageCollector(m) := false
  ifnone
    mode := "ready"
  endchoose
endasm
```

Extend Rules

Extend rules allow us to introduce new elements. The rule

```
  extend C with o
    x := o
  endextend
```

extends a universe *C* with a new element. This element is accessible within the *extend*-rule as bound variable *o*. The element is implicitly added to *C* by triggering `C(o) := true`. Further in the example, the new element is assigned to variable *x*. Intuitively this corresponds to a

$$x := new\ C$$

statement known from object oriented languages.

These examples only give a rough overview of the existing programming constructs in ASM. The detailed definition and formal semantics are given in the next section.

4.2 Formal Semantics of ASMs

The mathematical model behind an ASMs is that a state is represented by an algebra or Tarski structure (207) i.e. a collection of functions and a universe of elements, and state transitions occur by updating functions point wise and creating new elements. Of course not all functions can be updated. The basic arithmetic operations (like add, which takes two operands) are typically not redefinable. The updatable or *dynamic functions* correspond to data-structures of imperative programming languages, while the static functions correspond to traditional mathematical functions whose definition does not depend on the current state. All functions are defined over the set $\mathfrak{U}$ of elements. In ASM parlance $\mathfrak{U}$ is called the *superuniverse*. This set always contains the distinct elements *true*, *false*, and *undef*. Apart from these $\mathfrak{U}$ can contain numbers, strings, and possibly anything – depending on what is being modeled. Subsets of the superuniverse $\mathfrak{U}$, called *universes*, are modeled by unary functions from $\mathfrak{U}$ to *true*, *false*. Such a function returns *true* for all elements belonging to the universe, and *false* otherwise. The universe *Boolean* consists of *true* and *false*. A function *f* from a universe *U* to a universe *V* is a unary operation on the superuniverse such that for all $a \in U$, $f(a) \in V \cup \{undef\}$ and *f(a) = undef* otherwise.

Functions from Cartesian products of $\mathfrak{U}$ to Boolean are called *relations*. By declaring a function as a relation, it is initialized for all arguments with *false*. A universe corresponds to a unary relation. Both universes and relations are special cases of functions. The dynamic functions not being relations are initially equal to *undef* for all arguments.

Formally, the *state* λ of an ASM is a mapping from a signature Σ (which is a collection of function symbols) to actual functions. We use f_λ to denote the function which corresponds to the symbol *f* in the state λ.

As mentioned above, the basic ASM *transition rule* is the update. An update rule is of the form

$$f(t_1, \ldots, t_n) := t_0$$

where $f(t_1, \ldots, t_n)$ and t_0 are closed terms (i.e. terms containing no free variables) in the signature Σ. The semantics of such an update rule is this: evaluate all the terms in the given state, and redefine the function corresponding to *f* at the value of the tuple resulting from evaluating $(t_1, \ldots, t_n)$ to the value obtained by evaluating t_0. Such a point wise redefinition of a function is called an *update*. Rules are composed in a parallel fashion, so the corresponding updates are all executed at once. Apart from the basic transition rule shown above, there also exist *conditional* rules, *do-for-all* rules, *choose* rules and lastly *extend* rules. Transition rules are recursively built up from these rules. The semantics of a rule is given by the set of updates resulting from composing updates of rule components. This so called *update denotation* of rules is formalized in the following.

Def. 1: **Update denotation.** *The formal semantics of a rule* R *in a state* λ *is given by its update denotation*

$$\mathit{Upd}(R, \lambda)$$

which is a set of updates.

The resulting state-transition changes the functions corresponding to the symbols in Σ in a point wise manner, by applying all updates in the set. The formal definition of an update is given as follows.

Def. 2: **Update.** *An update is a triple*

$$(f, (e_1, \ldots, e_n), e_0)$$

where f *is a* n*-ary function symbol in* Σ *and* $e_0, \ldots, e_n$ *are elements of* $\mathfrak{U}$*.*

Intuitively, firing this update in a state λ changes the function associated with the symbol *f* in λ at the point $(e_1, \ldots, e_n)$ to the value e_0, leaving the rest of the function (i.e. its values at all other points) unchanged. Firing a rule is done by firing all updates in its update denotation.

Def. 3: **Successor state.** *Firing the updates in Upd*(R, λ_i) *in the state* λ_i *results in its successor state* λ_{i+1}*. For any function symbol f from* Σ*, the relation between* f_{λ_i} *and* $f_{\lambda_{i+1}}$ *is given by*

$$f_{\lambda_{i+1}}(e_1, \ldots, e_n) \;=\; \begin{cases} e_0 \quad if \quad (f, (e_1, \ldots, e_n), e_0) \in \mathit{Upd}(R, \lambda_i) \\ f_{\lambda_i}(e_1, \ldots, e_n) \quad otherwise \end{cases}$$

There are two remarks concerning this definition. First, if there are two updates which redefine the same function at the same point to different values, the resulting equations are inconsistent, and the next state $f_{\lambda_{i+1}}$ cannot be calculated. Consistency of rules cannot be guaranteed in general, and an inconsistent rule results in a system abort.

The second remark is about completeness of the successor-state relation. The above complete definition of the next state (Definition 3) could be relaxed to a partial definition as follows:

Def. 4: **Partial successor state.** *Firing the updates in Upd*(R, λ_i) *in the state* λ_i *results in its successor state* λ_{i+1}*. For any function symbol* f *from* Σ*, the relation between* f_{λ_i} *and* $f_{\lambda_{i+1}}$ *must be a model for the following equations:*

$$f_{\lambda_{i+1}}(e_1, \ldots, e_n) \;=\; e_0 \quad if \quad (f, (e_1, \ldots, e_n), e_0) \in \mathit{Upd}(R, \lambda_i)$$

The advantage of the partial definition is that the evolution of the part of the state which does not change is not specified at all, and therefore it is easier to combine such definitions. This advantage becomes visible in approaches where ASM rules are modeled as equation systems, for instance if ASMs are modeled with Algebraic Specifications (125; 136; 177; 178). The complete definition results in an exploding number of equations (125; 136) while the partial definition allows to solve this problem elegantly (178). Further the partial definition Definition 4 allows to *compose* the equations of the subrules, whereas the complete definition does not allow for such a composition.

The different forms of rules are given below. We use $eval_\lambda$ to denote the usual term evaluation in the state λ. In all definitions, $t_0, \ldots, t_n$ are terms over Σ.

Def. 5: Update denotations of ASM rules.

Basic Update

if $R = f(t_1, \ldots, t_n) := t_0$

then $\quad$ *Upd*$(R, \lambda) = (f, (eval_\lambda(t_1), \ldots, eval_\lambda(t_n)), eval_\lambda(t_0))$

Parallel Composition

if $R = R_1 \ldots R_m$

then $\quad$ *Upd*$(R, \lambda) = \bigcup_{i \in \{1,\ldots,m\}}$ *Upd*(R_i, λ)

Conditional Rules

if R = **if** t **then** R_{true} **else** R_{false} **endif**

then $\quad$ *Upd*$(R, \lambda) = \begin{cases} \textit{Upd}(R_{true}, \lambda) & if \quad eval_\lambda(t) = true \\ \textit{Upd}(R_{false}, \lambda) & otherwise \end{cases}$

Do-for-all

if R = **do forall** *x* **in** *U : Q(x)*

R'

enddo

then $\quad$ *Upd*$(R, \lambda) = \bigcup_{e \in U'}$ *Upd*$(R', \lambda \cup \{x \mapsto e\})$

where

- $U' = \{e | \ eval_\lambda(U(e) \wedge Q(e))\}$ *are U elements fulfilling Q.*
- $\lambda \cup \{x \mapsto e\}$ *is state* λ *with x interpreted as e.*

Choose

if R = **choose** *x* **in** *U : Q(x)*

R'

ifnone

R"

endchoose

then

$$\textit{Upd}(R, \lambda) = \begin{cases} \textit{Upd}(R', \lambda \cup \{x \mapsto \textit{ORACLE}\}) & \\ & if \quad \exists e : eval_\lambda(U(e) \wedge Q(e)) \\ \textit{Upd}(R'', \lambda) & otherwise \end{cases}$$

where ORACLE is a nondeterministically chosen element $e \in U_\lambda$ [4]*, fulfilling Q(e).*

Extend

if R = **extend** *U* **with** *x*

R'

endextend

then $\quad$ *Upd*$(R, \lambda) =$ *Upd*$(R', \lambda \cup \{x \mapsto e\}) \cup \{(U, (e), true)\}$,

where e does not belong to the domain or the co-domain of any of the functions in state λ*, i.e. is a new, unused element.*

[4] As mentioned, U_λ is the definition of U in state λ.

4.3 The XASM Specification Language

Due to the fact that the ASM approach defines a notion of *executing* specifications, it provides a perfect basis for a language, which can be used as a specification language as well as a high-level programming language. However, in order to upgrade to a realistic programming language, such a language must – besides other features – add a modularization concept to the core ASM constructs in order to provide the possibility to structure large-scale ASM-formalizations and to flexibly define reusable specification units. XASM realizes a component-based modularization concept based on a unification and generalization of ASM's rule and expression semantics. The unification of rules and expressions is done by considering each ASM construct, whether rule or expression, to have both an *update set denotation*, and to evaluate to a result, the so called *value denotation*.

In addition to the existing ASM constructs, we introduce a new feature, so called *external functions*[5]. External functions can be evaluated like normal functions, but as a result, both a value, and an update set are returned. For each external function, we need to specify its *update denotation* and its *value denotation*. Both denotations can be freely defined. The formal definition of external functions, their denotations, and the propagation of these denotations through the existing ASM term and rule constructors is given in Section 4.3.1.

While external functions make the calculation of rule sets, and thus the semantics of XASM rules *extensible*, we introduce a second new construct called *environment functions* in order to make XASM *open* to the outside computations. *Environment functions* are special dynamic functions whose initial definition is given as a parameter to an ASM. After an ASM terminates, the aggregated updates of the environment functions are returned as update denotation of the a complete ASM run. The formalization of ASM runs, environment functions, the update denotation of an ASM run in terms of state-delta, and the value denotation of an ASM run are given in Section 4.3.2.

For intuition, it is a good idea to think about environment functions as dynamic-functions passed to an ASM as reference parameters, and about external functions as locally declared procedures. Having both concepts we can plug the two mechanisms together by defining update and value denotation of an external function by means of an ASM run. Thus the evaluation of such an external function corresponds to running, or *calling* another ASM. The environment functions of the called ASM are given as functions of the calling ASM. The details how an external function can be realized as ASM are given in Section 4.3.3. The formalization is given by using the definition of update and value denotations of an ASM, as defined in Section 4.3.2 as the definition of the update and value denotations of the realized external function.

[5] In the context of ASMs the term ”external function” has been used in a different way. For the sake of simplicity we are using the term ”external function” only in connection with XASM, and not with ASMs and we are always referring to the XASM definition of ”external function”.

4.3.1 External Functions

In Section 4.2 the denotation of each ASM rule construct has been given as a set of updates. Denotation of terms has been formalized by means of the usual $eval_\lambda$ term evaluation. The denotation of each existing ASM construct is thus either a set of updates or an element, the result of its evaluation. The ASM constructs denoted by updates are the *rules*, and the ASM constructs denoted by values are the *terms*.

The idea of eXtensible ASMs (XASM) is to unify rules and terms, by considering each construct to have both an update and a value denotation. In pure ASMs rules would have the value denotation *undef* and expressions have the empty set as update denotation. In XASM *external functions* are introduced as a new construct having both denotations.

In order to avoid confusion with the standard $eval_\lambda$ function, we introduce a new function which gives the value-denotation.

Def. 6: **Value denotation.** *The value denotation of each rule or expression* R *in a state* λ *is defined to be an element of* $\mathfrak{U}$ *given by*

$$\text{Eval}(R, \lambda)$$

The external functions are declared using the keyword *external function*. Syntactically the external functions are used like normal functions. Function composition which involves external functions may thus result in updates, and we need therefore to redefine the update denotations of all rule constructions involving expressions, by refining Definition 5.

In order to simplify the presentation of semantics, we denote the external function symbols with underlined symbols, for instance $\underline{\text{f}}$. These symbols are grouped in the set Σ_{ext} of external symbols.

Def. 7: **Extended signature.** *The signature* Σ *is extended with the symbols* Σ_{ext} *of* external functions *to signature* Σ'*.*

$$\Sigma' = \Sigma \cup \Sigma_{ext}$$

Since the external functions are not part of an ASM's state, the definition of state λ is not affected, it is still a mapping from signature Σ of dynamic functions to the actual definitions of these functions. However, terms can be built over the extended signature Σ'.

Def. 8: **Denotations of external functions.** *For each external function* $\underline{f} \in \Sigma_{ext}$ *their updates and value denotations in state* λ *are given by*

$$\text{ExtUpd}(\underline{f}, (e_1, \ldots, e_n), \lambda)$$

and

$$\text{ExtEval}(\underline{f}, (e_1, \ldots, e_n), \lambda)$$

XASM features interfaces allowing to give these definitions in arbitrary external languages, which leads to a non-formal system, or in XASM itself, which leads to a formal system which is described in Section 4.3.3.

In the following we give the definition of *Upd* and *Eval* for function composition of dynamic functions $f \in \Sigma$, external functions $\underline{f} \in \Sigma_{ext}$, and all six rule constructors.

Def. 9: **Update and value denotations of XASM constructs.** *Assume in all following definitions*

- $t_0, \ldots, t_n$ *are terms over* Σ'*,*
- $e_0 = \mathit{Eval}(t_0, \lambda)$ *and* … *and* $e_n = \mathit{Eval}(t_n, \lambda)$ *are the elements these terms evaluate to,*
- $f \in \Sigma$ *is the symbol of a dynamic function, and*
- $\underline{f} \in \Sigma_{ext}$ *is the symbol of an external function.*

Function Composition

if $R = f(t_1, \ldots, t_n)$
then
$$\mathit{Upd}(R, \lambda) = \bigcup_{i \in \{1,\ldots,n\}} Upd(t_i, \lambda)$$
$$\mathit{Eval}(R, \lambda) = f_\lambda(e_1, \ldots, e_n)$$

External Function Composition

if $R = \underline{f}(t_1, \ldots, t_n)$
then
$$\mathit{Upd}(R, \lambda) = \mathit{ExtUpd}(\underline{f}, (e_1, \ldots, e_n), \lambda) \quad \cup \quad \bigcup_{i \in \{1,\ldots,n\}} Upd(t_i, \lambda)$$
$$\mathit{Eval}(R, \lambda) = \mathit{ExtEval}(\underline{f}, (e_1, \ldots, e_n), \lambda)$$

Basic Update

if $R = f(t_1, \ldots, t_n) := t_0$
then
$$\mathit{Upd}(R, \lambda) = \{(f, (e_1, \ldots, e_n), e_0)\} \quad \cup \quad \bigcup_{i \in \{1,\ldots,n\}} Upd(t_i, \lambda)$$
$$\mathit{Eval}(R, \lambda) = \mathit{undef}$$

Conditional Rules

if R = **if** t **then** R_{true} **else** R_{false} **endif**
then
$$\mathit{Upd}(R, \lambda) = \begin{cases} \mathit{Upd}(R_{true}, \lambda) \ \cup \ \mathit{Upd}(t, \lambda) & if \quad \mathit{Eval}(t, \lambda) = true \\ \mathit{Upd}(R_{false}, \lambda) \ \cup \ \mathit{Upd}(t, \lambda) & otherwise \end{cases}$$
$$\mathit{Eval}(R, \lambda) = \begin{cases} \mathit{Eval}(R_{true}, \lambda) & if \quad \mathit{Eval}(t, \lambda) = true \\ \mathit{Eval}(R_{false}, \lambda) & otherwise \end{cases}$$

Parallel Composition

if $R = R_1 \ldots R_m$

then

$\mathit{Upd}(R, \lambda) = \bigcup_{i \in \{1,\ldots,m\}} \mathit{Upd}(R_i, \lambda)$

$\mathit{Eval}(R, \lambda) = undef$

Do-for-all

```
if R = do forall x in U : Q(x)
           R'
         enddo
then
```

$$\begin{aligned}\mathit{Upd}(R, \lambda) = &\bigcup_{e1 \in U'} \mathit{Upd}(R', \lambda \cup \{x \mapsto e1\}) \\ &\cup \quad \bigcup_{e2 \in U_\lambda} Upd(Q(e2), \lambda)\end{aligned}$$

$\mathit{Eval}(R, \lambda) = undef$

where

- $U' = \{e|\ \mathit{Eval}(U(e) \wedge Q(e), \lambda)\}$ *are U elements fulfilling Q.*
- $\lambda \cup \{x \mapsto e\}$ *is state* λ *with x interpreted as e.*

Choose

```
if R = choose x in U : Q(x)
           R'
         ifnone
           R"
         endchoose
then
```

$$\mathit{Upd}(R, \lambda) = \begin{cases} \mathit{Upd}(R', \lambda \cup \{x \mapsto \mathit{ORACLE}\}) \\ \quad \cup\ \mathit{Upd}(Q(\mathit{ORACLE}), \lambda) \\ \qquad\qquad if \quad \exists e : \mathit{Eval}(U(e) \wedge Q(e), \lambda) \\ \mathit{Upd}(R'', \lambda) \\ \qquad\qquad otherwise \end{cases}$$

$$\mathit{Eval}(R, \lambda) = \begin{cases} \mathit{Eval}(R', \lambda \cup \{x \mapsto \mathit{ORACLE}\}) \\ \qquad\qquad if \quad \exists e : \mathit{Eval}(U(e) \wedge Q(e), \lambda) \\ \mathit{Eval}(R'', \lambda) \\ \qquad\qquad otherwise \end{cases}$$

where

ORACLE is a nondeterministically chosen element $e \in U$*, fulfilling Q(e) in* λ.

Extend

```
if R = extend U with x
          R'
```

```
        endextend
    then
      $\mathit{Upd}(R, \lambda) = \mathit{Upd}(R', \lambda \cup \{x \mapsto e\}) \cup \{(U, (e), true)\}$,
      $\mathit{Eval}(R, \lambda) = undef$
    where
      e does not belong to the domain or the co-domain of any of the
      functions in state $\lambda$.
```

4.3.2 Semantics of ASM run and Environment Functions

We have given the semantics of ASM rules and expressions in terms of defining the relation of one state to the next. In this section we formalize how the state of an ASM is initialized, by means of *parameters* and so called *environment functions*, and what is an ASM run. We give both value and update denotations of ASM runs.

We mentioned earlier that dynamic functions are initialized everywhere with undef, except for relations, which are initialized everywhere with false. Parameters and environment functions allow to initialize functions with different values. As example we take the following ASM.

ASM 5:

```
asm InitializationExample(p1, p2)

  updates function  f(_,_)
  accesses function g(_,_)
is
 function h(_,_)
R
endasm
```

The example shows two parameters, *p1* and *p2*, two environment functions, *f* and *g*, and one normal dynamic function *h*. If the ASM is started, or *called*, actual values for the parameters have to be given, as well as definitions for the environment functions. Parameters result in normal, 0-ary dynamic functions, which are initialized with the actual value. Environment functions are used to initialize functions of arity higher than zero. As we can see, there are two ways to declare environment functions, one for read-only access as ”accesses” and the other for read-write access as ”updates”. In addition to such declared functions there is the special 0-ary function *result* which is used to return values from an ASM run.

Intuitively environment functions correspond to reference parameters passed to an ASM call. The aggregated updates to these functions constitute the update denotation of an ASM run. In contrast parameters can be considered call-by-value arguments. Updates to such arguments are possible in Xasm, but they have only local effects.

The signature Σ of the state of an ASM consists thus of the normal dynamic functions, the 0-ary dynamic functions initialized by actual parameters, the environment functions, and the special function *result*.

Def. 10: Local and environment functions. *The signature Σ of dynamic functions is built by a set of locally defined functions Σ_{loc}, the set of parameter functions Σ_{par}, the set of environment functions Σ_{env} and the special function* result*. All of them must be pairwise disjoint.*

$$\begin{array}{rcl} \Sigma & = & \Sigma_{env} \cup \Sigma_{loc} \cup \Sigma_{par} \cup \{result\} \\ & \wedge & \\ \Sigma_{env} \cap \Sigma_{loc} \cap \Sigma_{par} \cap \{result\} & = & \{\} \end{array}$$

An ASM can now be called by providing it with actual parameters, and an initial state for the environment functions.

Def. 11: ASM call. *An ASM with rule R parameters $p_1, \ldots, p_n$ and environment functions Σ_{env} is called by the following triple*

$$(R, (a_1, ..., a_n), \lambda^e)$$

where $(a_1, \ldots, a_n)$ are actual values for the parameters of the ASM, and λ^e is a mapping from the function symbols of Σ_{env} to actual definitions for these functions.

Given an ASM call, we can define the initial state of the called ASM as follows.

Def. 12: Initial state. *Given an ASM call $(R, (a_1, ..., a_n), \lambda^e)$ with parameters $p_1, \ldots,$ p_n the initial state λ_0 of the called ASM is defined as follows.*

$$\lambda_0 = \lambda^e \cup \{p_1 \mapsto a_1, \ldots, p_n \mapsto a_n\}$$

Given the definition of the initial state and of the next state relation we can define the fixpoint semantics of an ASM run as follows.

Def. 13: Fixpoint semantics. *Given an ASM call $(R, (a_1, ..., a_n), \lambda^e)$, the definition of the initial state λ_0 of such a call, according to Definition 12, and the relation of state λ_i to λ_{i+1}, according to Definition 3, we define the fixpoint semantics Λ as a mapping from ASM calls to final states or $\perp$ if there is no fixpoint.*

$$\Lambda(R, (a_1, ..., a_n), \lambda^e) = \begin{cases} \lambda_i & if \quad \lambda_i = \lambda_{i+1}, not(\exists j : j < i : \lambda_j = \lambda_{j+1}) \\ \perp & if \quad not(\exists i :: \lambda_i = \lambda_{i+1}) \end{cases}$$

where $\perp$ denotes a non-terminating call.

Given the fixpoint semantics of an ASM call, we can define the update and value denotation of such a call. The value denotation is simply the value of function *result* in the final state of the call.

Def. 14: Value denotation of ASM call. *Given an ASM call* $(R, (a_1, ..., a_n), \lambda^e)$*, and the fixpoint semantics, according to Definition 13, the value denotation* CallEval *is the value of* result *in the final state of the call.*

$$\mathit{CallEval}(R, (a_1, ..., a_n), \lambda^e) = \mathit{result}_{\Lambda(R,(a_1,...,a_n),\lambda^e)}$$

The update denotation *CallUpd* of a call is given by the aggregated updates to environment functions. The aggregated updates are calculated by comparing the initial state and the terminal state of these functions. The comparison of states is done by *state subtraction*

Def. 15: State subtraction. *Given two states* λ_1 *and* λ_2 *over the same signature* Σ*, the formal definition of state subtraction is*

$$\begin{aligned} \lambda_1 - \lambda_2 \;\; = \;\; & \{(f, (e_1, \ldots, e_n), e_0) | \\ & \;\; f \in \Sigma \land e_0, \ldots, e_n \in \mathfrak{U} \\ & \;\; \land\; f_{\lambda_1}(e_1, \ldots, e_n) = e_0 \\ & \;\; \land\; f_{\lambda_2}(e_1, \ldots, e_n) \neq e_0 \\ & \} \end{aligned}$$

Using this definition, the update denotation of an ASM call is defined as follows.

Def. 16: Update denotation of ASM call. *Given an ASM call* $(R, (a_1, ..., a_n), \lambda^e)$*, the signature* Σ_{env} *of environment functions, the fixpoint semantics, according to Definition 13, and the definition of state subtraction according to Definition 15, the update denotation* CallUpd *is the environment part of the final state minus the initial state* λ^e *of the environment functions.*

$$\mathit{CallUpd}(R, (a_1, ..., a_n), \lambda^e) = \Lambda(R, (a_1, ..., a_n), \lambda^e)|_{\Sigma_{env}} - \lambda^e$$

4.3.3 Realizing External Functions with ASMs

After we specified both external functions, for which we need to give value and update denotations *ExtEval* and *ExtUpd*, and as well ASM calls, for which we defined value and update denotations *CallEval* and *CallUpd*, the next natural thing to do is to use the denotations of an ASM call as definitions of the denotations of an external function. With other words, we realize an external function with an ASM. The environment functions of the called ASM are naturally taken from the dynamic functions of the called ASM, and the resulting updates to these functions fit thus naturally in the update set of the calling ASM.

The definition of update set and value denotations of an external function realized by ASM can now be given by using *CallUpd* and *CallEval* as definitions of *ExtUpd* and *ExtEval*.

Def. 17: Denotations of ASM call. *Assume the external function* $\underline{f}$ *to be implemented by the following ASM:*

```
asm _f(p1, ..., pn)
 updates functions SIGMA_ENV
is
functions SIGMA_LOC
 R
endasm
```

where SIGMA_ENV *is the signature* Σ^c_{env} *of environment functions of the called ASM, and* SIGMA_LOC *is the signature* Σ^c_{loc} *of locally declared dynamic functions of the called ASM.*

Given a state λ *of the ASM calling* $\underline{f}$*, the denotations* ExtUpd *and* ExtEval *are defined as follows.*

$$\begin{array}{rcl} \mathit{ExtUpd}(\underline{f}, (e_1, \ldots, e_n), \lambda) & = & \mathit{CallUpd}(\underline{f}, (e_1, \ldots, e_n), \lambda|_{\Sigma_{env}}) \\ \mathit{ExtEval}(\underline{f}, (e_1, \ldots, e_n), \lambda) & = & \mathit{CallEval}(\underline{f}, (e_1, \ldots, e_n), \lambda|_{\Sigma_{env}}) \end{array}$$

Examples

Consider our previous example the ASM *AP*. An ASM *AQ*, can refer to *AP*, by declaring it as *external "ASM" function*, or short *external function*.

ASM 6:

```
asm AQ is
    function i <- 0
    external function AP
  if i < 10 then
     i := i+1
     AP
  endif
endasm
```

In *AQ* there is a local 0-ary function *i*, and the external function *AP*, which is realized as ASM. The if-clause in the rule of *AQ* guarantees that *AP* is called 10 times. Each time, *AP* is called, it runs until its termination, the final state of *AP* is interpreted as an update set, and the value of the function *return* in *AP* is used as return value. The update set generated by each run of *AP* is

$$\{(x_1, (), 1), (x_2, (), 1), (x_3, (), 1)\}$$

Since all of the updated functions are local to *AP*, the generated update set has no effects on the state of *AQ*. Further, in this simple case, the value of *return* is *undef*, since there is no update to *return* in *AP*. Thus the value denotation of calling *AP* is *undef*.

As second example consider two ASMs *A* and *B*. We abstract from concrete rules and consider *A* to execute the parallel composition of a rule *Ra* and a call to *B*, while *B* is considered to execute a rule *Rb*. *A* has locally defined functions $a_1, \ldots, a_n$ and *B* has locally defined functions $b_1, \ldots, b_m$.

ASM 7:

```
asm A is
    functions a1, ...., an
    external function B
  Ra
  B
endasm
```

ASM 8:

```
asm B
    updates functions a1, ..., an
is
    functions b1, ..., bm
  Rb
endasm
```

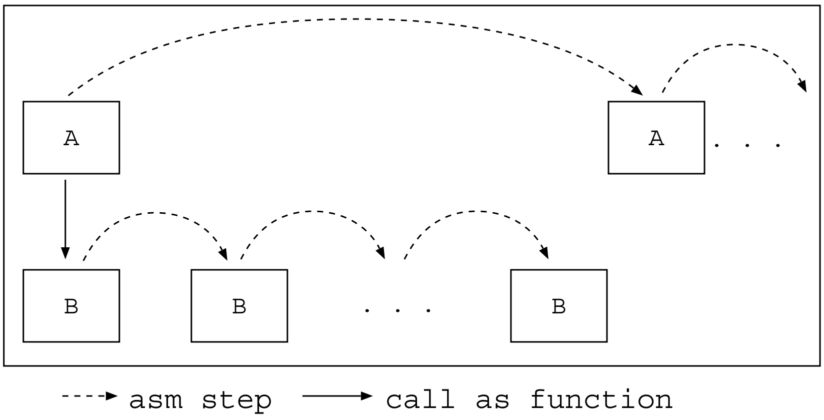


Fig. 38: ASM A calls ASM B

The interface of *B* determines that ASM calling *B* must provide dynamic functions $a1, \ldots, a_n$ which are allowed to be updated by *B*.

The situation of *A* calling *B* is visualized in Fig. 38. In each step of *A*, the rule *Ra* as well as ASM *B* are executed. If *B* is called, the current state of *A*'s functions is passed to *B* as the initial state of the environment functions. From this state, *B* runs until its termination, updating the state of its local functions as well as the state of the environment functions. After termination, the state of the local functions of *B*, is discarded, and the state of the environment functions is compared with their initial state, passed by the environment. The changes with respect to the initial state are returned as the update-denotation of the *B*-call.

The updated-denotation of the *B*-call is combined with the update-denotation of the *Ra*-rule, and applied to the current state of *A*. Only now *A*'s locally defined functions are really updated. The internal steps of *B* are not visible to *A*. From *A*'s perspective, calling *B* is considered an atomic action. The XASM call provides thus an abstraction from sequentiality.

Returning values We have mentioned several times the special role of the function *result*, but we have not shown its use and examples. Based on the above definitions, *result* must be declared as local function and updated like any other function. The termination of an ASM does not a priori depend on the state of *result*. A typical "factorial"-program would look as follows.

ASM 9:

```
asm factorial(n) is
    function result
  if result != undef then
    if n = 0 then
      result := 0
    else
      result := n * factorial(n-1)
    endif
  endif
endasm
```

For convenience a shorthand notation allows the user to skip the explicit declaration of the variable ”result”, as well as the outer ”if result != undef”-clause, and it introduces the more intuitive syntax ”return x” instead of ”result := x”. Applied to the previous example, the shorthand notation results into the following formulation.

ASM 10:

```
asm factorial(n) is
  if n = 0 then
    return 0
  else
    return n * factorial(n-1)
  endif
endasm
```

As a last example of this section, we would like to show a formulation of “factorial” which avoids call-recursion.

ASM 11:

```
asm factorial(n) is
    function n0 <- n, r <- 1
  if n0 > 0 then
    r := n0 * r
    n0  := n0 - 1
  else
    return r
  endif
endasm
```

Every tail-recursive algorithm can be reformulated in this iterative style. We will use this stile throughout the thesis, since it shows clearer how ASMs work. In the following variant of factorial we use the fact, that the parameters of an ASM can be used as normal 0-ary dynamic functions.

ASM 12:

```
asm factorial(n) is
    function r <- 1
  if n > 0 then
    r := n * r
    n  := n - 1
  else
    return r
  endif
endasm
```

4.4 Constructors, Pattern Matching, and Derived Functions

Most theoretical case studies using ASMs start with a mathematical model of some static system, formalized as a fixed set of statically defined functions and elements, and add a number of dynamic functions on top of this algebra. With the up to now discussed features, the static models must be either provided by an external implementation, or simulated with dynamic functions as well.

4.4.1 Constructors

While experimenting with early versions of XASM, we identified one mathematical concept which is on one hand often used, and on the other hand very awkwardly simulated with dynamic functions. The identified concept is *free-generated-terms*. Unlike terms over dynamic functions, evaluating initially all to the same element *undef*, free-generated-terms, or *constructors* are expected to map to the same element, if and only if all their arguments are equal. This concept corresponds to free-data-types in functional programming languages like Standard ML (155; 40) or term algebras in algebraic specifications (65). XASM features an untyped variant of classical constructor terms, as well as pattern matching and derived functions. These three features form a pure functional subset of XASM. In Section 4.4 we give the details of these features.

In functional languages, typically each element of a constructor is typed with some free-data-type. In contrast, the XASM constructors take arbitrary arguments, even dynamically allocated elements, and construct a unique element from each unique sequence of arguments.

The definition of the two constructors

```
constructors zero, successor(_)
```

is thus not only creating the elements $\{$*zero, successor(zero), successor(successor(zero),* $\ldots\}$, but as well unexpected elements like *successor(true)* or *successor(*e_0*)*, where e_0 is an element created by an *extend*-rule; since such dynamically elements elements do not correspond to any symbol for built-in constants, XASM allows the user to define constructor-terms having no syntactical representation.

4.4.2 Pattern Matching

In combination with constructors, it is very useful to have *pattern matching* and *derived functions*. As an example for pattern matching, consider an abstract-data-type stack, being specified by the following equations.

$$\begin{aligned} pop(push(s,v)) &= s \\ top(push(s,v)) &= v \\ top(empty) &= undef \\ pop(empty) &= empty \end{aligned}$$

Two constructors *empty* and *push(_, _)* are used to build stacks in the usual way. *top(_)* and *pop(_)* are declared as external functions and realized as ASMs. Within these ASMs, pattern matching is used.

ASM 13:

```
constructors empty, push(_,_)
external functions top(_), pop(_)

asm top(s)
  accesses function push(_,_)
is
  if s =~ push(&, &v) then
    return &v
  else
    return undef
  endif
endasm

asm pop
  accesses functions empty, push(_,_)
is
  if s =~ push(&s, &) then
    return &s
  else
    return empty
  endif
endasm
```

We see the pattern matching symbol “=~” and the pattern variables, which all start with the symbol &. The plain symbol & is a placeholder for pattern variables, whose value is not used anymore. The matching-expression is given as condition of an if-then-else rule. If a match happens, the pattern-variables can be used, otherwise they cannot. Thus pattern-variables can only be used in the then-clause of an if-then-else rule.

4.4.3 Derived Functions

A third construct which is useful in combination with constructors and pattern-matching is the *derived function*. The value of derived function is defined by an expression. The derived function

```
derived function f(p1, ..., pn) == t
```

where *t* is a term build over Σ and the parameters $p_1, \ldots, p_n$, is semantically equivalent to an external function defined as follows.

```
external function f(p1, ..., pn)

asm  f(p1, ..., pn)
  accesses ...
is
  return t
endasm
```

Tab. 3: Properties of XASM function types

Function Types	updatable?	initial value	generate updates?
dynamic function	yes	undef	no
constructor	no	free-generated	no
derived function	no	calculated	yes
external function	yes	calculated	yes
asm	yes	calculated	yes

Using derived functions, the above example ASM 13 can be reformulated as follows:

ASM 14:
```
constructors empty, push(_,_)
derived function top(s) ==
  (if s =~ push(&, &v) then &v else undef)
derived function pop(s) ==
  (if s =~ push(&s, &) then &s else empty)
```

4.4.4 Relation of Function Kinds

Using only constructors, pattern-matching, and derived functions, XASM can be used as a *pure functional language*. An arbitrary part of an XASM specification can thus be written in the functional paradigm.

However, if derived functions are defined over dynamic functions, their value depends on the state, and if derived functions are used in combination with extension functions, they may even produce updates. Table 3 lists the different types of functions in XASM, as well as the information

- whether they can be updated,
- what is their initial value, and
- whether they generate new updates if they are evaluated[6].

We marked both external functions, as well as locally defined ASMs as updatable. This feature is useful to refine models, by replacing dynamic functions with external functions, for instance data-bases. The XASM implementation is organized such, that first all read accesses to external functions are done, and then all updates.

4.4.5 Formal Semantics of Constructors

The concept of terms built up by constructors can be mapped to the ASM approach as follows: each of the function names may be marked as *constructive*, expressing that constructor functions are one-to-one and total.

[6] New updates are those resulting from the function evaluation itself, and not from the evaluation of the functions argument.

Let $\Sigma_c \subseteq \Sigma$ be the set of all constructive function symbols. If $f \in \Sigma_c$, be of arity *n*, $g \in \Sigma_c$, be of arity *m*, and $t_1, \dots t_n, s_1, \dots, s_m$ be terms over Σ', then the following condition hold for all states λ_i of the ASM:

(i)

$$f(t_1, \dots, t_n) = g(s_1, \dots, s_m)$$

iff

$$(f = g) \text{ and } (n = m) \text{ and } (eval_{\lambda_i}(t_k) = eval_{\lambda_i}(s_k))$$

for all $1 \leq k \leq n$ where $eval_{\lambda_i}(t)$ stands for the evaluation of the term *t* in state λ_i of the ASM. Informally speaking it means that each constructive function is total with respect to $\mathfrak{U}$ and injective.

(ii) For all $j > i, e_1 \in \mathfrak{U}_{\lambda_i}, \dots, e_n \in \mathfrak{U}_{\lambda_i}$

$$f_{\lambda_i}(e_1, \dots, e_n) = f_{\lambda_j}(e_1, \dots, e_n)$$

This means that constructive functions do not change their values with time, but whenever a new element is created, the domain of all constructive functions is automatically extended to the new element; from that moment on, all elements constructed from the newly defined element do not change in time either.

If $f \in \Sigma_c$, then *f* is called a *constructor*, and terms $f(t_1, \dots, t_n)$ built only over Σ_c are called *constructor terms*. In the following, we use the constructor term *t* as a synonym for its unique value $eval_\lambda(t)$.

4.5 EBNF and Constructor Mappings

XASM features specialized programming constructs to define EBNF grammars, to parse strings according to these grammars, and build during the parsing a constructor term representing the AST. In this section we introduce these programming language related features which have been integrated into the XASM language as a means to support the implementation of various meta-programming algorithms, such as the later presented self-interpreter (Section 5.4), type-checkers, attribute grammar engine (Section 7), partial-evaluators (Section 5.5), as well as the specification and implementation of Montages in Section 8.

The existing XASM implementation features a relatively direct integration with the Lex/Yacc tool-set, supporting only BNF rules, instead of EBNF, and forcing the user to program the construction of constructor terms or other structures during the parsing. We introduce here a refined version where full EBNF rules can be specified, and where the construction of the terms representing the AST is done with a declarative mapping from EBNF productions into constructor terms. The purpose of our refined definitions is to allow for a complete specification of the parsing and AST construction process of Montages, without having to code the detailed construction, and especially without having to simulate EBNF with BNF rules. We abstract here from the problems of integrating our refined features with a specific parser generator.

4.5.1 Basic EBNF productions

As mentioned in Section 3, the *EBNF production rules* are used for the context-free syntax of the specified language *L*, and allow the generation of a parser for programs of *L*. Given an *L* program, a parser reconstructs the (recursive) applications of the EBNF productions such that the generated string corresponds to the program.

The result of parsing is a syntax-tree, being formalized in our framework as a constructor-term built up during the parsing. The mapping from programs into constructor terms can be given by denoting for each EBNF production a constructor, and defining how the constructor-representations of the parsed symbols on the right hand side are embedded into the constructor term. Basic EBNF productions and the difference between *characteristic* and *synonym productions* have been given in Section 3.2.1.

Characteristic productions

References to the right-hand symbols in *characteristic productions* are done via their names, possibly marked by their number of occurrence. Assume $a(\_, \_, \_, \_)$ is a 4-ary XASM constructor. A characteristic production

```
A ::= B  C  D D
```

extended with mapping

```
=> a(B, C, D.1, D.2)
```

returns a constructor-term $a(arg_1, arg_2, arg_3, arg_4)$, whose arguments arg_i are the constructor-terms returned by the parsed right-hand sides symbols.

Micro syntax
In the case of variable terminals, the term *Name* returns the micro-syntax. For brevity we are not giving here the details how to define variable terminals, but of course we use the standard technique of *regular expressions*. For instance, the definition of a typical *Ident* symbol returning its micro-syntax could be given as follows.

```
Ident = [A-Za-z][A-Za-z0-9]* => Name
```

In all other cases, *Name* returns a string representation of the left-hand-side symbol. For instance, the following mapping of the above characteristic rule

```
A ::= B  C  D D
=> characteristic(Name, B, C, D.1, D.2)
```

results in a constructor term

$$characteristic(''A'', arg_1, arg_2, arg_3, arg_4)$$

where again arguments arg_i are the constructor-terms returned by the parsed right-hand sides symbols.
Synonym productions
For *synonym productions*, the chosen right-hand side is accessible as term *rhs*. A synonym production

```
E = F | G | H => e(rhs)
```

returns the term

$$e(x)$$

where *x* is the chosen right-hand side. As an alternative one can return only the right-hand side, e.g. the production

```
E = F | G | H => rhs
```

returns directly the chosen right-hand side. Returning a constructor from a synonym rule allows to keep information which synonym rules have been triggered, while returning directly *rhs* allows to compactify the resulting terms.

A third alternative is to map the results of the synonym-production into a special constructor *synonym* and to use the *Name* term to store which synonym rule has been used. The production

```
E = F | G | H => synonym(Name, rhs)
```

returns a constructor term

$$synonym(''E'', arg_0)$$

where arg_0 is the constructor term returned from parsing one of the right-hand side symbols.

Example

As an example we extend the syntax rules of language $\mathcal{S}$ (Gram. 1 in Section 3.2.2) with a mapping from parsed programs into constructor terms.

Later in Section 4.5.3 the same grammar is used with an alternative mapping, using the above solutions with the ”characteristic” and ”synonym” constructors. The interested reader is invited to consult these examples already now.

Gram. 3:

```
Expr          =     Sum | Factor
                => expr(rhs)
Sum           ::=   Factor "+" Expr
                => sum(Factor, Expr)
Factor        =     Variable | Constant
                => factor(rhs)
Variable      ::=   Ident
                => variable(Ident)
Constant      ::=   Digits
                => constant(Digits)
Ident         =     [A-Za-z][A-Za-z0-9]*
                => ident(Name)
Digits        =     [0-9]+
                => digits(Name.strToInt)
```

If a ”Sum” is parsed, the constructor *sum(_,_)* is returned, having as first argument the constructor returned for the parsed ”Factor”, and as second argument the constructor returned for the parsed ”Expr”. If one of the synonyms is parsed, the chosen right-hand side is returned as unique argument of the constructor corresponding to the synonym. For instance an instance e_0 of symbol *Expr* is returned as term *expr(*e_0*)*. If a ”Variable” is parsed, constructor *variable(_)* with the representation of the Ident as argument is returned, and finally, if a ”Constant” is parsed, the constructor *constant(_)* with the representation of the Digits is returned. Finally, Ident and Digits return constructors with their micro-syntax as arguments.

Considering again the example program ”2 + x + 1” of Section 3.2.2, the textual version of the constructor term resulting from applying the above mapping is given as follows.

Term 6:

```
expr(sum(factor(constant(digits(2))),
         expr(sum(factor(variable(ident("x"))),
                  expr(factor(constant(digits(1))))
                 )
             )
         ))
```

A visualization of this term can be seen on the left-hand side of Figure 40.

4.5.2 Repetitions and Options in EBNF

On the right-hand side of characteristic productions, not only non-terminal symbols, but repetitions and options are allowed. Repetitions and options are treated similar to the way how they are treated in Montages, as described in Section 3.4. Symbols within curly repetition brackets return a list of representations of the corresponding symbol. The EBNF list

```
{ A B }
```

parses sequences of AB, but returns as A a sequence of A, and as B a sequence of B. For instance a production

```
L ::= { A B } => l(A, B)
```

parsing ”$A_1B_1A_2B_2A_3B_3$” results in constructor term

```
l([A1, A2, A3], [B1, B2, B3])
```

Further, a single symbol, followed or preceded by a list containing the same symbol and possibly some terminals is collected in one list. The EBNF clause

```
A {";" A}                {A ";"} A
```

are both parsing sequences like A;A or A;A;A, and return as A one list of A instances.

Symbols within square option brackets return an empty list, if the optional symbol is not present and the representation of the symbol otherwise. This is especially practical in combination with the above feature, since an EBNF clause

```
["(" A {";" A} ")"]
```

is returning as A an empty list, if nothing is present (as defined by the rule for square brackets), a single A, if one A is present, and a list of A's, if two or more A's are present (as defined by the rule for curly brackets.)

4.5.3 Canonical Representation of Arbitrary Programs

In addition to the possibility of defining custom mappings, we define a default, *canonical mapping* into a generic term representation using the above mentioned constructors *characteristic* and *synonym*. This canonical mapping is later used as starting point to construct ASTs like those introduced in Section 3.2.

- Given a *characteristic EBNF rule*

  ```
  A  ::= B C D D
  ```

  the generic mapping is

  ```
  => characteristic(Name, [B,
                           C,
                           D.1,
                           D.2])
  ```

- Given a *synonym rule*

  ```
  E = F | G | H
  ```

the generic mapping is

```
=> synonym(Name, rhs)
```

where rhs is an operator allowing to access what comes back on the right-hand side.

- Given a *terminal* ”x”, the generic mapping is omitting the terminal.
- Given a *right-hand side symbol within a list*, the mapping is that symbol. For instance the Rules

```
K ::=   { L }
K ::=   L  {"," L}
K ::=   ["(" L {';" L} ")"]
```

all result in the mapping

```
=> characteristic(Name, L)
```

- Correspondingly, if a *symbol is in option brackets*, the mapping is the symbol.

Following this rules it is possible to write a generator, taking as input a term representation of EBNF rules, and outputting a term representation of the same EBNF decorated with constructor mappings according to the above description of a canonical mapping. This generator is called *GenerateEBNFmapping(_)*. For the sake of brevity, we are not giving the full definition of this generator.
Example
Given again the grammar $\mathcal{S}$ (Grammar 1 in Section 3.2.2) the result of applying *GenerateEBNFmapping(_)* is the following grammar.

Gram. 4:	*Expr*	=	*Sum* | *Factor*
		=> synonym(Name, rhs)	
	Sum	::=	*Factor* “+” *Expr*
		=> characteristic(Name, [Factor, Expr])	
	Factor	=	*Variable* | *Constant*
		=> synonym(Name, rhs)	
	Variable	::=	*Ident*
		=> characteristic(Name,[Ident])	
	Constant	::=	*Digits*
		=> characteristic(Name, [Digits])	
	Ident	=	*[A-Za-z][A-Za-z0-9]**
		=> terminal(”Ident”, Name)	
	Digits	=	*[0-9]+*
		=> terminal(”Digits”, Name.strToInt)	

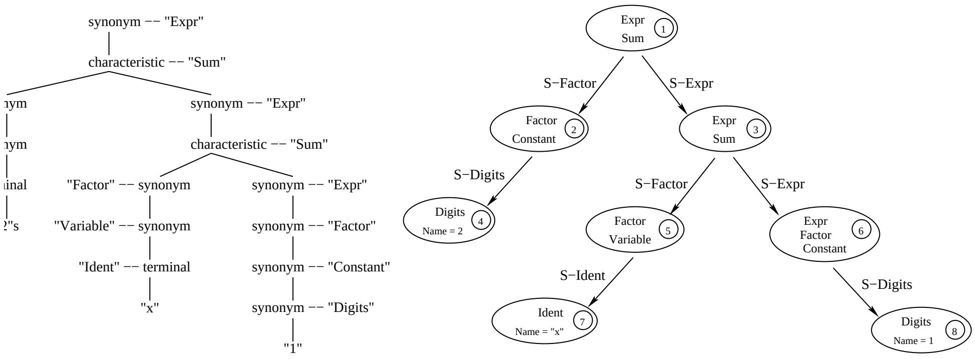


Fig. 39: The canonic constructor term and the abstract syntax tree for `2 + x + 1`

As we can see, in contrast to the customized mapping of Grammar3 in Section 4.5, the canonical mapping uses only the generic constructors synonym and characteristic.

Considering once again the example program ”2 + x + 1” of Section 3.2.2, the textual version of the resulting constructor term is given as follows.

Term 7:
```
synonym("Expr",
 characteristic("Sum",
    [synonym("Factor",
      characteristic("Constant",[terminal("Digits",2)])),
     synonym("Expr",
      characteristic("Sum",
          [synonym("Factor",
            characteristic("Variable",  [terminal("Ident","x")])),
           synonym("Expr",
            synonym("Factor",
             characteristic("Constant", [terminal("Digits", 1)])))
          ]))
    ]))
```

Compared to the customized version Term 6 the above term is longer, but it is easier to process this kind of generic terms, using only a fixed set of constructors, in a generic way. In Figure 39 we show on the left-hand side a visualization of the constructor term, resulting from the application of the new, canonic mapping. The mentioned customized mapping (Grammar3 in Section 4.5) is visualized in Figure 40. The right-hand side of both figures show the parse tree which needs to be created for the Montages models. The advantage of the canonical mapping is, that a generic XASM formalization of the parse tree creation can be given more easily.

4.6 Related Work and Results

ASMs are a combination of parallel execution, treatment of data-structures as variable functions, and implicit looping. Parallel execution is well know from Hardware description languages like VHDL (144). The treatment of data structures as variable functions is known from early work on axiomatic program verification (95; 59) and has been stated explicitly in (190). While existing work aimed at modeling concrete memory- or data-structures in hardware or software, Gurevich's ASM are defined as dynamic versions of Tarsky structures (207). Based on the fact, that Tarsky structures are the most common tool of mathematicians to describe static systems, they are logical candidate to represent in a most general way a single state of a dynamic computation. Another field using structures to describe static systems are algebraic specifications (72). As well in that field it has been observed that the absence of state makes many interesting applications infeasible. This lead to work proposing extension of algebraic specifications with state (52; 17; 181). Unlike these approaches, ASMs allow to define evolution of the state in the most direct form: by explicit enumeration of the pointwise difference from one state to the next. All other approaches try to reduce the allowable state-updates to a minimum, in order to guarantee the preservation of certain properties from one state to the next. In contrast to this, ASMs allow to make arbitrary many changes from one state to the next.

Still Gurevich's initial program for ASMs is pure mathematical: a mathematically defined dynamic system, which would allow to model arbitrary algorithms. His thesis (79) is that unlike Turing machines (211) his machines would allow to model algorithm without encoding data-structures and splitting execution steps. He observed that every conceivable data-structure can be modeled as a Tarsky structure, and every possible state change of the algorithm can be modeled by a set of explicit, pointwise changes to the structure. A proof of the thesis for sequential algorithms is given (83; 84).

This pure mathematical program, has been implicitly transformed in a computer science project, by defining a concrete *rule-language* for constructing the update sets. While in earlier publications (79; 80) Gurevich is investigating the concept of dynamically changeable Tarsky structures, later he defines a set of fixed, minimal languages for defining rules (81). ASMs are then defined to correspond to this rule-programming-language, and under this interpretation the thesis has subsequently provoked a lot of polarization among computer scientists. The lack of modularization and reuse feature in the proposed languages is for computer scientists not compatible with the claim, that arbitrary algorithms can be modeled on their natural abstraction level. While the initial mathematical meaning of this sentence makes a lot of sense, it contradicts computer scientist's experience, if “algorithm” is interpreted as software or hardware, and “modeled” is interpreted as “prototyped” or even “implemented” in a feasible and maintainable way.

However, the debate on ASMs in computer science has led to an impressive collection of case studies, each of them using ASMs to model a system which is considered to be complex. Examples are referenced in the annotated bibliography (29). While most models try to restrict the used rule-languages to the predefined ones, in many cases additional machinery has been used in order to manage the complexity. Such machinery reuses typically common concepts from programming languages.

The functional programming paradigm has been considered as the best candidate for extending the minimal rule-languages. The reason is, that many theoretical ASM case studies use a considerable amount of higher mathematics to describe the static part of algorithms. Functional programming is ideal to model higher mathematics and it uses modularization concepts based on mathematical concepts. This approach has led to a number of ASM implementations based on functional languages (220; 54). Odersky (168) proposes the opposite way, e.g. to use variable functions as an additional construct in functional programming languages. In both cases a functional type system is proposed. The introduction of such a type system is helpful for cases where the described algorithms fits well into the type system. On the other hand, Gurevich's original untyped definition of ASMs still provides the highest level of flexibility. We do not know of an ASM implementation based on functional languages which provides an implementation of the original, untyped definition of ASMs.

Today's software systems reached a level of complexity leading to use of multiple paradigms (48). Our experience shows that untyped ASMs are useful to use different paradigms in parallel. The idea behind XASM is to start with Gurevich's untyped definition of ASMs (80) and to make it extensible. The exact mechanisms have been discussed before. Unlike other extensions of ASMs, the XASM approach does not alter the semantics idea of Tarsky structures and update sets. The only difference of XASM to Gurevich's ASMs is, that we allow extensible rule languages. Since the means for extension are again ASMs, the XASM call can be seen as well as a way to structure ASMs.

An algebraic view of a similar structuring concept has been given by May in (150). The XASM call is a special case of notions defined in (150). While May applies the state of the art in algebraic specification technologies to ASMs, the idea of XASM is to generalize the original idea of Gurevich, resulting in a more practical specification and implementation tool. Unlike many other proposals for extending ASMs, the XASM approach tries to follow Gurevich's style to introduce as few concepts as possible. In fact, the XASM call, which is a simple generalization of Gurevich's denotational semantics of ASMs (82), is the only new concept and can be used to define all other extensions.

Another field using structures to describe static systems are *algebraic specifications* (72). As well in that field it has been observed that the absence of state makes many interesting applications infeasible. This lead to work proposing extension of algebraic specifications with state (52; 17; 181). Unlike these approaches, ASMs allow to define evolution of the state in the most direct form: by explicit enumeration of the pointwise difference from one state to the next. All other approaches try to reduce the allowable state-updates to a minimum, in order to guarantee the preservation of certain properties from one state to the next. In contrast to this, ASMs allow to make arbitrary many changes from one state to the next. Some newer work on modeling transition systems with algebraic specifications (125; 136; 177; 178) led to the *Especs* formalism which allows to map full ASMs into their framework, combining their power with the structuring and refinement techniques of algebraic specifications.

Based on our experience we would like to challenge the ASM thesis as follows. Agreeing on the choice of Tarsky structures and update set for modeling algorithms, we claim that the current choice of ASM constructs is not able to fulfill the ASM thesis. There are two problems with the current rule-language.

- Although theoretically every update set can be denoted by an appropriate ASM rule, the abstraction level how the update set is calculated is fixed.
- Although theoretically an arbitrary signature can be chosen, the abstraction level for defining this signature is fixed.

We propose to remedy these problems by extending ASM such, that both the update sets, and the definitions of signatures can be calculated by means of another ASM. The XASM call is a way to calculate updates sets with other ASMs, and Mapping Automata (101) (Appendix B) or parameterized XASM (Chapter 5) are proposals how to use ASMs to calculate the signature. It would go beyond the scope of this thesis to discuss whether this is a real challenge of the ASM thesis or whether it is only an indication that the choice of a fixed rule language should be reconsidered.

The XASM language is fully implemented and available as Open Source (8). The system is used as the basis for the Montages/Gem-Mex, where generated XASM code is translated into an interpreter for the language specified using Montages. Other case studies are an application to microprocessor simulation (208) and the application of XASM as gluing code in legacy systems (13).

Additional theoretical applications outside the ASM area are possible, since ASMs can be considered as an instance of so called *transitions system models* (194), which form as well the basis for other popular formalisms such as UNITY (41), TLA (140), SPL (149) and the SAL intermediate language (194). Using Montages, both syntax and semantics of new or alternative XASM constructs can be developed in the integrated development environment Gem-Mex. Such an extensible system architecture allows to tailor XASM as a tool for one of the above mentioned formalisms based on transition systems.

5

Parameterized XASM

The purpose of this chapter is to extend XASM with features for *parameterization* of their signature. Parameterization allows us to ”program” the signature of an algorithm. This possibility is especial useful if abstract algorithms are defined, which are intended to operate on concrete data-structures. As an example imagine an XASM-algorithm *INTERP* which interpretes textual representation of XASM-rules in such a way, that the interpretation of a rule has exactly the same effects as the direct execution of it. The algorithm *INTERP* needs thus to access and update functions which are given by the signature of the interpreted XASM-rule. This is only possible if we can parameterize the signature of *INTERP* with the signature of the interpreted rule. Another example is partial evaluation of interpreters, where it is often desirable that the resulting specialized program has a signature similar to the signature of the interpreted program. Otherwise the author of the program cannot validate the specialized code with respect to her/his original formulation. In our context, we aim at using parameterization to give an XASM semantics of Montages which can be specialized to a simple XASM for each program in the described language.

In Section 5.1 we motivate parameterized XASM (PXasm), by showing that they are needed for a generic algorithm constructing the abstract syntax trees (ASTs) used in Montages. The new programming-features of PXasm are introduced in Section 5.2. The design principle of these new features is that if an ASM B is called by an ASM A, the information dynamically calculated by A before the call can be used to defined the signature of B. From B's point of view, the signature is still static, but it is instantiated differently at each time B is called. Therefore our design of parameterized XASM can be seen as another conservative extension to standard ASMs. In the run of a parameterized ASM, the state is still a Tarski structure, and the transition rule can be easily specialized to a traditional ASM rule.

In order to avoid confusion we use the term ASM to refer to an abstract machine given by the XASM construct `asm ... is ... endasm`, and we say *traditional ASM* if we mean Abstract State Machines as defined by Gurevich (82). Parameterized XASM are referred to as PXasm.

The construction of ASTs for Montages, which serves as a motivating example for PXasm, is formalized with the new features in Section 5.3. Another example for the use of the new features is the definition of an XASM self-interpreter, which executes rules and evaluates terms (Section 5.4). In Section 6.1 of the next Chapter this self-interpreter will be applied to give a tree finite state machine (TFSM) interpreter, which later serves as core of our Montages semantics.

Finally in Section 5.5 we come to *partial evaluation*, the main application of PXasm. We define a partial evaluator for the PXasm formalism written in PXasm. In the next Chapter we will show in detail how the previously given TFSM interpreter can be specialized in compiled code by assuming that a given TFSM is static. This process of specializing the TFSM interpreter corresponds directly to the process of specializing the Montages meta-interpreter into compiled code. Since the details of the full process are not given, this section serves as a more detailed description of the Montages system architecture described in Figure 37.

Throughout the chapter we define and explain in detail a number of longer and more complex XASM programs for constructing canonic trees (ASM 18), finding enclosing instances of tree nodes (ASM 20), doing self-interpretation of XASM rules (ASM 25). We include the full definitions because they are integral parts of the formal Montages semantics given in Chapter 8.

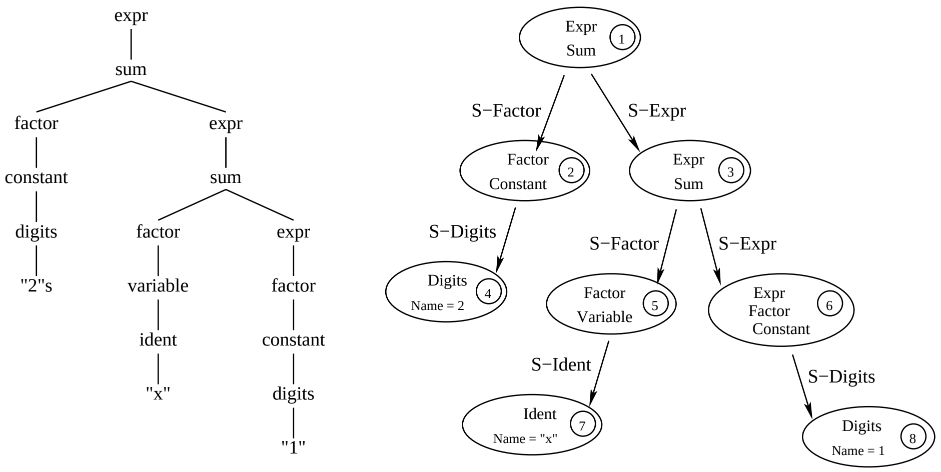


Fig. 40: The constructor term and the abstract syntax tree for `2 + x + 1`

5.1 Motivation

In Section 3 we have given an example and an informal model of a language specification in the Montages style. Since the signature of rules and actions of such a model depends on the specific EBNF of the described language, it is not possible to give a standard XASM modeling Montages of different languages with one fixed signature. Since defining a different XASM for each language described is too much overhead, we need additional features which allow to parameterize the signature of an XASM model.

As an example consider the XASM model for the ASTs of the presented example language $\mathcal{S}$ in Section 3. The model features special universes for each symbol in the EBNF of $\mathcal{S}$ and selector functions with names derived from the symbols in the EBNF. The rule *Sum ::= Factor “+” Expr*, for instance, introduces universes *Sum*, *Factor*, and *Expr*, as well as unary selector functions *S-Factor* and *S-Expr*. Formal semantics of the parse-tree construction can now be given based on the representation of programs as constructor terms. A possible mapping of $\mathcal{S}$ to constructors has been defined in Section 4.5.1. In Figure 40 we show on the left-hand side a visualization of the constructor term Term 6, resulting from the application of the mapping, and on the right-hand side the corresponding parse tree as shown already in Figure 19 of Section 3.2.2. The ASM *ConstructTree* which will be given below implements the construction of parse trees from constructor terms. While it is easy to write such an ASM for each possible EBNF, we cannot easily give a conventional ASM taking a constructor-term generated for an arbitrary EBNF, and constructing a corresponding AST along the guidelines of Section 3.2.2. Even if the mapping into constructor terms is the same for each EBNF productions, for instance using the canonical mapping as described in Section 4.5.3, we still would have to solve the problem how to parameterize the signature of universes and selector functions with the symbols existing in a specific EBNF grammar.

ASM 15:

```
asm ConstructTree(t)
  accesses constructors sum(_,_), expr(_), factor(_),
                        variable(_), constant(_)
  updates universes Expr, Sum, Factor, Constant, Variable
  updates functions S-Factor(_), S-Expr(_),
                    S-Digits(_), S-Ident(_)
is

  if     t =~ sum(&l, &r) then
    extend Sum with n
      n.S-Factor := ConstructTree(&l)
      n.S-Expr   := ConstructTree(&r)
      return n
    endextend
  elseif t =~ expr(&a) then
    let n = ConstructTree(&a) in
      Expr(n) := true
      return n
    endlet
  elseif t =~ factor(&a) then
    let n = ConstructTree(&a) in
      Factor(n) := true
      return n
    endlet
  elseif t =~ variable(&a)
     extend Variable with n
      n.S-Ident := ConstructTree(&a)
      return n
    endextend
  elseif t =~ constant(&a)
     extend Constant with n
      n.S-Digits := ConstructTree(&a)
      return n
    endextend
  elseif t =~ ident(&a)
     extend Ident with n
      n.Name := &a
      return n
    endextend
  elseif t =~ digits(&a)
     extend Digits with n
      n.Name := &a
      return n
    endextend
  endif
endasm
```

5.2 The $, Apply, and Update Features

For situations where the needed signature is not known in advance, we allow to declare and use functions by referencing them with a string-value, using the $, *Apply*, and *Update* features of PXasm[1]. The design principle of the new features is that if an ASM *B* is called by an ASM *A*, the information dynamically calculated by *A* can be used to define the signature of *B*. From *B*'s point of view, the signature is still static, but it is instantiated differently at each time of *B*'s call.

Because of the design principle, the string references to functions are resolved at different times for the declaration part and the rule part of an ASM. The occurrences in the declaration part are resolved at the time when the ASM is called, and the occurrences in the rule are resolved at execution time. The rules have dynamic signature, depending on the evaluation of the terms referring to functions. Nevertheless, the signature of such an ASM is not dynamic, but determined at call time. In the rule evaluation they are checked each time to be consistent with the signature determined at call time. If a term evaluates to an undeclared signature, an inconsistent state is reached. With this mechanism, the user of XASM is forced to put redefinitions of signatures at the beginning of an ASM call. During the execution of one ASM, the signature is static, as in traditional ASMs[2].

5.2.1 The $ Feature

The $-feature is explained best by means of an example. Using the $-feature, instead of the declaration and rule

```
function f(_)
f(3) := 5
```

we can write equivalently

```
function $"f"$(_)
$"f"$(3) := 5
```

As a more complex example, we show ASM *Partition* (ASM 16), an algorithm to partition a set of nodes in different universes. Consider as read-only environment functions a universe *N* of nodes and a unary function *Name(_)* denoting the kind of each node[3]. Kinds are simply given as strings. Now ASM Partition declares for each kind a universe and partitions the set of nodes in these universes. The derived universe function *K(_)* calculates the set *K* of all kinds. Then for

[1] In fact the system would also work with arbitrary values, resulting in a system similar to *Mapping Automata* (101), see Appendix B. For our purposes it is general enough to allow only string-values.

[2] In contrast to parameterized XASM, mapping automata allow the user to calculate and change the signature completely dynamically. In fact, mapping automata are defined such that every element is both a value, and a function symbol.

[3] Please remember that ”universe” is the same as a unary relation, and a relation is the same as a function ranging over Boolean, initially defined to produce *false* for each argument.

each string in *K* a universe with that name is declared, using the $-feature. The actual partition is done by the “do forall” rule. Please note that for this example, absence of runtime-errors due to dynamic signature mismatch can be proved, while in the general case this cannot be done.

ASM 16:
```
asm Partition
 accesses universe N
 accesses function Name(_)
is
   derived universe K(k) ==
     (exists n in N: n.Name = k)
   (forall k in K
     universe $k$
   )

  do forall n in N
    $Name(n)$(n) := true
  enddo
endasm
```

5.2.2 The Apply and Update Features

Another problem which has to be solved for parameterized XASM is how to feed an unknown number of arguments to a function. For this purpose we introduce the *Apply* construct, having as arguments a function symbol and arguments represented as a tuple or list. For instance the function application

```
f(t1, t2, t3)
```

can be equivalently written as

```
Apply("f", [t1, t2, t3])
```

or as well as

```
Apply("f",(t1, t2, t3))
```

The reason we allow both kind of syntax is that we want to have a flexible way of passing arguments available in form of lists or tuples to functions whose signature is given using the $-feature.

The rule

```
Apply("f", [t1, t2, t3]) := t
```

is equivalent to

```
f(t1, t2, t3) := t
```

To increase readability we allow as well the following alternative syntax.

```
Update("f", [t1, t2, t3], t)
```

For convenience, Apply can also be used in combination with all built-in functions, as well as unary and binary operators, for instance ”+”, ”-”, e.t.c. The term *1 + 2* can thus be written as *Apply(”+”, [1,2])*.

5.3 Generating Abstract Syntax Trees from Canonical Representations

In Section 5.1 we motivated the need for parameterized XASM by showing that they are needed for an algorithm constructing abstract syntax trees (ASTs) as described in Section 3.2. In this section we give such an algorithm based on the canonical mapping described in Section 4.5.3. The presented AST construction algorithm will be used directly as part of the formal semantics of Montages in Section 8.

5.3.1 Constructing the AST

We assume that a given EBNF has been decorated with canonical mappings as defined in Section 4.5.3 and that the EBNF has been analyzed to define the universe *CharacteristicSymbols* containing all strings corresponding to characteristic symbols in the EBNF, and to define the universe *SynonymSymbols* containing all strings corresponding to synonym symbols in the EBNF. We define the following generic ASM *ConstructCanonicTree* which constructs the corresponding universes, nodes, and selector functions for all possible EBNF definitions. For the sake of simplicity we ignore the ”S1-” and ”S2-” selectors and treat only the ”S-” selectors.

Interface of ConstructCanonicTree

The constructors *characteristic* and *synonym* are used to decompose the argument t, being a canonic representation of the program. The mentioned symbol universes, selector functions, and the *Parent* function must be ”update” accessed, in order to create the AST. This accesses are declared in the following interface of *ConstructCanonicTree*.

ASM 17:
```
asm ConstructCanonicTree(t)
  accesses constructors characteristic(_,_), synonym(_,_)
  accesses universes CharacteristicSymbols, SynonymSymbols
  (for all c in CharacteristicSymbols:
      updates universe $c$
      updates function $"S-"+c$(_)
  )
  (for all s in SynonymSymbols:
      updates universe $s$
      updates function $"S-"+s$(_)
  )
is
  ...
endasm
```

Processing of Synonyms

If the argument t matches the constructor *synonym*, it constructs a tree for the right-hand-side $\&rhs$ of the synonym, adds the resulting root-node n to the corresponding synonym-universe, and returns n as result of the construction.

```
  ...
  if     t =~ synonym(&s, &rhs) then
```

```
      let n = ConstructCanonicTree(&rhs) in
        $&s$(n) := true
        return n
      endlet
   ...
```

Processing of Characteristics

If t matches constructor *characteristic*, the corresponding characteristic universe is extended with a new node n, a node *child* is constructed for all elements t' in the list of right-hand-sides $\&l$, the attribute *Parent* of each *child* is set to node n, and the selector functions of n are defined according to the informations in the right-hand-side terms t'.

```
   ...
   elseif t =~ characteristic(&c, &l) then
      extend $&c$ with n
         n.Name := &c
         do forall t' in list &l
            let child = ConstructCanonicTree(t') in
              child.Parent := n
              if     t' =~ characteristic(&c, &l) then
                 n.$"S-" + &c $ := child
              elseif t' =~ synonym(&s, &rhs) then
                 n.$"S-" + &s $ := child
              endif
            endlet
         enddo
         return n
      endextend
   ...
```

Lists and Options

In Section 4.5.3 we explained that symbols in square option bracket or in curly list-brackets are returning a (possibly empty) list of instances. In Section 3.4 we defined that a list of length 0 is represented in the AST with a specially created node, which is an instance of universe *NoNode*[4], lists with length 1 are represented in the AST with the node representing the unique member, and lists with length 2 or longer are represented in the AST as lists. A list with length one would be treated exactly like its member.

The parts needed to process lists, and options are given as follows. In order to simplify later processing of the tree, a universe *ListNode* containing all lists being part of the AST, and the attribute *Parent* are defined as well. The Interface of *ConstructCanonicTree* is extended with update accesses to universe *NoNode* and *ListNode*. The interface of ASM 17 is refined to the following definition

ASM 18: `asm ConstructCanonicTree(t)`

```
   ...
```

[4] This subtle details results from the fact, that we use constructor terms to represent lists in the AST. As long as we have at least one node inside the list, this works perfectly, but if we have an empty list, it does not have its own identity and would destroy the structure of the AST immediately.

```
  updates universe NoNode, ListNode
is
  ...
endasm
```

and the processing of synonyms and characteristics as described before remains unchanged.

The processing of an empty list creates an element of *NoNode* and returns it as result.

```
  ...
  elseif t =~ [] then
     extend NoNode with n
       return n
     endextend
  ...
```

Otherwise a derived function *ProcessList* is used to construct a tree for each element in the list of constructor terms, and the resulting list of root-nodes is added to universe *ListNode* and returned as result. The *Parent* attribute of each list element is set to the list itself.

```
  ...
  elseif t =~ [& | &] then
     let res = ProcessList(t) in
        ListNode(res) := true
        do forall e in list res
          e.Parent := res
        enddo
        return res
     endlet
  endif
```

The ASM *ProcessList* is given as follows. It constructs for each element of the list a canonic tree, and appends the root of that tree to the local variable r. If the complete list is processed, the list of root-nodes r is returned.

ASM 19:

```
asm ProcessList(l: [NODE])
  accesses function ConstructCanonicTree(_)
is
  function r <- []

if l =~ [&hd | &tl] then
  r := r + [ConstructCanonicTree(&hd)]
  l := &tl
else
  return r
endif
endasm
```

5.3.2 Navigation in the Parse Tree

A very important feature for modeling various structural programming concepts is the possibility to access the *least enclosing instance* of a certain kind of programming language constructs.

The following ASM *enclosing* takes as arguments a node of an AST, and a set of strings, being names of node-universes, and returns the least enclosing node, which is an instance of a universe corresponding to one of the node-universe-names.

ASM 20:
```
asm enclosing(node, setOfUniverseNames)
  (forall s in set setOfUniverseNames
     accesses universe $s$
  )
  accesses function Parent
is
  if node.Parent = undef then
       undef
  else
    if (exists s in setOfUniverseNames:
         $s$(node.Parent))
    then
      node.Parent
    else
      node.Parent.enclosing(setOfUnivNames)
    endif
  endif
endasm
```

The function *enclosing* is a very powerful tool for static semantics definition, since it allows to access directly enclosing statements. The enclosing function is used for name resolution, break/continue statements, exception handling, as well as many aspects of an object oriented type system, such as our Java typing specification in Section D.

Typically, information such as declaration tables or visibility predicates are defined as attributes of the corresponding node, and all enclosed statements for which the information is valid can access it directly via enclosing. Interestingly, the same function enclosing is already used by Poetzsch-Heffter in the MAX system (184; 186). In the MAX case studies this feature is very important to specify all kinds of scoping and name resolution aspects of a language. Both in MAX and in our system, the *enclosing* function allows to simplify the specification of such features by being able to point directly to the least enclosing instance of a certain feature, or the the least enclosing instance of a set of features. In Part III we will use the enclosing-function together with sets of universe names for scopes of variable visibility (Chapter 11) and frames representing jump targets of all kind of abrupt control flow features, such as continues, exceptions, but as well returns from procedure calls (Chapter 14). Simplified versions of such applications are given in the next section.

5.3.3 Examples: Abrupt Control Flow and Variable Scoping

Our first example for navigation in the AST is *abrupt control flow*. Abrupt control flow is a term used for all kinds of control flow not being sequential, but leaving a statement abruptly. Examples of abrupt control flow are breaks and continue jumps out of loops, exceptions, but as well certain aspects of the return from a procedure call. Leaving the statement means climbing up the syntax tree towards the root, resuming the sequential flow in some enclosing statement. For instance, the *break statement* leaves a loop, in order to terminate it and continue after the loop, the *continue statement* leaves a loop in order to start again at the beginning of the loop, exception statements try to find a matching catch clause.

Variable Scoping

The first example is *scoping*. Different constructs like procedure declarations and blocks define a new *scope*. A scope typically opens a new *name space*, and references to functions and variables are resolved first in the least enclosing scope, then in the next outer, and so on. By defining a derived function *Scope* being a set of strings being the scoping-constructs of the described language, the function *enclosing(n, Scope)* can be used to access the least enclosing scope, and typically a binary function *declTable(Node, Ident)* is defined for each scope, mapping the names in the scope's name space to the corresponding entities.

The following ASM *lookUp(Node, Ident)* is following this pattern to look up definitions through the scopes. The first parameter is the reference, and the second the identifier to be looked up.

ASM 21:

```
asm lookUp(node, ident)
  accesses functions Scope, enclosing(_,_), declTable(_,_)
is
  let scopeNode = node.enclosing(Scope) in
    if scopeNode = undef then
      return undef
    else let decl = scopeNode.declTable(ident) in
           if decl = undef then
             node := scopeNode
           else
             return decl
           endif
         endlet
   endif
  endlet
endasm
```

Break and Continue

In the case of breaks and continue, the *enclosing* function can be used to find the least enclosing loop statement, having a matching label. Consider the following grammar of Java loops, coming from Chapter 14:

Gram. 5:	*stm*	=	... | *continueStm* | *breakStm* | *iterationStm* | *labeledStm*
	iterationStm	=	*whileStm* | *doStm*
	continueStm	::=	*"continue" [labelId] ";"*
	breakStm	::=	*"break" [labelId] ";"*
	labelId	=	*id*
	whileStm	::=	*"while" exp body*
	doStm	::=	*"do" body "while" exp ";"*

labeledStm ::= *labelId* “:” *iterationStm*

If a break or continue statement is executed, the following function *getLoop(Node)* takes as parameter a break or continue statement and returns the least enclosing while or do statement, whose label matches the second argument of the function. If the first argument is a continue or break statement whose label is not defined, the least enclosing loop is returned.

ASM 22:

```
asm getLoop(node)
  accesses functions enclosing(_,_), Name(_),
                     S-labelId(_), S-iterationStm(_)
is
  function label <- node.S-labelId.Name

if label = undef then
  return node.enclosing({"whileStm", "doStm"})
else
  let e = node.enclosing({"labeledStm"}) in
      if e = undef then
        return undef
      else if e.S-labelId.Name = label then
             return e.S-iterationStm
           else
             node := e
           endif
      endif
  endlet
endif
```

In Montages such a solution is typically combined with non-local transitions, like the ones showed in the goto-example of Section 3.4.5. In Chapter 14 the control flow of break and continue statements of the imperative core of Java is specified by combining enclosing with non-local transitions. This solution leads to a high level of decoupling. Additional iteration statements can be added without changing the specifications of break, continue, and labeled statement. Other types of abrupt control flow, such as exception handling and procedure calls can be added without changing the specifications. Most interestingly, statements which do not know the concept of abrupt control flow, need not be adapted. The detailed specifications providing this empirical findings are given in Chapters 14.3 and 14.4.

5.4 The PXasm Self-Interpreter

In this section we present an PXasm interpreter *INTERP*, written in PXasm. The special property of this interpreter is, that while interpreting a rule R it accesses and updates the same functions as the direct execution of R does. Given an XASM rule R, the rules R and *INTERP(R)* are equivalent in the sense that given a longer rule P, of which R is a part, the result of replacing R by *INTERP(R)* does not affect the outcome of executing P. This program equivalence property is known as *full abstraction* (78).

We use the introduced techniques to represent PXasm rules as constructor terms, and use the signature of the represented rule to parameterize the interpreter's signature. The interpreter function *INTERP(_)* executes XASM rules according to their semantics. The definition of the constructor term representation of PXasm rules and expression is given in Section 5.4.1. Using this representation the self-interpreter definition is given in Section 5.4.3. As an example for the use of the self-interpreter we refer to Section 6.1 where the definition of a TFSM interpreter is given.

5.4.1 Grammar and Term-Representation of PXasm

To transform PXasm rules into constructor terms, we give the EBNF of PXasm together with a mapping into constructor terms. For the sake of simplicity we completely neglect parsing problems and operator precedence.

Gram. 6:

```
Rule          ::=  { BasicRule }
              =>   BasicRule
BasicRule     =    DoUpdate | Conditional | Let
                   | DoForAll | Choose | Extend | Application
              =>   rhs
DoUpdate      ::=  Symbol [ Arguments ]":=" Expr
              =>   update(Symbol, Arguments, Expr)
Arguments     ::=  "(" Expr { "," Expr } ")"
              =>   Expr
Symbol        =    Meta | Ident
              =>   rhs
Meta          ::=  "$" Expr "$"
              =>   meta(Expr)
Ident         =    [A-Za-z][A-Za-z0-9]*
              =>   Name
Conditional   ::=  "if" Expr "then" Rule
                   ["else" Rule] "endif"
              =>   conditional(Expr, Rule.1, Rule.2)
DoForAll      ::=  "do" "forall" Symbol "in" Symbol [":" Expr]
                   Rule "enddo"
              =>   doForall(Symbol.1, Symbol.2,
                   (if Expr = [] then constant(true) else Expr), Rule)
```

```
Choose        ::=  “choose” Symbol “in” Symbol [“:” Expr]
                   Rule
                   “ifnone”
                   Rule “endchoose”
              =>  choose(Symbol.1, Symbol.2,
                   (if Expr = [] then constant(true) else Expr),
                   Rule.1, Rule.2)
Extend        ::=  “extend” Symbol “with” Symbol
                   Rule “endextend”
              =>  extendRule(Symbol.2, Symbol.1, Rule)
Expr          =    Unary | Binary | CondExpr
                   | Application | Constant | Let
              =>  rhs
Constant      =    “true” | “false” | String | Number
              =>  constant(...corresponding ASM constant...)
Unary         ::=  Op Expr
              =>  apply(Op, [Expr])
Binary        ::=  Expr Op Expr
              =>  apply(Op, [Expr.1, Expr.2])
Application   ::=  Symbol [ Arguments ]
              =>  apply(Symbol, Arguments)
Let           ::=  “let” { LetDef }
                   “in” Both “endlet”
              =>  letClause(LetDef, Both)
LetDef        Symbol “=” Expr
              =>  letDef(Symbol, Expr)
Both          =    Rule | Expr
              =>  rhs
```

Examples The rule of the first example, ASM 1 is represented as follows,

Term 8:

```
[update("x1", [], constant(1)),
 update("x2", [], apply("x1", [])),
 update("x3", [], apply("x2", []))]
```

Accordingly, the rule of example ASM 3 can be rewritten in the following form:

ASM 23:

```
doForall("i",
           "Integer",
           apply("and",
                 [apply(">=", [apply("i",[]),
                               constant(2)]),
                  apply("<",  [apply("i",[]),
                               apply("n",[])])]),
           [update("x",
                   [apply("-",[apply("i",[]),
                               constant(1)   ] ) ],
                   apply("x",[apply("i",[])])
                  )
                  ]
                 )
```

Finally consider the above ASM 16. Its rule represented with constructors looks as follows.

Term 9:

```
doForall("n", "N", constant(true),
         update(meta(apply("Name",
                           [apply("n",
                                  [])
                           ] ) ),
                [apply("n",[])],
                constant(true)))
```

5.4.2 Interpretation of symbols

A symbol in the EBNF grammar is either an identifier, or a meta-constructor, which represents the application of the $-feature. Since Symbols are not XASM rules or expressions, we define a special XASM*SymbolINTERP* which deals only with the Symbol-case.

ASM 24:

```
asm SymbolINTERP(t)
  accesses function INTERP(_)
  accesses constructor meta(_)
is
  if t =~ meta(&s) then
    return INTERP(&s)
  else
    return t
  endif
endasm
```

5.4.3 Definition of INTERP(_)

The interface of INTERP is calculated from the parameter t using the functions

- *MaxArity(t)*, calculating the maximal arity of functions accessed or updated in t,
- *UpdFct(n,t)* providing a comma-separated string listing all n-ary functions updated by t, and finally
- *AccFct(n,t)* providing a comma-separated string listing all n-ary functions accessed by t.

Given these informations, the interface to the 3-ary updated functions can be given as

```
  updates functions with arity 3 $UpdFct(3, t)
```

Interface of INTERP

The interface of INTERP are its parameter t, being the rule or expression to be interpreted, and its access to the functions contained in the lists *AccFct*, the constructors used to represent XASM rules, as well as its update of functions in the lists *UpdFct*. The ASM *SymbolINTERP* is an external function.

ASM 25:
```
asm INTERP(t: Rule | Expr)
 accesses functions UpdFct(_, _), AccFct(_, _), MaxArity(_)
 (forall n in {0 .. MaxArity(t)}:
  updates functions with arity n $UpdFct(n, t)$
  accesses functions with arity n $AccFct(n,t)$
 )
 accesses constructors update(Symbol, [Expr], Expr),
                       conditional(Expr, Rule, Rule),
                       doForall(Symbol, Symbol, Expr, Rule),
                       choose(Symbol, Symbol, Expr, Rule),
                       extendRule(Symbol, Symbol, Rule),
                       constant(Value),
                       apply(Symbol, [Expr]),
                       letClause([LetDef], Rule),
                       letDef(Symbol, Expr)
is
  external function SymbolINTERP(_)
...
```

Interpretation of rules

The interpretation of the XASM rules is relatively straightforward. The components of the rule are evaluated by using recursively the interpreter INTERP. Then depending on the result, the main construct is executed using the corresponding XASM construct. The conditional rule and parallel rule blocks are interpreted as follows.

```
  ...
  if     t =~ [&hd | &tl] then
    return [INTERP(&hd) | INTERP(&tl)]
  elseif t =~ conditional(&e, &r1, &r2) then
    if INTERP(&e) then INTERP(&r1) else INTERP(&r2) endif
  return true
  ...
```

For the update- rule the Update-operator is used and as result the constant *true* is returned.

```
  ...
  elseif t =~ update(&s, &a, &e) then
    Update(SymbolINTERP(&s), INTERP(&a), INTERP(&e))
    return true
  ...
```

In the case of doForall, choose, and extend, the name of the bound variable and the universe are evaluated using *SymbolINTERP(_)* and then the $-operator is used to transform the names into the corresponding symbols.

```
  ...
  elseif t =~ doForall(&i, &s, &e, &r) then
```

```
    do forall $SymbolINTERP(&i)$
                  in $SymbolINTERP(&s)$ : INTERP(&e)
      INTERP(&r)
    endo
    return true
  elseif t =~ choose(&i, &s, &e, &r1, &r2) then
    choose $SymbolINTERP(&i)$ in $SymbolINTERP(&s)$ : INTERP(&e)
      INTERP(&r1)
    ifnone
      INTERP(&r2)
    endchoose
    return true
  elseif t =~ extendRule(&i, &s, &r) then
    extend $SymbolINTERP(&s)$ with $SymbolINTERP(&i)$
      INTERP(&r)
    endextend
    return true
  ...
```

Interpretation of expressions
The interpretation of constants is done by removing the constant-constructor. Please note that the constant-constructor is needed, since a constructor term representing an XASM rule is as well a constant, and it is thus necessary to encapsulate real constants with the constant-constructor.

```
  ...
  elseif t =~ constant(&c) then
    return &c
  ...
```

The interpretation of an application is done with the built-in Apply operator.

```
  ...
  elseif t =~ apply(&o, &a) then
    return Apply(SymbolINTERP(&o), INTERP(&a))
  ...
```

Interpretation of let-clauses
Finally, the parallel let-clause is interpreted, by first interpreting the terms in all let clauses, and then building up recursively a structure of lets. Since we first evaluate all terms, our constructed recursive let-structure correctly interprets the parallel one.

```
  ...
  elseif t =~ letClause(&defList, &r) then
    if &defList =~ [letDef(&p, &t)|&tl] then
      return INTERP(letClause(INTERP(&defList), &r))
    elseif &defList =~ [(&p, &o) | &tl] then
      let $&p$ = &o in
        return INTERP(letClause(&tl, &r)
      endlet
    else return INTERP(&r)
    endif
  elseif t =~ letDef(&p, &t) then
    return (SymbolINTERP(&p), INTERP(&t))
  else return "Not matched"
```

```
endif
endasm
```

We claim that every XASM rule or expression X is equivalent to the XASM rule $INTERP(X')$ where X' is the term-representation of X. The rule (expression) X and *INTERP(X')* are equivalent in the sense that given a longer rule (expression) Y, of which X is a part, the result of replacing X by *INTERP(X')* does not affect the outcome of executing (evaluating) Y. This program equivalence property is known as *full abstraction* (78). The proof of this property would involve a structural induction over rule constructors, and their interpreted versions, calculating their rule and value denotations, and showing that they are the same for both the rule and its interpreted version.

5.5 The PXasm Partial Evaluator

Partial evaluation (108; 46) allows us to specialize PXasm descriptions if some of the access functions in the interface are known to be static. For instance, an interpreter together with a fixed program can be specialized to compiled code. The same technique can be applied to implement Montages. An abstract meta-algorithm is given as semantics of Montages. Applying partial evaluation to this algorithm results in specialized interpreters for the specified languages and, subsequently, for compiled, transparent XASM code for programs written in these languages. This process has already been visualized in Figure 37, and discussed in the introduction of Part II. Parameterization of signature can be used to obtain compiled code whose signature corresponds to terminology introduced by either the language semantics or the program code, allowing us to tailor the readability of the generated code.

In this Section we give some details on how to define partial evaluators using parameterized XASM (Section 5.5.1), and later on in Section 6.4 we show how to apply it to TFSM interpretation.

5.5.1 The Partial Evaluation Algorithm

We give a partial evaluator *PE*, whose arguments are an ASM rule t to be partially evaluated, and a set *sf* of those function symbols which are considered static. For simplicity we assume that *sf* always contains the built in functions and all used constructors, which are static by nature. The decision whether an external function is static can be made by the user under the condition that external functions marked as static are always producing an empty update denotation. If an external function is marked as static, it will be pre-evaluated by our PE-algorithm, independent whether it is really independent from dynamic functions or not. We do not discuss here how external functions could be analyzed, and marked as static by the PE-algorithm. Such analysis would be possible and interesting in the case of external function realized as XASM.

In order to simplify the algorithm, we define *PE* such that partial evaluation of a rule always returns a list of rules, whereas partial evaluation of expressions returns an expression. In the extreme case, the partial evaluation algorithm reduces a rule to an empty list of rules, and an expression to a constant. Typically the outcome is an ASM where the parameterization features are not used anymore and where do-forall and choose rules are replaced with a finite set of simpler rules.

Partial Evaluation of Symbols

We give a special ASM *SymbolPE* covering the partial evaluation of symbols. A symbol is either a string, or the *meta*-constructor representing the $-operator. Partial evaluation of a symbol tries to partially evaluate the argument of the meta constructor, and if the result is a *constant* constructor containing a string, this string is returned.

ASM 26:
```
asm SymbolPE(s: Symbol, sf: set of String)
```

```
 accesses function PE(_)
 accesses constructors meta(_), constant(_)
is
  if s =~ meta(&t) then
    let tPE = PE(&t, sf) in
      if tPE =~ constant(&symb) then
        return &symb
      else
        return meta(tPE)
      endif
    endlet
  endif
endasm
```

Interface of PE

The interface of PE are its access to the constructors used to represent XASM rules. External functions are the above mentioned ASM *SymbolPE*, and later introduced ASMs *ArgumentPE*, *RemoveConstant*, and *InstantiateRules*.

ASM 27:

```
asm PE(t: Rule, sf: set of String)
  accesses constructors update(Symbol, [Expr], Expr),
                        conditional(Expr, Rule, Rule),
                        doForall(Symbol, Symbol, Expr, Rule),
                        choose(Symbol, Symbol, Expr, Rule),
                        extendRule(Symbol, Symbol, Rule),
                        constant(Value),
                        apply(Symbol, [Expr]),
                        letClause([LetDef], Rule),
                        letDef(Symbol, Expr)
is
  external functions SymbolPE(_,_),
                     ArgumentPE(_,_),
                     RemoveConstant(_),
                     InstantiateRules(_,_,_,_)
...
```

Partial Evaluation of Constants

This first case is the simplest case at all. It returns the constant as it is. Thus the following fragment is added to ASM 27:

```
  ...
  if     t =~ constant(&c) then
    return t
  ...
```

Partial Evaluation of Function Application

As a second case function applications are processed. The idea behind partial evaluation of a function application is to partially evaluate the symbol, and the arguments (using ASM *ArgumentPE*), and then to check whether

- the partially evaluated symbol is a string,
- this string is in the set *sf* of static functions, and

- all arguments partially evaluated to constants.

If all this conditions hold, the *RemoveConstant* function is used to transform the argument-list of *constant* constructors into a list of values, and the *Apply* function is used to calculate the result of applying the corresponding function. This result is then wrapped into a *constant*-constructor and returned as result of the partial evaluation.

```
  ...
  if     t =~ apply(&op, &a) then
   let opPE = SymbolPE(&op, sf),
        aPE = ArgumentPE(&a, sf) in
     if opPE isin sf andthen
          (forall a in list aPE: a =~ constant(&)) then
         let argList = RemoveConstant(aPE) in
           return constant(Apply(opPE, argList))
         endlet
     else ...
```

In all other cases an apply constructor with partially evaluated arguments is returned.

```
...  else
       return apply(opPE, aPE)
     endif
   endlet
   ...
```

The above rule uses the ASM *ArgumentPE* to partially evaluate argument lists, and the ASM *RemoveConstant* to remove the *constant* constructor from list of constant arguments. The definitions are given now.

ASM 28:
```
asm ArgumentPE(l: [Expression], sf: set of String)
 accesses function PE(_)
is
  function r <- []
  if l =~ [&hd | &tl] then
    r := r + [PE(&hd, sf)]
    l := &tl
  else
    return r
  endif
endasm
```

ASM 29:
```
asm RemoveConstant(l: [Constant])
is
  function r <-[]
if l =~ [constant(&hd) | &tl] then
  r := r + [&hd]
  l := &tl
else
  return r
endif
endasm
```

Partial Evaluation of Rules

The partial evaluation of updates, rule lists, conditional rules, and extend rules is straightforward. In order to allow for homogeneous processing, our algorithm always returns a list or rules.The following fragment is added to ASM 27.

```
...
elseif t =~ update(&s, &a, &e) then
  let sPE = PE(&s, sf),
      aPE = ArgumentPE(&a, sf),
      ePE = PE(&e, sf) in
    return [update(sPE, aPE, ePE)]
elseif t =~ [&hd | &tl] then
  return PE(&hd, sf) + PE(&tl, sf)
elseif t =~ conditional(&e, &r1, &r2) then
  let ePE  = PE(&e,  sf),
      r1PE = PE(&r1, sf),
      r2PE = PE(&r2, sf) in
    if     ePE = constant(true) then
      return r1PE
    elseif ePE = constant(false) then
      return r2PE
    else
      return conditional(ePE, r1PE, r2PE)
    endif
  endlet
elseif t =~ extendRule(&i, &s, &r) then
  let iPE = SymbolPE(&i, sf),
      sPE = SymbolPE(&s, sf),
      rPE = PE(&r, sf) in
    if rPE = [] then
      return []
    else
      return extendRule(iPE, sPE, rPE)
    endif
  endlet
...
```

Partial Evaluation of Choose

The partial evaluation of choose can only simplify the rule, if the bound variable, and the universe are not meta, if the universe is static, and if for each element of the universe the guarding predicate partially evaluates to either *constant(true)* or *constant(false)*. If there is exactly zero or one elements for which the guard partially evaluates to *constant(true)*, the rule can be simplified. Otherwise, a static set of the elements for which the guard evaluated to true could be constructed. This last simplification is not given here.

```
...
elseif t =~ choose(&i, &s, &e, &r1, &r2) then
  let iPE = SymbolPE(&i, sf),
      sPE = SymbolPE(&s, sf) then
  if sPE isin sf and not iPE =~ meta(&) and
     (forall $iPE$ in $sPE$:
        (let ePE = PE(&e, sf + {iPE}) in
              ePE = constant(true)
           or ePE = constant(false))) then
    if     not (exists $iPE$ in $sPE$:
                  PE(&e, sf + {iPE}) = constant(false))  then
      return PE(&r2, sf)
    elseif (exists unique $iPE$ in $sPE$:
                  PE(&e, sf + {iPE}) = constant(true))  then
      let i0 = (choose $iPE$ in $sPE$:
                  PE(&e, sf + {iPE}) = constant(true)) in
        return (let $iPE$ = i0 in PE(&r1, sf + {iPE}))
      endlet
    else
      return choose(iPE,
                    sPE,
                    PE(&e,sf),
                    PE(&r1,sf),
                    PE(&r2,sf))
    endif
  else
    return choose(iPE,
                  sPE,
                  PE(&e,sf),
                  PE(&r1,sf),
                  PE(&r2,sf))
  endif
...
```

Partial Evaluation of Parallel Let Definitions

The partial evaluation of parallel let definitions tries to find a let definition, where the let-symbol partially evaluates to a string, and where the definition partially evaluates to a constant. If such a let definition is found, consisting of symbol s, defining constant c, and a rule r, the rule can be partially evaluated with the set sf of static function symbol extended by s:

```
let $s$ = c in
  PE(r, sf + {s})
endlet
```

Subsequently the let definition for s can be removed. This is the core of the partial evaluation of let. The remaining parts are concerned with processing the list of let definitions, and reassembling those lets, which cannot be removed.

The first if checks, whether the list of *letDef* constructors is empty. If the list is empty, the partially evaluated rule is returned. Otherwise, in the “then” part of the first if construct, the symbol and the term of the first let are partially evaluated to *pPE* and *tPE*, respectively. If as result from the partial evaluation the symbol is no more meta, and the term did evaluate to a constant, the let clause is removed by partially evaluating the rule, extending the set of static functions *sf* with the symbol *pPE*, and setting the value of *pPE* to the constant *tPE* by a simple let-construct:

```
...
      if (not pPE =~ meta(&)) and tPE =~ constant(&tConst) then
        return PE(letClause(&tl,
                            (let $pPE$ = &tConst in
                               PE(&r1, sf + {pPE}))),
                  sf)
...
```

Otherwise, the non-constant *pPE* and *tPE* are remembered, the rule is partially evaluated with the remaining lets, and at the end the let-definition with *pPE* and *tPE* is added to the rule again. Adding the let definition is done by appending it to the list of parallel lets, if the rule returned from partial evaluation is a let-construct, otherwise a new let-clause with the single let-definition (*pPE*,*tPE*) is created:

```
...
        let rPE = PE(letClause(&tl, &r1), sf) in
          if rPE =~ letClause(&defList2, &r2) then
            return letClause([letDef(pPE, tPE)|
                              &defList2],
                             &r2)
          else
            return letClause([letDef(pPE, tPE)],
                             rPE)
          endif
        endlet
...
```

The full PE-definition for parallel lets is given as follows.

```
...
elseif t =~ letClause(&defList1, &r1) then
  if &defList1 =~ [letDef(&p, &t)|&tl] then
    let pPE = SymbolPE(&p, sf), tPE = PE(&t, sf) in
      if (not pPE =~ meta(&)) and tPE =~ constant(&tConst) then
        return PE(letClause(&tl,
                            (let $pPE$ = &tConst in
                               PE(&r1, sf + {pPE}))),
                  sf)
      else
        let rPE = PE(letClause(&tl, &r1), sf) in
          if rPE =~ letClause(&defList2, &r2) then
            return letClause([letDef(pPE, tPE)|
                              &defList2],
                             &r2)
          else
            return letClause([letDef(pPE, tPE)],
                             rPE)
          endif
        endlet
      endif
    endlet
  else
    return PE(&r1, sf)
  endif
endif
```

Partial Evaluation of Forall Rules

The partial evaluation of a forall rule does a kind of parallel loop unrolling, if the universe of elements is static.

```
...
elseif t =~ doForall(&i, &s, &e, &r) then
  let iPE = SymbolPE(&i, sf),
      sPE = SymbolPE(&s, sf),
      ePE = PE(&e, sf),
      rPE = PE(&r, sf) in
    if ePE = constant(false) then
      return []
    else
      if sPE isin sf and not iPE =~ meta(&) then
        return InstantiateRules(iPE, sPE, ePE, &r)
      else
        return doForall(iPE, sPE, ePE, rPE)
      endif
    endif
  endlet
...
```

The ASM *InstantiateRules* has four arguments, the bound variable i, the universe s, the rule r and the set of static functions *sf*. A local universe *SetCollector* is used to collect an ASM rule for each element in universe s, and a variable *ListCollector* is then used to construct a parallel rule-block from these rules. A variable *trigger* is used to sequentialize the phases for collecting the rules and then building the list representing the rule-block. The interface of the ASM is given as follows.

ASM 30:

```
asm InstantiateRules(i: String,
                     s: String,
                     e: Expr,
                     r: Rule,
                     sf: set of Strings)
      accesses function PE(_,_)
    is
      relation trigger
      universe SetCollector
      function ListCollector <- []
      if not trigger then
        ...
```

The collection of rules is done by a ”do forall”-rule, which ranges i over universe s, and partially evaluates rule r in an environment where i is bound to an element of s and the set of static functions is extended with i.

```
...
do forall $i$ in $s$
  let ePE = PE(e, sf+{i}),
      rPE = PE(r, sf+{i}) in
    ...
```

Depending on whether the guard condition e partially evaluates to a constant or not, the partially evaluated rule is either returned, skipped, or embedded into a *conditional*-constructor. Having processed each \$i\$ in \$s\$, the *trigger* is set to *true*, and the next mode is entered in the else-branch of the outermost if-construct is entered.

```
              ...
              if     ePE = constant(false) then
              elseif ePE = constant(true) then
                SetCollector(rPE) := true
              else
                SetCollector(conditional(ePE,
                                         rPE,
                                         []) := true
              endif
          enddo
          trigger := true
        else
          ...
```

Once relation *trigger* is set to true, a choose rule is fired, which selects an element of universe *SetCollector*, appends it to list *ListCollector* and removes it from *SetCollector*. This choose-rule is repeated until *SetCollector* is empty, then *ListCollector* is returned as result.

```
          ...
          choose r0 in SetCollector
            SetCollector(r0) := false
            ListCollector := [r0|ListCollector]
          ifnone return ListCollector
          endchoose
        endif
      endasm
```

Our algorithm does not check whether the set of static symbols makes sense. A more sophisticated version of the algorithm would try to deduce itself which functions could be static by analyzing which functions are updated, and which are not. Such an analysis, and the partial evaluation of XASM call would result in a more powerful partial evaluator.

5.5.2 The do-if-let transformation for sequentiality in ASMs

In Section 4.1.2 we have shortly discussed how sequentiality is typically modeled in ASM by means of a variable holding the “program counter”. We call such a variable a *sequentialization variable*. Besides the initial example, we have seen many ASMs using such variables. In simple cases such functions could be replaced with a simple sequentiality operator. More interesting are cases where several such variables exist, and the sequential steps are not within a one-dimensional space, but within a space having as many dimensions as there are sequentialization variables. An example for such a more complex case is TFSM interpretation where the variables holding the current node and the current state span a two dimensional space.

We present here a transformation of XASM rules, which takes advantage of information about sequentialization variables, and reformulates an XASM rule in such a way that partial evaluation of the resulting rule will result in a high portion of pre-evaluation, and remarkably simplified rules.

Def. 18: **do-if-let transformation of ASM rules.** *Given the* sequentialization variables $v_1, \ldots, v_n$ *ranging over universes* $V_1, \ldots, V_n$ *and an ASM rule*

$$R(v_1, \ldots, v_n)$$

the do-if-let transformation is defined to be

```
do forall $v'_1 \in V_1, \ldots v'_n \in V_n$
  if $(v_1, \ldots, v_n) = (v'_1, \ldots, v'_n)$ then
    let $v_1 = v'_1, \ldots, v_n = v'_n$ in
      $R(v_1, \ldots, v_n)$
    endlet
  endif
enddo
```

The idea behind this transformation is to enumerate all possible states of the sequentialization-variables in an outermost do-for all. If this do-forall is partially evaluated, the rule is instantiated for each such state. Now by introducing the guard of the if, it is guaranteed, that always only one of the instantiated rules is executed. Thus a flat structure of rules, which are guaranteed to be visited in some sequential order has been created. This rule can then easily be transformed into sequential fast code.

As last step of the transformation a let is introduced, which overrides the definition of the sequentialization-variables, by introducing bound let-variables with the same names. The values of these variables are set to the bound variables of the do-forall loop.

The PE algorithm can now extend the set of static function-symbols *sf* with all bound variables $v'_1, \ldots, v'_n$, and by means of the let-clause, they are renamed into $v_1, \ldots, v_n$, and finally the rule $R(v_1, \ldots, v_n)$ can be partially evaluated at each instance with static definitions of the sequentialization variables.

In Section 6.4 we will show how the do-if-let transformation is applied to compilation of TFSMs .

5.6 Related Work and Conclusions

We have motivated and introduced PXasm by showing that they are needed for situations where a family of related problems exists, but the most natural models for the family members do not share one unique signature. Introducing a unique signature may lead to a natural model of the problem family, but if we are interested in models of the family members, a unique signature is often inappropriate. PXasm are a means for constructing the signature of each family member, as soon as the exact member is determined.

PXasm can therefore be seen as another approach to domain engineering, which we discussed in Section 2.8. In contrast to the domain-specific languages (DSL) approach, PXasm does not allow us to introduce new language features, having a specialized syntax and static semantics. PXasm allows us to mirror with the signature the terminology of the problem-domain. We use this technique in this thesis to describe the meta-formalisms Montages for DSLs, where each problem is a specific DSL which is using the terminology of the corresponding domain.

For a meta-formalism like Montages there are four implementation patterns. The four choices result from the fact that for both the language-description and the program written in the language we have to decide whether a compilation approach, or an interpretation approach is chosen. Even more complexity has to be handled if additional configuration information exists. Again the configuration information can be interpreted at runtime, or compiled into specialized code. If we continue categorizing the full problems, we end up with eight different implementation patterns.

Using partial evaluation all of these patterns can be implemented. Those parts which should be compiled are marked as static, and those which should be interpreted are marked as dynamic. The detailed discussion of partial evaluation and its use to generate interpreters and compilers from Montages descriptions would go beyond the scope of this thesis and we refer to the literature (46; 108). Nevertheless we would like to refer to the work of Bigonha, Di Iorio, and Maia (57; 56) who investigated the general problem of partial evaluation for language interpreters written with ASMs. Combining their advanced partial evaluation techniques with our relatively simple problem of partially evaluating TFSMs may result in very good code.

Since the aim of PXasm is to parameterize the signature of traditional ASMs, we restrict the possible values for the signature-parameters to strings. Partial evaluation can then be used to reduce them back to traditional ASMs. *Mapping Automata (MA)* (101) allow one to use arbitrary elements as signature. While traditional ASMs and PXasm view each dynamic function as set of mappings from locations to values, MA views dynamic functions as objects associated with mapping from attributes to values. Therefore in MA the signature Σ is equivalent to the superuniverse $\mathfrak{U}$. The extend rule can be used to create a new element, and at the same time a new dynamic function is created. The details of MA are given in Appendix B. In contrast to MA, in PXasm the signature Σ is still a static collection of function symbols, but the collection may be calculated while initializing the PXasm. A PXasm is thus an MA, where the signature is restricted to a collection of symbols (string values) which is calculated at initialization and remains static during execution.

As presented, XASM rules must be transformed into constructor terms before they can be interpreted or partially evaluated. A further improvement could be achieved by allowing one to use XASM rules directly as values. Instead of writing the rather unreadable ASM 23 we could then write:

ASM 31:
```
asm P'  is
 function x(_)
 accesses  universe Integer

 INTERP(  "" do forall i in Integer: i >= 2 and i < n
               x(i - 1) := x (i)
              endo
           ""
       )
```

where the quadruple quotes ”” are used to indicate that a rule value is used. Since these rule values correspond to the constructor terms representing the rules, it makes sense to allow pattern matching on such rules. For instance the rather clumsy formulation of partial evaluation of the conditional rule in Section 5.5.1 could be given as follows:

```
  ...
  elseif t =~ "" if &e then &r1 else &r2 endif "" then
    let ePE  = PE(&e,  sf),
        r1PE = PE(&r1, sf),
        r2PE = PE(&r2, sf) in
      if     ePE = "" true "" then
        return r1PE
      elseif ePE = "" false "" then
        return r2PE
      else
        return "" if #PE# then #r1PE# else #r2PE# endif ""
      endif
    endlet
  ...
```

where the # operator is used within quadruple quotes to evaluate rule-values, similar to the way how the $-operator evaluates strings to symbols. The term within the #-operator must evaluate to a rule, which has previously been created with the quadruple quotes, and it is checked that the result is a correct PXasm rule. The double quotes together with the # feature build a so called *template language*, as described by Cleaveland (44; 45). Cleaveland discusses in detail the advantages of a full featured template language. The implementation and design of the above sketched XASM template language, possibly integrating Cleaveland's XML template language, remains for future work.

As well the combination of partial evaluation and parameterized signature can be considered to work like a template language (127). The actual “generation” of the program happens only if the partial evaluation results in a complete evaluation of the signature-parameters, whereas in traditional template languages or the case of the above discussed ””/# features, the content of the templates can always be evaluated. Further our parameterization of signature is integrated with our development language XASM in such a way, that programs can be executed even if partial evaluation did not completely evaluate the parameterized signature. In contrast unevaluated templates are typically not valid programs. Therefore the combination of parameterized signature with partial evaluation could be described as a template-language, which allows for incremental and partial instantiation of templates, and which allows one to execute templates which are fully instantiated, but as well partially instantiated, and not-instantiated templates. The combination of ””/# works more like a conventional template language

XASM has shown to be well suited to our approach to code generation via partial evaluation and signature parameterization, since it has a very simple denotational semantics, and everything is evaluated dynamically. As discussed, in XASM the semantics of the available programming constructs is composed by combining the update-sets and values of sub-constructs; this system is fully referentially transparent, and does not suffer from the side-effects problem in normal imperative languages. Based on such a model, it is easier to use partial evaluation and to add parameterization of signatures, than implementing them on top of an existing language such as C or Java.

6

TFSM: Formalization, Simplification, Compilation

In this section we show in detail the TFSM interpreter (corresponding to the algorithm *Execute* we have given in Section 3.3.5) and how it can be specialized in compiled code by assuming that a given TFSM is static. The partial evaluation of a full Montages meta-interpreter works in a similar way, but the details for the full problem are left for future work. Nevertheless this section serves as well as a more detailed description of the Montages system architecture described in Figure 37.

In Section 6.1 the TFSM interpreter is given in two versions, one for deterministic, and one for non-deterministic TFSMs. The following two sections show how to simplify TFSMs, by eliminating transitions without action rules (Section 6.2, and by partially evaluating action rules and transitions, once a TFSM is built (Section 6.3). Finally in Section 6.4 compilation of TFSMs is discussed, and in the last section of the chapter some conclusions are drawn.

6.1 TFSM Interpreter

In Section 3.3 we have given the construction of TFSMs and in Section 3.3.5 we sketched how they are executed. Given the formalization of the AST we can give now an ASM *Execute* which executes a TFSM. Later in Section 6.4 it will serve as example for the new Montages tool architecture, and finally in Section 8.4.6 it is used as part of the formal semantics of the Montages formalism itself. We repeat the major definitions from previous sections.

The state of TFSM execution is given by two 0-ary, dynamic functions, the current node *CNode* and the current state *CState*. If the state *(n0, s0)* is visited, or in other words if

$$\begin{aligned} CNode &= n0 \\ CState &= s0 \end{aligned}$$

then the action rule associated with *CState* is executed, using fields of *CNode* to store and retrieve intermediate results. Fields are modeled unary dynamic functions. The function

```
function getAction(Node, State) -> Action
```

is defined such, that for each node n, and state t the term *n.getAction(s)* returns the corresponding XASM action represented as constructor term.

Transitions in TFSMs change both the current node and the current state. A TFSM-transition t is defined to tuples having five components, the source node sn, the source state ss, the condition c, the target node tn, and the target state ts.

$$t = (sn, ss, c, tn, ts)$$

In the condition expression c, the source node sn can be referred to as bound variable *src*, and the target node tn as bound variable *trg*. All TFSM transitions are contained in the universe *Transition*.

In the following two sections we give now two variants of a TFSM interpreter, one which can execute non-deterministic TFSMs, e.g. a TFSM where it is possible that several transitions can be triggered, and therefore one has to be chosen nondeterministically, and one interpreter which is specialized for deterministic TFSMs.

6.1.1 Interpreter for Non-Deterministic TFSMs

The interface of ASM *Execute(n,s)* consists of

- the parameters n and s used to initialize the variables *CNode*, and *CState*, respectively,
- the access to universes *CharacteristicSymbols* and *SynonymSymbols*, and subsequently the accesses to the node-universes and selector functions defined by these universes, and finally
- the access to universe *Transitions* containing all transitions of the TFSM, and the access to function *getAction(_,_)* associating all TFSM-states with the corresponding action-rule.

The declaration part defines a boolean variable (or 0-ary relation, in ASM terminology) *fired*, which is switched between true and false, indicating whether we are in step 1 or 2 of the algorithm given in Section 3.3.5. The interpreter *INTERP* is defined as external function, and the two variables *CNode* and *CState* are declared.

ASM 32:
```
asm Execute(n,s)
  accesses universes CharacteristicSymbols,
                     SynonymSymbols,
  (forall c in CharacteristicSymbols:
      accesses universe $c$)
      accesses function $"S-"+c$(_))
  (for all s in SynonymSymbols:
      accesses universe $s$
      accesses function $"S-"+s$(_))
  accesses universe  Transitions
  accesses function  getAction(_, _)
is
  relation  fired
  functions CNode <- n, CState <- s
  external function INTERP(_)
  ...
```

The rule of ASM *Execute* has two parts which are executed in alternation. If *fired* equals false, the first part is executed, interpreting the action rule of the current state, using the *INTERP* function, and providing the correct binding of the *self* variable using a let construct. The first part redefines *fired* to *true* such that in the next step the second part if executed.

```
  ...
  if not fired then
    let self = CNode in
      INTERP(getAction(CNode, CState))
    endlet
    fired := true
  else ...
```

The second part tries to choose a transition, whose source node and state match the current state *(CNode, CState)* and whose condition evaluates to true, if the *src* and *trg* variables are defined to be the current node *CNode*, and the target node of the transition, respectively.

```
  ...
  else
    choose t in Transitions:
          t =~ (CNode, CState, &c, &tn, &ts) and
          (let src = CNode in
             (let trg = &tn in
                 INTERP(&c)))
      CNode  := &tn
      CState := &ts
    ifnone ...
```

If no transition with valid condition is found, a transition with a *default* condition is chosen, and activated. Subsequently the relation *fired* is set to *false*.

```
... ifnone
      choose t in Transitions:
          t =~ (CNode, CState, default, &tn', &ts')
        CNode :=  &tn'
        CState := &ts'
      endchoose
    endchoose
    fired := false
  endif
endasm
```

6.1.2 Interpreter for Deterministic TFSMs

For the later sections reusing the TFSM interpretation algorithm, it is advantageous to transform the non-deterministic form using the choose-construct into a deterministic form using the do-forall-construct. Such a transformation is possible if the provided TFSM is *deterministic*, thus if

- conditions on transitions from the same node/state pair are mutually exclusive and
- there is exactly one transition with *default* condition sourcing in each node/state pair,

Given such a deterministic TFSM we can replace each *default* condition with the negation of the conjunction of all other transitions sourcing in the same node/state pair. The ASM *TransformTransitions* replaces each transition with *default* condition with a transition whose condition is calculated by the ASM *NegateConjunction(_,_)*.

ASM 33:

```
asm TransformTransitions
  updates universe Transitions
is
  external function NegatedConjunction(_,_,_,_)
  forall t1 in Transitions:
      t1 =~ (&sn, &ss, default, &tn, &ts)
    Transition(t1) := false
    let c' = NegatedConjunction(&sn, &ss)
    Transition((&sn, &ss, c', &tn, &ts)) := true
  endforall
  return true
endasm
```

The ASM *NegateConjunction(_,_)* takes as argument a node *sn* and a state *ss*. The ASM has two modes, in the first, where function *trigger* is equal to *false*, a universe *SetCollector* is filled with all transitions whose source node and state are *(sn, ss)* and whose condition is *not* default. If the universe is built up, the algorithm changes in the second mode by setting *trigger* to true.

ASM 34:

```
asm NegateConjunction(sn, ss)
  accesses relation Transition
is
  relation trigger
  universe SetCollector
  function ListCollector <- []

  if not trigger then
```

```
    do forall t in Transitions:
                t =~ (sn, ss, &c, &, &)
         andthen &c != default
      SetCollector(&c) := true
    enddo
    trigger := true
  else ...
```

In the second mode, the transitions in the *SetCollector* are transformed into a list, and then as result of *NegateConjunction* the constructor corresponding to the negated conjunction of this list is returned as result of *NegateConjunction*.

```
  ...
  else
    choose r0 in SetCollector
      SetCollector(r0) := false
      ListCollector := [r0|ListCollector]
    ifnone
      return apply("not", [apply("and",ListCollector)])
    endchoose
  endif
endasm
```

Given the preconditions and after applying the above transformations, we eliminated all *default* transitions and we know that for every TFSM state, at most one transition can be triggered. Under these circumstances the following deterministic ASM can be used, instead of the above non-deterministic ASM 32. The interface is not changed and directly reused from ASM 32.

ASM 35:
```
asm Execute(n,s)
  ...
  if not fired then
    let self = n0 in
      INTERP(getAction(n0, s0))
    endlet
    fired := true
  else
    do forall t in Transitions:
          t =~ (n0, s0, &c, &tn, &ts)
      if (let src = n0 in
            (let trg = &tn in
              INTERP(&c)))
      then
        CNode  := &tn
        CState := &ts
        fired  := false
      endif
    enddo
  endif
endasm
```

6.2 Simplification of TFSMs

The simplification phase applies the TFSM simplification algorithm of Section 3.3.4. The following ASM *SimplifyTFSM* removes all states with empty action rules, as visualized in Figure 27.

The algorithm tries to find two transitions $t1$ and $t2$, such that $t1$ goes from a to b, and $t2$ goes from b to c, and such that intermediate state b is not associated with an action rule. In this case the conditions of $t1$ and $t2$ can be combined, and the two transitions can be replaced with a transition from a to c.

The condition of the new transition is the conjunction of the conditions of $t1$ and $t2$. Since these transitions have different *src* and *trg* nodes, the right *src* and *trg* definitions are fed to them via let-clauses.

ASM 36:
```
asm SimplifyTFSM
  updates universe TRANSISTIONS
  accesses function getAction(_,_)
is
choose t1, t2 in Transitions:
            t1 =~ (&n, &s, &cond1, &n’, &s’)
    andthen t1 =~ (&n’,&s’,&cond2, &n’’,&s’’)
    andthen &n’.getAction(&s’) = []
  Transitions(t1) := false
  Transitions(t2) := false
  Transitions(&n, &s,
              apply("and",
                    [letClause([letDef("src",constant(&n)),
                                letDef("trg",constant(&n’))],
                               &cond1),
                     letClause([letDef("src",constant(&n’)),
                                letDef("trg",constant(&n’’))],
                               &cond2)
                    ]),
              &n’’,&s’’) := true
endchoose
endasm
```

The above algorithm works only if there are no default conditions[1], e.g. deterministic TFSMs where the above ASM 33 has been applied

6.3 Partial Evaluation of TFSM rules and transitions

Show how to apply PE to rules and transitions, taking advantage from the fact that self for the rules, and src/trg for the transitions are static. Further we assume

[1] A second problem is, if there are states where the control may remains for ever, or cycles among nodes without transition rules. Such cycles may again arise the problem that the control may reside there for ever. Since such a cycle has never occurred in our examples, and since we never experimented with examples where it is important that the ”ever remains at same state” behavior is maintained, we do not further treat these cases.

that the selector functions and universes are static. In order to simplify the algorithms we skip the parts which are defining the access interfaces to selector functions, node universes, and which are adding these functions to the sets of static functions provided to the PE-algorithm.

The first ASM *PartialEvaluateTFSMrules(sf)* replaces each action rule with its partially evaluated version, taking as set of static functions those given as argument and *self*. The argument *sf* will typically contain the selector-functions, the node-universes, as well as some static functions defined by the environment. The decision which functions are static, and when to call the partial evaluation is again with the user.

ASM 37:
```
asm PartialEvaluateTFSMrules(sf)
  updates function getAction(node)
  (for all f in set sf
     accesses function $f$
  )
is
  external function PE(rule,staticSet)

  for all n in NODE
    n.getAction :=
      let self = n in
        PE(n.getAction, sf + {"self"})
      endlet
  enddo
endasm
```

The second ASM *PartialEvaluateTFSMtransitions* replaces each transition with a variant where the condition has been partially evaluated, assuming that the term *src* statically evaluates to the source node of the transition, and assuming that the term *trg* statically evaluates to the target node of the transition.

ASM 38:
```
asm PartialEvaluateTFSMtransitions(sf)
  updates universe Transitions
  (for all f in set sf
     accesses function $f$
  )
is
  external function PE(rule,staticSet)

  for all t in Transitions: t =~ (&sn, &ss, &c, &tn, &ts)
    Transitions(t) := false
    let cPE = let src = &sn,
                  trg = &tn in
                PE(&c, sf + {"src", "trg"}) in
      Transitions((&sn, &ss, cPE, &tn, &ts))
    endlet
  enddo
endasm
```

6.4 Compilation of TFSMs

In this section we show the compilation of TFSMs in specialized ASM code. We apply partial evaluation to the the transition rule of *Execute*, given in Section 6.1.2, ASM 35.

As a first step of the compilation we reformulate the original formulation of *Execute* (ASM 35) using the *do-if-let transformation* (Definition 18, Section 5.5.2), taking *CNode* and *CState* as sequentialization variables.

ASM 39:

```
asm Execute(n,s)
  ...
is
  relation  fired
  functions CNode <- n, CState <- s
  external function INTERP(_)
  do forall n0 in NODE, s0 in STATE
    if (CNode, CState) = (n0, s0) then
      let CNode = n0,
          CState = s0 in
        if not fired then
          let self = CNode in
            INTERP(getAction(CNode, CState))
          endlet
          fired := true
        else
          do forall t in Transitions:
              t =~ (CNode, CState, &c, &tn, &ts)
            if (let src = CNode in
                 (let trg = &tn in
                   INTERP(&c)))
            then
              CNode  := &tn
              CState := &ts
              fired  := false
            endif
          enddo
        endif
      endlet
    endif
  enddo
endasm
```

We take the TFSM defined in Section 3.4.5, Figure 33 representing the example program of the goto language given by Grammar 2 and the Montages in Figures 31, 30, and 32. We assume that the TFSM of the example program as well as the rules associated with the states are static. Further, we can see that if the simplification algorithm of Section 3.3.4 is applied consequently, the TFSM of Figure 33 can be further reduced such that all ”initial” and ”go” states disappear. As a consequence the transition relation of our example in Figure 33 is simplified. Introducing the names *Program*, *Const1*, *Print1*, *Const2*, and *Print2* for the remaining AST nodes, a visual representation of the TFSM is given in

Figure 41, and the textual representation of the relation *Transition* is given as the following set, containing five quintuples.

Term 10:

```
{(Program, "I",        true, Const1, "setValue"),
 (Const1,  "setValue", true, Print1, "print"),
 (Print1,  "print",    true, Const2, "setValue"),
 (Const2,  "setValue", true, Print2, "print"),
 (Print2,  "print",    true, Const1, "setValue")}
```

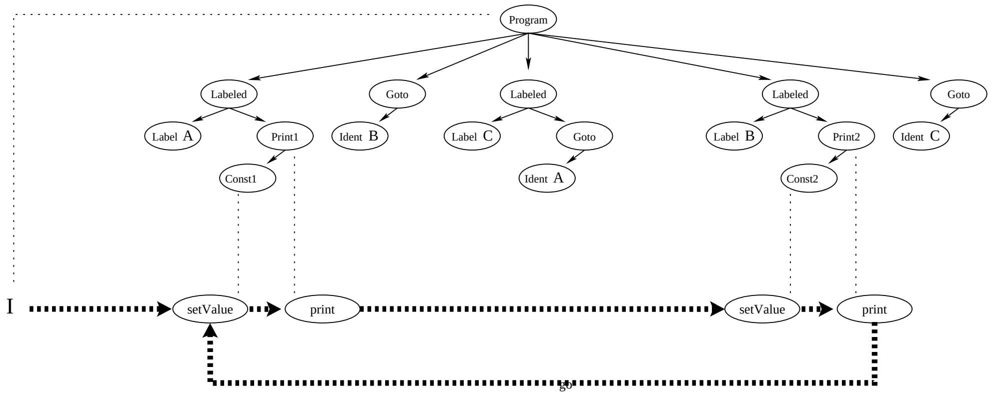


Fig. 41: The simplified version of Figure 33

According to our assumptions, all functions in the interface of ASM *Execute* are static. Now we apply *PE* to the rule of ASM 39. As a result the outermost do-forall is unrolled, the first case being given as follows.

```
if     (CNode, CState) = (Const1, "setValue") then
  let CNode = Const1, CState = "setValue" in
      if not fired then
        let self = CNode in
          INTERP(getAction(CNode, CState))
        endlet
        fired := true
      else
        do forall t in Transitions:
            t =~ (CNode, CState, &c, &tn, &ts)
          if (let src = CNode in
                 (let trg = &tn in
                   INTERP(&c)))
          then
            CNode  := &tn
            CState := &ts
            fired  := false
```

Based on the fact that *Const1* and *"setValue"* are constants, the PE-algorithm is now pushing these constants into the static-let variables *CNode* and *CState* which are overriding the dynamic functions *CNode* and *CState*. As a result the above case is partially evaluated to

```
if     (CNode, CState) = (Const1, "setValue") then
```

```
      if not fired then
        let self = Const1 in
          INTERP(getAction(Const1, "setValue"))
        endlet
        fired := true
      else
        do forall t in Transitions:
            t =˜ (Const1, "setValue", &c, &tn, &ts)
          if (let src = Const1 in
               (let trg = &tn in
                 INTERP(&c)))
          then
            CNode  := &tn  CState := &ts  fired  := false
```

As a simplification, we assume that the actions returned by *getAction* match the signature, and that the partial evaluation of *INTERP(a)* for all involved actions a results in rule $PE(a)$.

The final result of partial evaluation of the above discussed case is

```
  if     (CNode, CState) = (Const1, "setValue") then
    if not fired then
      value(self) := "1"
      fired := true
    else
      CNode  := Print1
      CState := "print"
      fired  := false
    endif
```

End the complete result is the following version of ASM *Execute*, ASM 40.

ASM 40:
```
asm Execute(n,s)
  ...
is
  relation  fired
  functions CNode <- n, CState <- s
  external function INTERP(_)
  if     (CNode, CState) = (Const1, "setValue") then
    if not fired then
      value(self) := "1"
      fired := true
    else
      CNode  := Print1
      CState := "print"
      fired  := false
    endif
  elseif (CNode, CState) = (Print1, "print") then
    if not fired then
      stdout := Const1.value
      fired := true
    else
      CNode  := Const2
      CState := "setValue"
      fired  := false
    endif
  elseif (CNode, CState) = (Const2, "setValue") then
    if not fired then
      value(self) := "2"
      fired := true
    else
      CNode  := Print2
      CState := "print"
      fired  := false
    endif
  elseif (CNode, CState) = (Print2, "print") then
    if not fired then
      stdout := Const2.value
      fired := true
    else
      CNode  := Const1
      CState := "setValue"
      fired  := false
    endif
  endif
endasm
```

6.5 Conclusions and Related Work

While our intention is to use PXasm for the semantics of Montages, we have shown in this chapter their usefulness for a TFSM interpreter and the compilation of TFSMS. The presented TFSM interpreter is the nucleus of the later presented Montages semantics, and the described compilation of TFSMs by means of partial evaluation shows the principles behind the new implementation of Montages. Using the same approach the later presented Montages semantics can be reduced to a specialized interpreter, and a program can be compiled to specialized XASM code.

The presented simplification and compilation allow for an efficient implementation of Montages based on our novel concept of TFSM. Further other meta formalisms can use TFSM as their virtual machine. In fact the basic ideas for TFSMs have been developed by the author while designing a different, XML based meta-specification formalism for the company A4M (126).

A very interesting field of development related to TFSMs are *model driven architectures*, proposed by the OMG group as successor of UML (25; 170). These architectures, which are driven by a model of the problem to be solved, are closely related to domain-engineering. DSLs are considered an important part in such architectures, and many UML based ways for defining such DSLs are discussed (43; 148). Montages, which combines ASTs of DSLs, and state-machines whose states are decorated with actions, may be a good candidate for such definitions: UML already uses such state machines for defining methods of classes, and using the same notation for defining semantics of DSL constructs may be natural. In order to examine this possibility we will redefine Montages based on UML's variant of state-machines and action-languages. The precise definition of such UML action-languages allows for executable variants of UML (152; 205) and integrating these technologies with Montages will help to move Montages into the domain of practicable software-engineering tools. Interestingly the proposed action languages (2; 229) have many similarities with XASM.

7

Attributed XASM

The description of main-stream programming languages with Montages (225; 98) has shown the need for a feature corresponding to *attribute grammars (AG)* (122). In fact, the experiments showed that the complexity of static semantics of a language like Java or C cannot be handled with a methodology less powerful than AGs. The simplicity of Montage's initially proposed one-pass technique (133), earlier combinations of AGs with ASMs (184; 186), and a proposal for extending AGs with reference values (89) have inspired us to design a new kind of AGs using XASM. The definition of this attribute grammar variant is based on a very simple mechanism called *Attributed* XASM or short *AXasm*.

The motivation for and introduction of AXasm is given in Section 7.1. In Section 7.2 Formal semantics of AXasm is given in three ways,

- by translating attributions into derived functions (Section 7.2.1),
- by extending the denotational semantics of XASM with attribution features (Section 7.2.2), and finally
- by extending the self-interpreter to full AXasm (Section 7.2.3).

The self-interpreter of AXasm is later used in Chapter 8 as part of the Montages semantics. Finally in Section 7.3 we shortly compare AXasm with traditional attribute grammars, and refer to related work. As example we combine in Appendix D attributions with abstract syntax trees, specifying an attribute grammar for the type system of the Java Programming Language.

7.1 Motivation and Introduction

If we compare object-oriented (OO) programming with procedural programming and attribute grammars (AGs) with functional programming, we find some interesting commonalities of the two relations. Both OO programming and AGs feature some sort of *dynamic binding* which allows to associate code with data, and to use this association to choose dynamically the right code for each kind of data. In OO programming the code comes in form of *procedures* changing the state, and in AGs, the code comes in form of *function definitions* calculating a result from the arguments. In both cases, the code is not directly associated with the data elements, but with types of data. In OO programming the types are called classes, and the procedures associated with classes are called methods. In the case of AGs, the types are the labels of the abstract syntax tree (AST) nodes, and the functions are called attributions.

This section contains a motivation of the AXasm design based on the comparison of the mentioned paradigms, object-orientedness, functional programming, and attribute grammar. The only purpose of our discussion is the motivation of AXasm, for the more in depth discussion of the topic we refer to the existing literature (174).

In Section 7.1.1 we compare OO programming to procedural programming and in Section 7.1.2 AGs and functional programming are related to each other. The commonalities of OO programming and attribute grammars are analyzed in Section 7.1.3, and in Section 7.1.4 we introduce AXasm, which achieves some of the same advantages as the other two approaches by adding dynamic binding to derived functions of XASM. Some features make AXasm look more like OO programs than AGs: attributes may have several parameters, and the values of attributes can be other elements having attributes. Further, using the extend construct, it is in principle possible to create new instances dynamically. Nevertheless in the context of Montages we will mainly use AXasm to simulate the behavior of traditional AGs.

For simplifying the presentation we define only dynamically bound derived functions, and do not introduce dynamically bound functions of other kinds. Therefore in our definition of AXasm, the elements have no local state. We do not forbid that attributes are evaluated at runtime, but we concentrate on the case that attributes are evaluated before runtime in order to check static semantics. Partial evaluation of Montages specification is more effective in the case of attributions evaluated before run time, and typical optimization of programming language implementations, such as static typing rely on pre run-time evaluation of attributes.

7.1.1 Object-Oriented versus Procedural Programming

The transition from *procedural* programming to OO programming has led to an increased productivity in software development. One of the reasons for this is that OO programming supports directly the modeling of a system as a number of *object-classes* whose instances share behavior and state structure. The behavior is given by methods, which may be differently implemented for different classes. If a method is applied to some value, the type of this value determines dynamically which method implementation is bound to the call. This feature is called *dynamic binding*

More detailed, the objects in a class are called its *instances*. The class of which an object is an instance is called its *type*. Each class has a number of variables associated, as well as a number of procedures. The variables of a class are called its *fields* and the procedures of a class are called its *methods*. Two classes may share the same fields and methods names, but each of them may define them differently. Given a method ***m***, classes A, and B, the m definition of A typically fits A-instances, and the m definition of B fits B-instances. If m is applied to some variable which may hold A or B objects, the type of the actual object determines which definition is applied. The following OO pseudo code shows a call of method m on variable o. Depending on the type of the value of o, either the A or B definition of m is executed.

```
class A
method m
  begin
    m-definition of A
  end
endclass

class B
method m
  begin
    m-definition of B
  end
endclass

call o.m
```

The same result can of course be achieved using a procedural programming language. The following procedure m executes the A-definition of m if the parameter $self$ of m is an A-instance, and the B-definition if the parameter is a B-instance. The call ***m(o)*** will thus result in the execution of either the A or B definition of m, depending whether o evaluates to an A or to a B instance.

```
procedure m(self: OBJECT)
begin
  if     self is A-instance then
    m-definition of A
  elseif self is B-instance then
    m-definition of B
  endif
end

call m(o)
```

The power of OO programming comes into play, if a third class C is added. In the procedural implementation, the definition of the unique m procedure has to be extended with the cases covering C instances. Thus the full source code has to be changed. In the OO style, simply a third class C is added to the system, and classes A and B are not touched. This is a little advantage, if we look at toy examples, but it is crucial, if realistic software systems are developed. Typically in realistic software system it is very hard to change existing code, since many other system components may rely on it.

Before we show how to add dynamic binding to XASM, we analyze functional programming and attribute grammars. It will be shown that attribute grammars can be considered a dynamically bound version of functional programming.

7.1.2 Functional Programming versus Attribute Grammars

Programs represented in the form of ASTs can be conveniently analyzed by decorating their nodes with properties of the corresponding programming construct. Many of such node-properties, such as static type, arguments, or constant value can be expressed as expressions over properties of other nodes in the AST. If the grammar is stable, and if the existing rules are known, a solution using *functional programming*, where each property is modeled as a function can be given as follows. Consider for instance a grammar with symbols A, B, and C and corresponding expressions defining the property *staticType*. The following functional definition of staticType can then be applied to calculate the static type of a node :

```
staticType(self: NODE) ==
  (if self is A node then    staticType-definition of A
   else
  (if self is B node then    staticType-definition of B
   else
  (if self is C node then    staticType-definition of C
   else
   undef)))

staticType(n)
```

Depending whether n is an A, B, or C node, the corresponding definition of *staticType* is evaluated. Unfortunately this solution is only feasible, if the grammar and rules are known, and if the grammar is not changing over time. This assumption is not realistic for real-world languages, or for the design process of new domain-specific languages. Therefore a notation which allows to add new definitions without changing the existing ones is needed.

A solution to our problem is provided by AGs, which allow to give the property definitions for each grammar symbol. Similar to OO programming, dynamic binding is used to evaluate the attributes. A formulation of the above property or *attribute* staticType in AG style is given as follows.

```
rule A ....
  attribute staticType == staticType-definition of A

rule B ...
  attribute staticType == staticType-definition of B
```

```
rule C ...
  attribute staticType == staticType definition of C
```

```
n.staticType
```

If a new kind of nodes D is added to the definition, the AG style allows to simply add the rule for D, while the functional style urges us to change the definition of the unique function *staticType*.

7.1.3 Commonalities of Object Oriented Programming and Attribute Grammars

If we try to analyze the commonality of both OO programming and AGs, we can identify the following points:

- The involved elements (objects or nodes) are typed by universes of elements (object-classes or nodes with the same label). The type of an element is determined at its creation (instance creation or production rule application) and is never changed.

- Expressions are *dynamically typed* by the element they evaluate to.

- The same function (method or attribute) can be differently defined for each of the type-universes; definitions are associated with these universes.

- The dynamic type of the first argument of a function-call (method-call or attribute-evaluation) is used to *dynamically bind* the call to the corresponding function definition (method definition, attribution).

- The first argument of a function which is used for dynamic binding is written before the function, using the dot notation, and within the function-definition this argument can be uniformly accessed with the symbol *self*[1].

In the next section these common features of OO programming and attribute grammars are added to the semantics of derived functions in XASM resulting in AXasm.

7.1.4 AXasm = XASM + dynamic binding

Dynamic binding allows to give specialized implementations of the same method for different classes, or of the same attribute for different node types. If a method is called, or an attribute evaluated, the type of the first argument, the so called *self* or *context* object, determines which implementation is chosen. In order to make the syntax more explicit, this first argument is typically written

[1] This is a simplification, since in each formalism this element is accessed with a different syntax, for instance *this* instead of *self*, and in many formalism it is even considered as an implicit argument.

in front of a dot. Given an attribute or method f, parameters $p_1, \ldots, p_n$, the call or evaluation of f, with context given by expression p_0, is written as follows:

$$p0.f(p1, ..., pn)$$

The type of p_0 determines dynamically which implementation is chosen for f, and within the code of f, the term *self* can be used to refer to the value of p_0. A subtle detail is that the arguments $p_1, \ldots, p_n$ are not expected to be evaluated with respect to the new context, but with respect to the outermost context, in which p_0 has also been evaluated. Therefore not only the context-object *cc*, but as well the *outermost context object (oc)* must be known to evaluated such terms.

As motivation consider the following term.

$$t_0.f_1(t_{1_1}, ..., t_{1_n}).f_2(t_{2_1}, ..., t_{2_m})$$

The two attributes f_1 and f_2 are naturally evaluated with respect to the context object defined by the terms before the ”dot”. On the other hand, it seems more natural that the arguments $t_{1_1}, \ldots, t_{1_n}, t_{2_1}, \ldots, t_{2_m}$ should be evaluated in the same context as the initial term t_0. Therefore the ”outermost” context-object must be passed through the calculations, and used whenever parameters are evaluated.

In order to introduce dynamic binding in XASM, we need a typing of elements. The idea of AXasm is to use an arbitrary set of disjoint universes to ”type” elements. Given such a partition, the *type* of an element is given by its membership in one of the universes. The type of an expression is dynamically determined by evaluating the expression. For each of these universes we allow the definition of *attributes*, a special kind of derived functions.

One possibility to guarantee disjointness of universes is to use only the extend function to populate them. For instance the ASM ConstructCanonicTree (ASM 17, Section 5.3.1) uses only extend-rules to populate the characteristic universes. Therefore these universes are disjoint and build a partition called the *characteristic partition*. This partition is used to combine AXasm with ASTs, resulting in the AG system of Montages.

An example for attribute definitions is the following declaration of universe *U_0*, given in concrete syntax.

ASM 41:
```
universe U_0
  attr a_1(p_1_1, p_1_2, ..., p_1_n1) == t_1_1
  attr a_2(p_2_1, p_2_2, ..., p_2_n2) == t_1_2
  ...
  attr a_m(p_m_1, p_m_2, ..., p_m_nm) == t_1_m
```

As mentioned the interpretation of each rule or expression of an attributed XASM is depending on a *context-object (cc)* and an *outermost context object (oc)*. A function γ maps context objects to the corresponding context universe definitions. The context-object itself is always accessible as function *self*.

Evaluating a function application

$$f(t_0, \ldots, t_n)$$

with respect to (cc, oc), first the parameters are evaluated with respect to (oc, oc), resulting in elements $e_0, \ldots, e_n$; then the attribute f is searched in the definitions of the context $\gamma(cc)$. If such an attribute definition is present, and the numbers of formal parameters match, the definition is evaluated with actual parameters $e_0, \ldots, e_n$, where during the evaluation of *f* the symbol *self* refers to cc, and terms are evaluated with respect to (cc, cc). Otherwise a function in the global context is searched, where all global dynamic functions, ASMs, constructors, and derived functions reside.

The dot-notation can be used to interpret an expression in the context given by another expression. Evaluating

$$t_0.t_1$$

with respect to *(cc, oc)*, the term t_0 is evaluated with respect to the same objects, evaluating to element e_0, and then t_1 is evaluated with respect to the new context-object e_0 and the old outermost context-object oc. The result of this second evaluation is the result of the complete dot-term.

7.1.5 Example

As an Example consider the following definitions, introducing global functions a, x, universe U, having attributes a, b, and a rule extending U.

```
function a <-1, x

universe U
  attr a == 3
  attr b == x.a

extend U with u
  x := u
  a := a+ u.b
endextend
```

First step
The rule within the extend clause updates the global function x to the new element u. In the next update the global function a is updated to it's value 1 plus the value of b in the context of u. Since u is created as element of U, the context of u is U, and therefore b is identified as an attribute of U. The definition of attribute b is $x.a$. Since x is initially *undef*, the a of $x.a$ is initially evaluated in the global context. In the global context, a is initially 1, thus the result of $x.a$ and thus of attribute b of u is 1. Thus the global a is updated to 2.
Second step
After this first step, the value of x is the newly create U instance, the value of a is 2. In the second step, again a new instance of U is created. The global function

a is incremented with the value of attribute b of the new element. In contrast to the first step, this time the attribute b evaluates to 3, since x is no more *undef* but evaluates to an element being member of universe U. The evaluation of term $x.a$ results thus in evaluation of a in the U context, where a is an attribute with constant value 3. After the second step, the global a is set to 5, and x is set to the second newly created element.

In all following steps, new U instances are created, and a is incremented with 3.

7.2 Definition of AXasm

In this section the formal semantics of AXasm is given in three ways. Section 7.2.1 explains AXasm by translating the dynamically bound derived functions into standard derived functions of XASM, following the pattern in Section 7.1.2 where the functional counterpart of an attribute grammar has been shown. A semantics without the help of a syntactical transformation is given in Section 7.2.2 where the denotational semantics of XASM, presented in Definition 9, Section 4.3 is extended to AXasm. Finally in Section 7.2.3 we extend the XASM self-interpreter of Section 5.4 to a self-interpreter of AXasm. Such a self interpreter will be used in situations where the attributions are not known in advance, for instance the definition of the Montages meta-interpreter needs an AXasm self-interpreter.

7.2.1 Derived Functions Semantics

We look at a more general example of attributions and explain their meaning by expressing them as an equivalent derived function. In the following attributed XASM, symbols $U_1, \ldots, U_n$ are used for universes, symbols $a_1, \ldots, a_m$ are used for attributes, the terms t_{i_j} are defining the attributes, and finally R is the transition rule.

ASM 42:

```
universe U_1
  attr a_1 == t_1_1
  attr a_2 == t_1_2
  ...
  attr a_m == t_1_m

universe U_2
  attr a_1 == t_2_1
  attr a_2 == t_2_2
  ...
  attr a_m == t_2_m

...

universe U_n
  attr a_1 == t_n_1
  attr a_2 == t_n_2
  ...
  attr a_m == t_n_m

R
```

The given definitions of attributes $a_1, \ldots, a_m$ can be transformed into an equivalent non-attributed XASM with m derived functions $a_i, i \in \{1, \ldots, m\}$. The definition of this function applies the attribute definitions, depending on the value of the context-object *self*. Instead of the dot-notation $t_1.t_2$, an explicit *Dot(_,_)* function must be used. $Dot(t_1, t_2)$ evaluates first t_1 and then makes the result of this evaluation available as context object *self* in the evaluation of t_2. The result of this t_2 evaluation is the result of $Dot(t_1, t_2)$.

```
derived function Dot(t1, t2) == (let self = t1 in t2)
```

Following this approach standard XASM declarations being equivalent to the above attributed XASM can be given as follows.

```
universes U_1, U_2, ..., U_n
function self

derived function a_1 ==
  (if      U_1(self) then t_1_1
  else (if U_2(self) then t_2_1
  ...
  else (if U_n(self) then  t_n_1
  else   undef ) ...))

derived function a_2 ==
  (if      U_1(self) then t_1_2
  else (if U_2(self) then t_2_2
  ...
  else (if U_n(self) then t_n_2
  else   undef ) ...))


derived function a_m ==
  (if      U_1(self) then t_1_m
  else (if U_2(self) then t_2_m
  ...
  else (if U_n(self) then t_n_m
  else   undef ) ...))
R
```

where all dot-applications in R and the terms t_{i_j} are rewritten using the derived function *Dot*, e.g. $t1.t2$ is replaced by *Dot(t1, t2)*

7.2.2 Denotational Semantics

From a denotational point of view, an attributed XASM (AXasm) describes an XASM where elements have local signatures, and the dot-notation can be used to evaluate a term in an other elements signature. Binding of function evaluation is thus done dynamically. The purpose of this section is to extend the denotational semantics of XASM, as given in Definition 9, Section 4.3.

At this moment we would like to note that we have chosen the definitions such, that they are suited as well for object based ASM (102), or even fully object oriented ASMs (128). Here we will restrict us to Attributed XASM, where objects have local signatures, but no local states.

On the other hand, since we formally introduced derived functions as special kinds of XASM calls (see Section 4.4.3), we will cover full ASM call functionality for the attributes. These correspond semantically to methods of OO systems, but to avoid confusions we call them *attributions*. A real OO system would be obtained by introducing local states, as in ObASM (102) and by introducing inheritance. Although these features would help to structure further the case studies, we decided to abstract from them in order to shorten the material.

In AXasm the elements of the superuniverse are typed by the so called *classes*, represented as universes.

Def. 19: Element Partition and Class Association. *Each element* $x \in \mathfrak{U}$ *has an associated* universe $\gamma(x)$ *from a set* $\mathfrak{C}$ *of disjoint universes. The element* x *is member of* $\gamma(x)$ *and not member of any other universe in* $\mathfrak{C}$*. We call the universe* $\gamma(x)$ *the* class *of* x*, and* x *is said to be an* instance *of* $\gamma(x)$*. The elements* undef, true *and* false*, as well as other built in constants, and constructor terms are the members of the so called global or main class* main .

Since the elements have no local state, the signature Σ of dynamic functions is not split into local signatures, the state is still a mapping from Σ to actual definitions of the functions. The class c of an element determines a local extended signature Σ_c. The global extension signature is considered to be the local extension signature of the global class *main*.

Def. 20: Local Extended Signature and Classes. *Associated with each class* c *is a set* $\Sigma_{ext}(c)$ *of* attributions *defined within the definition of* c*. The global extension signature* Σ_{ext} *corresponds to* $\Sigma_{ext}(main)$*.*

The extended signature Σ' *with respect to an object* o *is*

$$\Sigma'(o) = \Sigma \cup \Sigma_{ext}(\gamma(o))$$

Terms in AXasm can be built over the union of all extended signatures, but in the context of an object o, only terms over $\Sigma'(o)$ can be defined, all others are undefined.

The attributions are derived functions defined locally for each class c. Formally, a derived function can be represented as a tuple of an expression, and the formal parameters.

Def. 21: Attributions. *The attributions are given by a family* $\mathcal{A}$ *of mappings. For each class* c*,* $\mathcal{A}(c)$ *maps the n-ary symbol* f *of* $\Sigma_{ext}(c)$ *to a (n+1) tuple*

$$f_{\mathcal{A}(c)} = (E, p_1, ..., p_n)$$

where E *is an expression, and* $p_1, \ldots, p_n$ *are the formal parameters.*

In concrete syntax, the definition of f in c would be given as follows.

```
universe c
...
  derived function f(p1, ..., pn) == E
  ...

universe...
```

In summary, an AXasm is given by a transition rule R, signature Σ of dynamic functions, a set of disjoint universes $\mathfrak{C} \subset \Sigma$ whose interpretation in each state builds a partition of the elements in $\mathfrak{U}$, a family of attributions (local external functions) $\Sigma_{ext}(c)$ for each class c and a family of mappings $\mathcal{A}$ giving the definitions of the attributions.

Def. 22: AXasm. *An AXasm is given by a quintuple,*

$$(R, \Sigma, \mathfrak{C}, \Sigma_{\text{ext}}, \mathcal{A})$$

- *the transition rule R,*
- *signature Σ of dynamic functions,*
- *a set of disjoint universes $\mathfrak{C} \subset \Sigma$ whose interpretation in each state builds a partition of the elements in $\mathfrak{U}$,*
- *a family of attributions (local external functions) $\Sigma_{\text{ext}}(c)$ for each class c, and*
- *a family of mappings $\mathcal{A}$ giving the definitions of the attributions.*

Given an AXasm, the mapping $\gamma(x)$ from elements to classes can be calculated in each state λ by

$$\gamma(x) = c \textit{ where } x \in \lambda_c$$

Current and outermost context
In AXasm rules and terms are evaluated with respect to a context given by two elements. The first element is the *current context*, referred to as *cc*, and the second one is the *outermost context*, referred to as *oc*. In the initial state of an AXasm, both *cc* and *oc* are equal to *undef*, and since *undef* is member of the *main* class, rules and terms are evaluated with respect to the main, or global context. Global external functions are considered the attributes of this global context, and in the global context, the behavior of an AXasm is the same as the behavior of a normal XASM.

Update and value denotations of AXasm constructs
In order to extend XASM's denotational semantics to full AXasm, the signature of value denotation (Definition 6) and update denotation (Definition 1), as well as external update and value denotations (Definition 8) must be extended with two additional arguments, the current context *cc* and the outermost context *oc*.

Def. 23: Denotations with Context. *With respect to the current context* cc *and the outermost context* oc, *the update and value denotation of a rule* R *in a state λ is given by*

$$\textit{Upd}(R, \lambda, cc, oc)$$
$$\textit{Eval}(R, \lambda, cc, oc)$$

and the denotations of an external function $\underline{f}$ with actual parameters $(e_1, \ldots, e_n)$ are given by

$$\textit{ExtUpd}(\underline{f}, (e_1, \ldots, e_n), \lambda, cc)$$
$$\textit{ExtEval}(\underline{f}, (e_1, \ldots, e_n), \lambda, cc)$$

Semantics of self

The update denotation of term *self* is the empty set, and the value denotation of term *self* is the current context object *cc*.

Def. 24: **AXasm *self* evaluation.** *if* R = self
then
$\mathit{Upd}(R, \lambda, cc, oc) = \{\}$
$\mathit{Eval}(R, \lambda, cc, oc) = cc$

Semantics of attributions

If an external function is realized with an ASM (Definition 14 and 16), the new argument is added to the state as value of *self*, and it is used as initial current and outermost context objects of the ASM[2]. The denotations *ExtUpd* and *ExtEval* are given as follows.

Def. 25: **Update and Value Denotations of Attributions.** *Given current context* cc *and n-ary attribute* $f \in Sigma_{\mathit{ext}}(\gamma(cc))$, *and the definition*

$$f_{\mathcal{A}(\gamma(cc))} = (E, p_1, ..., p_n)$$

the ExtUpd *and* ExtEval *functions are given as follows:*

$$\begin{aligned} \mathit{ExtUpd}(f, (e_1, \ldots, e_n), \lambda, cc) &= \\ \mathit{Upd}(E, \lambda \cup \{\mathit{self} \mapsto \mathit{cc}, p_1 \mapsto e_1, \ldots, p_n \mapsto e_n\}, cc, cc) & \\ \mathit{ExtEval}(f, (e_1, \ldots, e_n), \lambda, cc) &= \\ \mathit{Eval}(E, \lambda \cup \{\mathit{self} \mapsto \mathit{cc}, p_1 \mapsto e_1, \ldots, p_n \mapsto e_n\}, cc, cc) & \end{aligned}$$

Semantics of function application

If a function application

$$f(t_1, ..., t_n)$$

is evaluated with respect to (cc, oc), the arguments of the application are evaluated with respect to (oc, oc), and the function f itself is evaluated with respect to the single context element cc. The class $\gamma(cc)$ determines the local signature of external functions (attributes) $\Sigma_{\mathit{ext}}(gamma(cc))$. If $f \in \Sigma_{\mathit{ext}}(gamma(cc))$ the definition of this external function (attribute) is applied, as defined above, otherwise the dynamic function $f \in \Sigma$ is evaluated.

Within the evaluation of f, the current context cc can be referred to as term *self*.

Def. 26: **AXasm Function Evaluation.**
if $R = f(t_1, \ldots, t_n)$
where $t_1, \ldots, t_n$ *are terms*
and $e_1 = \mathit{Eval}(t_1, \lambda, oc, oc)$ *and* ... *and* $e_n = \mathit{Eval}(t_n, \lambda, oc, oc)$

[2] Since for our purpose we purpose we use AXasm only with attributions being derived functions, we are not giving the details of the refined definitions for the general ASM call.

```
then
  if $f \in \Sigma_{\mathit{ext}}(\gamma(cc))$
  then
    $Upd(R, \lambda, cc, oc) = ExtUpd(f, (e_1, \ldots, e_n), \lambda, cc)$
                      $\cup \quad \bigcup_{i \in \{1,\ldots,n\}} Upd(t_i, \lambda, oc, oc)$
    $\mathit{Eval}(R, \lambda, cc, oc)(R) = ExtEval(f, (e_1, \ldots, e_n), \lambda, cc)$
  else
    $\mathit{Upd}(R, \lambda, cc, oc) = \bigcup_{i \in \{1,\ldots,n\}} Upd(t_i, \lambda, oc, oc)$
    $\mathit{Eval}(R, \lambda, cc, oc) = f_\lambda(e_1, \ldots, e_n)$
```

Semantics of dot application
The dot notation is used to change the current context and allows to evaluate external functions of other classes. For instance the term

$$t_0.t_1$$

is evaluated with respect to (cc, oc), evaluates first t_0 with respect to (cc, oc) to element e_0, and then evaluates t_1 with respect to (e_0, oc).

Def. 27: AXasm Dot Term Evaluation. *if* $R = t_1.t_2$

```
  where $t_1, t_2$ are terms
  and $e_1 = \mathit{Eval}(t_1, \lambda, cc, oc)$
then
  $Upd(R, \lambda, cc, oc) = Upd(t_1, \lambda, cc, oc) \cup Upd(t_2, \lambda, e_1, oc)$
  $\mathit{Eval}(R, \lambda, cc, oc) = \mathit{Eval}(t_2, \lambda, e_1, oc)$
```

In all other cases the definitions of Definition 9 remain valid, except that the additional arguments *cc* and *oc* are passed as well.

7.2.3 Self Interpreter Semantics

In this section the formal semantics of AXasm is given by extending the definition of the PXasm self-interpreter INTERP such that it takes the context-object as additional argument, and evaluates both normal functions and references to attributes. We assume that the current list of attributions is available as constructor term, being assigned to the global 0-ary function *AttrDefs*. In Section 7.2.3.1 the mapping from attribute definitions in constructor terms is defined. Then we explain first an interpreter for attributions without parameters (Section 7.2.3.2) and then we extend the definitions to a self interpreter for attributions with parameters (Section 7.2.3.3).

7.2.3.1 Constructor Term Representation of Attributes
The attributions are provided in the form of constructor terms, built up from the constructors

attrDefs(Ident,[Attribute])

whose first argument is the name of the universe, and whose second argument is a list of attributions valid for that universe. Each attribution is a three-ary constructor

attribute(Ident, [Ident], Expr)

whose arguments are the name of the attribute, a list of parameters, as well as a term-representation of the XASM-expression defining the attribute. The initial example of an attribution, ASM 41 is represented using the introduced constructors as follows,

Term 11:
```
attrDefs("U_0",
     [attribute("a_1",
                [p_1_1, p_1_2, ..., p_1_n1],
                MOD(t_1_1)),
      attribute("a_2",
                [p_2_1, p_2_2, ..., p_2_n2],
                MOD(t_1_2)),
      ...
      attribute("a_m",
                [p_m_1, p_m_2, ..., p_m_nm],
                MOD(t_1_m))])
```

where *MOD(_)* denotes a function transforming an XASM expression or rule into its constructor term representation. Correspondingly the representation of the previously defined attributes in ASM 42 is given in Term 12.

Term 12:
```
[attrDefs("U_1",
     [attribute("a_1", [], MOD(t_1_1)),
      attribute("a_2", [], MOD(t_1_2)),
      ...
      attribute("a_m", [], MOD(t_1_m))]),
 attrDefs("U_2",
     [attribute("a_1", [], MOD(t_2_1)),
      attribute("a_2", [], MOD(t_2_2)),
      ...
      attribute("a_m", [], MOD(t_2_m))]),
 attrDefs("U_n",
     [attribute("a_1", [], MOD(t_n_1)),
      attribute("a_2", [], MOD(t_n_2)),
      ...
      attribute("a_m", [], MOD(t_n_m))])
]
```

The above representation is generated by extending the EBNF of PXasm (Grammar 6, Section 5.4.1) with the following productions and constructor mappings.

Gram. 7: *UniverseDef* ::= *"universe" Symbol* { *AttrDef* }
=> attrDefs(Symbol, AttrDef)
AttrDef ::= *"attr" Symbol [Arguments] "==" Expr*
=> attribute(Symbol, Arguments, Expr)

The constructor representations of the XASM constructs dot and self expressions are given by the following definitions.

Gram. 8: *Expr* = ... | *Dot* | *Self*

	=> rhs
Dot	::= *Expr* *"."* *Expr*
	=> dot(Expr.1, Expr.2)
Self	::= *"self"*
	=> selfSymb

7.2.3.2 Extending the Self-Interpreter for Attributions without Parameters

The definition of the self interpreter of such an attribution system is relatively complex. We try to simplify understanding by first concentrating on non-parametric attributions, which are nearer to classical attribute grammars. Later in Section 7.2.3.3 we come back to attributions with parameters. This allows us to abstract in this section from the outermost context object, which is only used for evaluating parameters. The signature of the self interpreter given in this section is *A_INTERP(_, _)*, the first argument being a term-representation of the ASM rule to be executed, the second being the current context-object.

The following function *EvalAttribute(cc,a)* is used to evaluate an attribute a with respect to a context-object *cc*. The attribute definitions are available as a list of their constructor term representation which is assigned to variable *AttrDefs*. If the attribute is not defined for the given context-object, the constant[3] *notDeclared* is returned, otherwise the result of evaluating the attribute definition by means of A_INTERP is returned. The derived universe

```
derived function UniverseSet(u) ==
   (exists a in list AttrDefs: a=~ attrDefs(u, &))
```

denotes a set of all universes for which attributes are defined. The following ASM chooses a universe u in the set of universes *UniverseSet*, such that the argument *cc* is in u. Then it chooses u's attribute definitions *ATTR_DEFS* in the list *AttrDefs*, and in the list *ATTR_DEFS* is chosen the attribution *ATTR* corresponding to the ident a. The defining expression of *ATTR* is then interpreted with respect to context object *cc*. If one of the three choose operators does not succeed, the element *notDeclared* is returned.

ASM 43:
```
asm EvalAttribute(cc: Object, a: Ident)
  accesses function A_INTERP(_,_)
  accesses function AttrDefs
  accesses constructor notDeclared, meta(_)
  accesses universe UniverseSet
  (forall u in UniverseSet
     accesses universe $u$)
is
  choose u in UniverseSet: $u$(cc)
    choose ATTR_DEFS in list AttrDefs:
        ATTR_DEFS =~ attrDefs(u, &DefList)
      choose ATTR in list &DefList:
          ATTR =~ attribute(a, [], &e)
        A_INTERP(&e, cc))
      ifnone
        return notDeclared
```

[3] Constants are modeled as 0-ary constructors.

```
      endchoose
    ifnone
      return notDeclared
    endchoose
  ifnone
    return notDeclared
  endchoose
endasm
```

Interpretation of Symbols

For non-parametric attributions, the arguments of the AXasm self interpreter *A_INTERP* are an XASM rule t and the context object *cc*. The same arguments are needed for the interpretation of symbols, which is similar to the interpretation of symbols without attributes (ASM 24), except that the context object must be passed as well.

ASM 44:

```
asm SymbolA_INTERP(t, cc)
  accesses function A_INTERP(_,_)
is
  if t =~ meta(&s) then
    return A_INTERP(&s, cc)
  else
    return t
  endif
endasm
```

Interpretation of Rules: Structure

The interpreter for attributed XASM needs to update all functions mentioned in the term t, and accesses the mentioned functions *EvalAttribute*, *AttrDefs*, as well as the constructor *notDeclared*, and the universe *UniverseSet*. The signature of the ASM is almost identical to the corresponding ASM 25 of the PXasm self-interpreter.

ASM 45:

```
asm A_INTERP(t, cc)
asm INTERP(t: Rule | Expr)
 (forall n in {0 .. MaxArity(t)}:
  updates functions with arity n $UpdFct(n, t)$
  accesses functions with arity n $AccFct(n,t)$
 )
 accesses constructors update(Symbol, [Expr], Expr),
                       conditional(Expr, Rule, Rule),
                       doForall(Symbol, Symbol, Rule),
                       extendRule(Symbol, Symbol, Rule),
                       constant(Value),
                       apply(Symbol, [Expr]),
                       letClause([LetDef], Rule),
                       letDef(Symbol, Expr)
  accesses function AttrDefs
 accesses constructor notDeclared
  accesses universe UniverseSet
is
 external function EvalAttribute(obj,par)
 external function SymbolA_INTERP(_,_)
...
```

Interpretations which do not change considerably

There are a number of rules whose interpretation is not changing its main functionality with respect to the PXasm self-interpreter. This rules are update, lists, conditionals, doForall, extendRule, and constant. The only difference of the interpretation for these rules is, that the additional context object *cc* is passed as argument to each interpretation of their components.

Interpretation of Apply

The interpretation of *apply* has now to take into consideration the context object *cc*. The interpreter calls ASM *EvalAttribute* to see whether in the context of *cc* an attribute is defined, which matches the symbol to be interpreted. First, it is tested whether context object *cc* is *undef*, e.g. if we are in the global context. If yes, a global dynamic function is evaluated, otherwise an attribute is assumed, or if later no such attribute is defined, the *cc* is added as first argument to a global function evaluation.

Since we assume attributes to have no parameters, the parameters are simply skipped in the call of *EvalAttribute*. If there is no attribute found, the function *EvalAttribute* returns *notDeclared* and the function is evaluated as global function using the built in *Apply* operator.

```
...
elseif t =~ apply(&op, &a) then
  let opINT = SymbolA_INTERP(&op, cc),
      aINT = A_INTERP(&a, cc) in
    if cc = undef then
      return Apply(opINT, aINT)
    else
      let r = EvalAttribute(cc, opINT) in
        if r = notDeclared then
          return Apply(opINT, [cc | aINT])
        else
          return r
        endif
      endlet
    endif
  endlet
...
```

Interpretation of dot

The *dot* operator is used to change the context object *cc*. In the case of attributed XASM without parameters this is easily done by replacing the argument *cc* with the newly created object.

```
...
elseif t =~ dot(&t1, &t2) then
  let lhs = A_INTERP(&t1, cc) in
    return A_INTERP(&t2, lhs)
  endlet
...
```

Interpretation of self

The term *self* refers to the context object, thus to the object *cc*.

```
...
elseif t =˜ selfSymb then
  return cc
...
```

Interpretation of let clauses

Finally we have to treat the *let clauses*. For this purpose we calculate first the value of each let definition and build then recursively the let rules. Since we first evaluate all let definitions, the used recursive let construction is equivalent to the intended parallel one of the interpreted term.

```
...
elseif t =˜ letClause(&defList, &r) then
  if &defList =˜ [letDef(&p, &t)|&tl] then
    return A_INTERP(letClause(A_INTERP(&defList,
                                     cc),
                              &r),
                    cc)
  elseif &defList =˜ [(&p, &o) | &tl] then
    let $&p$ = &o in
      return A_INTERP(letClause(&tl, &r), cc)
    endlet
  else return A_INTERP(&r, cc)
  endif
elseif t =˜ letDef(&p, &t) then
  return (SymbolA_INTERP(&p, cc), A_INTERP(&t, cc))
...
```

7.2.3.3 Attributions with Parameters

Parametric attributions extend attributes with parameters. If such attributions are evaluated, the parameters are evaluated in the outermost context, and only the context of the attribute evaluation is changed by the ”dot”-notation. Therefore two context objects *(cc, oc)* must be passed as arguments of the evaluation function, one for the parameters, and one for the attribute. As a consequence, all of the above rules have to be extended with a second context-object parameter.

As an exception, the ASM *EvalAttribute* still only needs access to one context object, but we need to pass the already evaluated arguments to the attributes. Further, the arity of the accessed *A_INTERP* has changed with respect to the old definition ASM 43. The ASM *CreatePairs* is needed to transform the lists of actual and formal parameters into a list of pairs, consisting of formal name and actual value, which then can be interpreted as a list of let-clauses, thereby using the self-interpretation of let-clauses to give self-interpretation of attributes with parameters.

ASM 46:

```
asm EvalAttribute(cc: Object, a: Ident, actual: [Object])
  accesses function A_INTERP(_,_,_)
  accesses function AttrDefs
  accesses constructor notDeclared
  accesses universe UniverseSet
  (forall u in UniverseSet
     accesses universe $u$)
is
  choose u in UniverseSet: $u$(cc)
    choose ATTR_DEFS in list AttrDefs:
        ATTR_DEFS =~ attrDefs(u, &DefList)
      choose ATTR in list &DefList:
          ATTR =~ attribute(a, &formal, &e)
        A_INTERP(letClause(CreatePairs(&formal,
                                       actual),
                           &e),
                 cc, cc))
      ifnone
        return notDeclared
      endchoose
    ifnone
      return notDeclared
    endchoose
  ifnone
    return notDeclared
  endchoose
endasm

asm CreatePairs(l1, l2)
is
 if l1 =~ [&hd1 | &tl1] then
   if l2 =~ [&hd2 | &tl2] then
     return [(&hd1, &hd2) | CreatePairs(&tl1, &tl2)]
   endif
 else
   return []
 endif
endasm
```

With respect to the formulation without parameters, ASM 45, the ASM *A_INTERP* has a third argument, the outermost object, and the referred function *EvalAttribute* is now 3-ary.

ASM 47:

```
asm A_INTERP(t: Rule, cc: Object, obj_outermost: Object)
...
is
  external function EvalAttribute(_,_,_)
...
```

For almost all rules, the third parameter is simply passed to the interpretation of the components.

The only place where the outermost context is used is in the fragment dealing with applications. There the parameters are evaluated in the context of the outermost object, while the attribute is evaluated with respect to the context-object. If there is no attribute defined, the *Apply*-operator is used to evaluate the function. The *self* is set to the context object.

```
  ...
  elseif t =~ apply(&op, &a) then
```

```
    let opINT = SymbolA_INTERP(&op,
                               obj_outermost,
                               obj_outermost),
        aINT = A_INTERP(&a,
                        obj_outermost,
                        obj_outermost) in
      let r = EvalAttribute(cc, opINT, aINT) in
      if r != notDeclared then
        return r
      else
        let self = cc in
          return Apply(opINT, aINT)
        endlet
      endif
    endlet
  ...
```

The complete *A_INTERP* ASM looks as follows.

ASM 48:

```
asm SymbolA_INTERP(t, cc, obj_outermost)
is
  if t =~ meta(&s) then
    return A_INTERP(&s, cc, obj_outermost)
  else
    return t
  endif
endasm

asm A_INTERP(t, cc, obj_outermost)
 updates *
 accesses function EvalAttribute(obj,att,par)
  accesses function AttrDefs
 accesses constructor notDeclared
  accesses universe UniverseSet
is
  if     t =~ update(&s, &a, &e) then
    Update(SymbolA_INTERP(&s, cc, obj_outermost),
           A_INTERP(&a, cc, obj_outermost),
           A_INTERP(&e, cc, obj_outermost))
    return true
  elseif t =~ [&hd | &tl] then
    return [A_INTERP(&hd, cc, obj_outermost)
           | A_INTERP(&tl, cc, obj_outermost)]
  elseif t =~ conditional(&e, &r1, &r2) then
    if A_INTERP(&e, cc, obj_outermost) then
         A_INTERP(&r1, cc, obj_outermost)
    else A_INTERP(&r2, cc, obj_outermost) endif
    return true
  elseif t =~ doForall(&i, &s, &e, &r) then
    do forall $SymbolA_INTERP(&i, cc, obj_outermost)$
         in $SymbolA_INTERP(&s, cc, obj_outermost)$:
              A_INTERP(&e, cc, obj_outermost)
      A_INTERP(&r, cc, obj_outermost)
    endo
    return true
  elseif t =~ choose(&i, &s, &e, &r1, &r2) then
    choose $SymbolA_INTERP(&i, cc, obj_outermost)$
        in $SymbolA_INTERP(&s, cc, obj_outermost)$:
          A_INTERP(&e, cc, obj_outermost)
      A_INTERP(&r1, cc, obj_outermost)
    ifnone
      A_INTERP(&r2, cc, obj_outermost)
    endchoose
    return true
  elseif t =~ extendRule(&i, &s, &r) then
    extend $SymbolA_INTERP(&s, cc, obj_outermost)$
       with $SymbolA_INTERP(&i, cc, obj_outermost)$
      A_INTERP(&r, cc, obj_outermost)
    endextend
    return true
  elseif t =~ constant(&c) then
    return &c
  elseif t =~ apply(&op, &a) then
    let opINT = SymbolA_INTERP(&op,
                               obj_outermost,
                               obj_outermost),
        aINT = A_INTERP(&a, obj_outermost, obj_outermost) in
      let r = EvalAttribute(cc, opINT, aINT) in
      if r != notDeclared then
        return r
      else
        let self = cc in
          return Apply(opINT, aINT)
        endlet
      endif
    endlet
  elseif t =~ dot(&t1, &t2) then
    let lhs = A_INTERP(&t1, cc, obj_outermost) in
      return A_INTERP(&t2, lhs, obj_outermost)
    endlet
  elseif t =~ selfSymb then
    return cc
  elseif t =~ letClause(&defList, &r) then
    if &defList =~ [letDef(&p, &t)|&tl] then
      return A_INTERP(letClause(
                       A_INTERP(&defList, cc, obj_outermost), &r),
                       cc, obj_outermost)
    elseif &defList =~ [(&p, &o) | &tl] then
      let $&p$ = &o in
        return A_INTERP(letClause(&tl, &r), cc, obj_outermost)
      endlet
    else return A_INTERP(&r, cc, obj_outermost)
    endif
  elseif t =~ letDef(&p, &t) then
    return (SymbolA_INTERP(&p, cc, obj_outermost),
            A_INTERP(&t, cc, obj_outermost))
  else return "Not matched"
endif
endasm
```

7.3 Related Work and Results

We have discussed the relation of Montages with Attribute Grammar based formalisms for dynamic semantics in Section 3.5 and we concentrate now on the comparison of AXasm and traditional Attribute Grammars (AGs) for the specification of static semantics of programming languages.

The application of AGs for specifying static semantics of programming languages has produced a large number of approaches. A good survey of the obtained results can is given by Waite and Goos (221). The actual algorithms for the semantic analysis are simple but will fail on certain input program if the underlying AG is not well-defined. Testing if a grammar is well-defined, however, requires exponential time (103). A sufficient condition for being well-defined can be checked in polynomial time. This test defines the set of ordered AGs as being a subset of the well-defined grammars (117). However, there is no constructive method to design such grammars. These problems have led to a number of alternative approaches based on predicate calculus (212; 167; 183) which avoid these problems, but do not allow for the generation of efficient semantics analyzer which can be used in practical compilers. Since AXasm allow both the use of arbitrary complex AGs and predicate calculus, they are not solving the traditional problems of AG research. The only purpose of AXasm is to simplify the specification of static semantics and they are not providing any solution for the problem of generating efficient semantics analysis tools. With other words, AXasm are not an alternative for AGs since AXasm are only reusing the ease of specification features of AGs, but not preserving the efficiency features of AGs.

In contrast to AXasm, traditional Attribute Grammars make the connection to the grammar explicit and declare not only the signature of attributes, but as well their typing and the direction of the information flow.

- *Synthesized Attributes* Attributes whose value is calculated from attributes of their siblings are called *synthesized attributes*. Information for the calculation of these attributes flows thus from the leafs of the tree towards the root.

- *Inherited Attributes* Those attributes whose value is calculated from the value of their parent's attributes are called *inherited attributes*. Information for these calculations flows from the root towards the leaves of the tree.

In AXasm only synthesized attributes are defined traditionally, inherited attributes are simulated using a special attribute *Parent* which links nodes in the parse tree to their parent node. The attribute *Parent* has been introduced in Section 3.2.2 and formalized in Section 5.3.1. We see clear limitations of not having inherited attributes, but on the other hand this allows us to considerably simplify the syntax of attribute definitions and to have the definitions look and feel like method declarations in object oriented programming.

On the other hand the existence of the *Parent* attribute and the *enclosing* function (Section 5.3.2) together with the fact that values of AXasm attributes can be references to other nodes in the tree allows in certain situation for a much more compact specification style. Instead of locally moving information from parent to sibling, using inherited attributes, the information can be directly accessed by using the *enclosing* function. For instance name resolution, a feature typically specified with inherited attributes, is covered in AXasm by directly accessing the declaration table of the least enclosing scope. Interestingly, the same function enclosing is already used by Poetzsch-Heffter in the MAX system (184; 186). Both in MAX and in our system, the *enclosing* function allows to simplify the specification of such features by being able to point directly to the least enclosing instance of a certain feature, or the the least enclosing instance of a set of features.

In summary the main differences of AXasm with respect to attribute grammars are the following.

- *Arbitrary Structure* AXasm can be defined over a number of object sets, which are not building a parse tree. In fact, AXasm do not start with a grammar, but with an arbitrary *partition* of the involved objects, independent whether they are nodes of a parse tree or not. For simplicity we still use the notion *node* for those objects which have attributes

- *Untyped* The terms defining attributions of AXasm are not typed.

- *Global References* While in traditional attribute grammars the definition of an attribute only depends from the attributes of its siblings or its parent, in AXasm attributes can be calculated by referring to any other object. Both the MAX system (186) and Hedin's reference attribute grammars (89) provide a similar feature.

- *Reference Values* In traditional attribute grammars, the values of attributes are restricted to constants, such as strings and numbers, or mappings. In XASM, the value of one attribute can be another node of the AST. Again the MAX system and reference AG provide a similar feature.

- *Parameterized Attributes* In XASM, an attribute can have additional parameters. Like this, it is not necessary to return higher order data-structures like mappings.

By further generalizing the idea and extending it with a mechanism for inheritance, an OO version of XASM would be obtained, but the definition of a full OO version of XASM is beyond the scope of this thesis and we refer the reader to the executable specification of OO XASM (128).

8

Semantics of Montages

In this section we give a formal semantics of Montages using parameterized, attributed XASM as introduced in Chapters 5 and 7. For simplicity we refer to them as XASM. The presented algorithms are based on code which has been implemented and carefully tested with the Gem-Mex tool. The running code has ten been rewritten for the thesis using the novel XASM features introduced in the last chapters. Testing the final version of the algorithms has not been possible since the new features are not yet implemented.

In Section 8.1 we reevaluate the meta-interpreter semantics of Montages by discussing different alternatives for giving semantics for a meta-formalism. As mentioned at the beginning of Part II the advantage of the given formalization are that it is executable, serves directly as implementation of Montages, and is easy to maintain, since it is based on one, fixed XASM specification. Based on TFSMs, we have shown in Chapter 6 how the meta-interpretation specification allows to use partial evaluation to transform language descriptions into specialized interpreters and to compile programs of the described language into specialized XASM code. The resulting specialized code is in both cases not only more efficient, but as well easier to understand and validate.

In this Chapter we abstract from partial evaluation and other efficiency and code transparency related issues and give algorithms building non-optimized and non-simplified TFSMs from Montages. The techniques of Chapter 6 can then be applied to get a maintainable and efficient implementation of Montages from the here presented meta-interpreter. In Section 8.2 the Montages meta-interpreter is structured, and then the details of processing Montages aspects relating to static semantics (Section 8.3), and to dynamic semantics (Section 8.4) are given.

Finally in Section 8.5 we conclude that the given meta-interpreter can be used to *meta-bootstrap* the XASM language, given a Montages description of XASM, and point to ongoing work on bootstrapping the complete Montages system.

8.1 Different Kinds of Meta-Formalism Semantics

A complete specification of a language is given by defining its syntax, its static semantics, and its dynamic semantics. A meta-formalism like Montages is used to give such language definitions.

Typically a language definition is given by means of a mathematical mechanism which takes as input a program in the given syntax, checks static semantics, and simulates dynamic semantics. If the language to be defined is a meta-formalism, e.g. a formalism to define other formalisms, the situation is more complex. Of course, as well a meta-formalism is given by defining its syntax and semantics. But each “program” written with the meta-formalism defines another formalism. The “programs” written with a meta-formalism are thus called *language-definitions*, and we use the term *program* for the programs written in the formalism specified by a language-definition. The specification of a meta-formalism defines thus syntax and semantics of language-definitions, and defines syntax and semantics for each language defined.

There are two different choices to formulate the specification of a meta-formalism. Either one gives a mathematical mechanism which takes both, the program and the language-definition as input, or one gives a mathematical mechanism, which transforms a language-definition into a mathematical mechanism being a definition of the described language.

The first choice, which takes as input both the program and the semantics definition is called *meta interpretation*. In our context, a meta-interpreter is an ASM which reads Montages descriptions of a language L plus an L-program P, and interpretes P according to the L-semantics. In Figure 37 we show a meta-interpreter, its input, and how it can be specialized to interpreters and compiled code for the specified language.

Alternatively, one can define a program generator, taking Montages descriptions of L as input and generating a specialized XASM model. This choice, which corresponds to the current architecture of the Gem-Mex tool, is visualized in Figure 36. The advantage of this approach is the simplicity of the resulting XASM model. The signature and structure of the model can be specialized for the given Montages. A simple language described by a few simple Montages results in a simple, specialized XASM model of the language. The disadvantage of this approach is that it is not trivial to formalize the generator. Further our experience with implementing this approach showed that the software generator can be a considerable maintenance problem. Because of this maintenance problem, and because we can achieve the advantages of the generator approach with partial evaluation of meta-interpreters, we decided to follow the meta-interpreter approach.

Following the meta-interpretation approach, we have the problem, that the signature of the terms used in Montages is specialized to the EBNF of the described language. One possibility to solve this problem is to transform the Montages descriptions into descriptions using a more generic signature, and give a meta-interpreter processing such generic Montages definitions. Like this we have the possibility to give a single, fixed ASM as semantics of Montages. The disadvantage of this solution is that the existing Montages modules must be transformed in a complex and context-dependent way. Another disadvantage is, that the complex, generic signature has to be understood even for simple Montages examples.

The author has experimented with this solution, described it for an XML based meta-formalism (126), and subsequently implemented it for Montages with the Gem-Mex tool. Although this results in a very small, highly abstract model, the outcome tends to be hard to understand. The reasons are that the complexity of the model is independent from the language described, and the terminology of the described language is not used for its description. The semantics given by such an abstract model can thus not be easily understood by the domain-experts.

Instead of transforming the Montages, we propose thus to use parameterized XASM to ”program” the specialized signature of the Montages. In the introduction to Part II we have already shown a simple example for this process. A meta-interpreter using this approach is as complex as one over a fixed signature. But using partial evaluation, the given parameterized meta-interpreter can be specialized into an interpreter or even a compiled program, using a signature corresponding to the terminology introduced by the EBNF rules of the described language. A meta-interpreter approach using parameterized XASM allows thus to take advantage of end-user terminology, and fits perfectly a framework for domain-specific languages. The resulting specialized XASM descriptions correspond both in signature and structure to the given Montages.

In the following sections one fixed parameterized ASM *MontagesSemantics* is given as semantics of the Montages meta-formalism. Given a language description, the signature-parameters of *MontagesSemantics* can be instantiated and the parameterized ASM is easily reduced to a simple specialized ASM, whose size and complexity is directly related to the complexity of the described language.

8.2 Structure of the Montages Semantics

To define the semantics of Montages, we give the meta-interpreter ASM *MontagesSemantics* which receives as parameters

mtg the list of Montages, and

prg the program to be analyzed and executed.

MontagesSemantics generates from these parameters an AST, collects the attribution rules from the Montages, checks the static semantics condition for each node, decorate the AST with states and transitions, and finally execute the resulting TFSM.

8.2.1 Informal Typing

Until now we have given no typing information, since XASM has no static type system. To make the descriptions of constructor-term representations more readable, we use an informal notation for typing. The following declaration

```
constructor c3(T1, T2, T3) -> T4
```

denotes that constructor $c3$ takes arguments of type $T1$, $T2$, $T3$ and produces constructor terms of type $T4$. As a convention we assume that constructor symbols are given with lower case letters, and that types start with a capital letter. The notion [T] denotes a list-type of T-instances, {T} denotes a corresponding set-type. The synonym notation known from the EBNF rules can be used to denote union types. For instance, the rule

Gram. 9: *Expr* = *Unary* | *Binary* | *CondExpr*
| *Application* | *Constant* | *Let*

from the grammar of XASM rules induces an informal typing definition of union type Expr built by the types on the right-hand-side. In general we will treat upper-case EBNF symbols from the XASM and attribution grammars as types of the corresponding constructor-terms.

8.2.2 Data Structure

Both *mtg* and *prg* are passed to *MontagesSemantics* as constructor terms. The program *prg* to be executed is passed as a constructor term built up by the constructor *characteristic*, representing applications of characteristic production rules, and the constructor *synonym*, representing applications of synonym productions. Section 4.5.3 gives the details of this canonical representation.

The elements of a Montage represented as constructor are its name, being an *Ident*, a list of *Attributes*, an XASM expression being the static semantics condition, a list of *States*, and a list of MVL *transitions*.

```
constructor montage(Symbol,
                    [Attributes],
                    Expr,
                    [State],
                    [Transition])
```

Examples of Montages containing all these parts are the A-Montage in Figure 9 and the While Montage in Figure 10. The transitions of these Montages have already been given as Term 1 and 2 in Section 3.3.2. The representation of the A-Montage as constructor term, modulo the free variables $T, C, C1, C2, C3$, and R looks as follows.

Term 13:

```
montage("A",
        [... , attribute("a", ["p1", ..., "pn"], T), ...],
        C,
        [ ..., state("s3", R), ...],
        [transition(siblingPath("B", undef, statePath("s1")),
                    C1,
                    siblingPath("B", undef, statePath("s2"))),
         transition(siblingPath("B", undef, statePath("T")),
                    C2,
                    statePath("s3")),
         transition(statePath("s3"),
                    C3,
                    siblingPath("B", undef, statePath("I")))]
)
```

The corresponding constructor term for the while is:

Term 14:

```
montage("While",
    [attribute("staticType", [],
               dot(apply("S-Expr",[]),
                   apply("staticType",[])))],
    apply["=", [apply("staticType",[]),
                apply("BooleanType,[])]],
    [state("profile",update("LoopCounter",
                            [],
                            apply("+",
                                  [apply("LoopCounter",[]),
                                   constant(1)]))))],
    [transition((statePath("I"),
                 default,
                 siblingPath("Expr", undef", statePath("I")))
     transition(siblingPath("Expr", undef, statePath("T")),
                src.value,
                statePath("profile")),
     transition(siblingPath("Expr", undef, statePath("T")),
                default,
                statePath("T")),
     transition(statePath("profile"),
                default,
                siblingPath("Stm", undef, statePath("LIST"))),
     transition(siblingPath("Stm", undef, statePath("LIST")),
                default,
                siblingPath("Expr", undef", statePath("I")))
    ]
```

8.2.3 Algorithm Structure

The ASM *MontagesSemantics* processes program and semantics in different phases. Starting with the *construction* of the parse tree, the next step is *collection* of attribution rules, and then follows the *check* of static semantics conditions. After this phase, a program is said to be *valid*. If the program is not valid, the string ”Program is not valid” is return and the process is stopped.

Parse trees of valid programs are then *decorated* with control-flow information and then *executed*. The current phase of this process is given by a dynamic function *mode* which changes its value from *construct* to *collect*, *validate*, then if the program is valid to *decorate* and finally to *execute*. In Figure 8 of Section 3 these phases have already been mentioned. Phase 1 of that figure is concerned with initialization and construction of the AST. Phase 2 relates to collection and phase 3 to validation. Finally the phase 4 of the referenced figure relates to decoration, and phase 5 to execution. The overall structure of the ASM *MontagesSemantics* looks as follows.

ASM 49:

```
asm MontagesSemantics(prg, mtg)
  ...
is
  constructors construct, collect, validate,
               notValid, decorate, execute
  function mode <- construct
if     mode = init then
  ... construct tree ...
  mode := collect
elseif mode = collect then
  ... collect attributions ...
  mode := validate
elseif mode = validate then
  if ... check static semantics ... = true then
       mode := decorate
  else return "Program is not valid."
  endif
elseif mode = decorate then
  ... decorate tree ...
  mode := execute
elseif mode = execute then
  ... execute ...
endif
endasm
```

In Section 8.3 we give all details of the Montages semantics concerned with static semantics of described programming languages and in Section 8.4 the formalization of the dynamic semantics aspects are given.

8.3 XASM definitions of Static Semantics

After the construction of the AST, which is described in Section 8.3.1, the attributions of each Montages are collected and assembled to an attributed XASM. This collection phase is described in Section 8.3.2. As the last phase of static semantics processing, the static semantics conditions are checked for all nodes of the abstract syntax tree. This process is described in Section 8.3.3.

8.3.1 The Construction Phase

In the construction phase, the abstract syntax tree is constructed from the given term representation of the program. The definition of ASM *ConstructCanonicTree* has been given as ASM 17 in Section 5.3. The universes and selector functions updated by *ConstructCanonicTree* are declared here, such that they are available by in later phases. Further a dynamic function *root* is declared, and the root of the constructed AST is assigned to it. The corresponding fragment of *MontagesSemantics* is given as follows, refining ASM 49.

ASM 50:
```
asm MontagesSemantics(prg, mtg)
  accesses constructors synonym(_,_), characteristic(_,_), ...
  accesses universess CharacteristicSymbols, SynonymSymbols, ...
is
    external function ConstructCanonicTree(Term), ...
    universe NoNode, ListNode
    (for all c in CharacteristicSymbols:
      universe $c$
      function $"S-"+c$(_)
  )
  (for all s in SynonymSymbols:
      universe $s$
      function $"S-"+s$(_)
  )
  function mode <- construct
  function root, ...
  constructors construct, collect ...
  ...
if     mode = construct then
  root := ConstructCanonicTree(prg)
  mode := constructed
...
endasm
```

8.3.2 The Attributions and their Collection

The list of attributes is a list of attribute constructors, as introduced in Section 7.2.3.1. The typing of the attribute constructor is

```
constructor attribute(Ident, [Ident], Expr) -> Attribute
```

where the first *Ident* is the name of the attribute, the list of *Idents* denotes the arguments of the attribute and the expression *Expr* is an XASM expression whose evaluation determines the value of the attribute.

The list of attributions is collected by the following ASM. The parameter *mtgList* is the list of montage-terms representing a language specification. The algorithm extracts from each montage-constructor the first and the second argument, and builds up a corresponding list of attributions, using the attrDefs-constructor.

ASM 51:
```
asm CollectAttributions(mtgList)
  accesses constructors montage(_,_,_,_,_),
                        attrDefs(_,_)
is
  function a <- []
  if mtgList =~ [montage(&Symbol, &Attrs, &, &, &) | &tl] then
     a := a + [attrDefs(&Symbol, &Attrs)]
     mtgList := &tl
  else return a
  endif
endasm
```

In the collect phase of the Montages semantics, the attributions are collected and assigned to function *AttrDefs*. The details of *MontagesSemantics* with respect to the collect phase are given as the following refinement of ASM 50

ASM 52:
```
asm MontagesSemantics(prg, mtg)
  ...
is
  ...
  external function CollectAttributions(Mtgs)
  function AttrDefs
  ...
  constructors  ..., collect, validate, ...

...
elseif mode = collect then
  AttrDefs := CollectAttributions(mtg)
  mode := validate
...
endasm
```

8.3.3 The Static Semantics Condition

The third element of a Montage is the *static semantics condition*. It is a normal XASM expression, which will be checked in the context of each instance of the Montage. For the evaluation of the conditions, the ASM 48, *A_INTERP* from Section 7.2.3 is used.

The derived function *getMontage(Ident)*, returns the Montage constructor having the name given with the argument, and the derived function *getCondition(Montage)* returns the static semantics condition from a Montage constructor.

```
derived function getMontage(id) ==
  (choose m in list mtg: m =~ montage(id, &,&,&,&))
derived function getCondition(m) ==
  (if m =~ montage(&, &, &Cond, &,&) then &Cond else undef)
```

The following ASM *CheckSemantics* evaluates for all instances n of a characteristic symbol s the corresponding static semantics condition

$$s.\mathit{getMontage}.\mathit{getCondition}$$

The ASM accesses the AXasm interpreter *A_INTERP*, the functions *getMontages* and *getCondition*, as well as the universe of characteristic functions. The body calculates the conjunction of all static semantics conditions of all nodes in the AST. The nodes are enumerated by ranging over all node-universes, given by the characteristic functions.

ASM 53:
```
asm CheckSemantics
 accesses function A_INTERP(Term, Obj, Obj)
 accesses functions getMontage(Symbol)
 accesses function getCondition(Montage)
 accesses universe CharacteristicSymbols
is
  return
   (forall s in CharacteristicSymbols:
      (let mtg0 = s.getMontage in
         (let cond0 = mtg0.getCodition in
            (forall n in $c$:
                A_INTERP(cond0, n, n)))))
endasm
```

The corresponding fragment of *MontagesSemantics* is given below. Together with ASM 49 and refinement ASMs 50 and 52 it covers the static aspects of a Montages specification.

ASM 54:
```
asm MontagesSemantics(prg, mtg)
  ...
is
  ...

  derived function getMontage(Symbol) ==
    (choose m in list mtg: m =~ montage(Symbol, &,&,&,&))
  derived function getCondition(Mtg) ==
    (if Mtg =~ montage(&, &Cond, &,&) then &Cond else undef)
  external function CheckSemantics
  external function A_INTERP(Term, Obj, Obj)
  ...
  constructors ..., validate, decorate, ...

...
elseif mode = validate then
  if CheckSemantics then
    mode := decorate
  else
    return "Program is not valid."
  endif
...
endasm
```

8.4 XASM definitions of Dynamic Semantics

Once the static semantics conditions are checked, the lists of states and transitions are used to build a tree finite state machine as described in Section 3.3.

- In Section 8.4.1 the association of states and actions is pre-calculated,
- in Section 8.4.2 the form of transitions is recapitulated,
- in Section 8.4.3 the instantiation of explicit transitions, and
- in Section 8.4.4 the creation of implicit transition are described.

In Section 8.4.6 the semantics of execution is given. Most material in this section is a refinement of algorithms introduced in Sections 3.3 and 6.1.

8.4.1 The States

A state has two elements, its name, being an identifier, and an XASM rule, its *action*.

```
  constructor state(Ident, Rule)
```

The structure of rules has been given in Section 5.4 grammar Grammar 6. A dynamic function

```
function getAction(Node, State) -> Action
```

is defined such, that for each node n, and state t the term *n.getAction(s)* returns the corresponding action-rule. The same function has been used in the TFSM interpreter of Section 6.1. Now we can give an ASM *DecorateWithStates* which defines function *getAction* for all nodes.

The derived function *getStates* extracts the state component from the *montage*-constructor

```
   derived function getStates(Mtg) ==
    (if Mtg =~ montage(&, &, &, &States,&) then &States else undef)
```

and the derived function *getMontages* returns the right Montage constructor.

ASM 55:
```
asm DecorateWithStates
  updates function getAction(_,_)
  accesses constructor state(_,_)
  accesses function getMontage(_), getStates(_)
  accessse universe CharacteristicSymbols
is
  do forall c in CharacteristicSymbols:
    let mtg0 = c.getMontage in
      let states0 = mtg0.getStates in
        do forall n in $c$:
          do forall s in list states0:
              s =~ state(&name, &action)
            n.getAction(&name) := &action
          enddo
        enddo
    endlet
  enddo
endasm
```

8.4.2 The Transitions

A MVL transition consists of three parts, the source of the transition, its firing condition, and the target of the transition. The ASM *InstantiateTransitions* instantiates MVL-transitions with TFSM transitions. In Section 3.3.2 we have defined basic paths, in Section 3.4.1 we introduced paths from and to lists, and an Section 3.4.3 paths to non-local target have been explained. Throughout these sections the algorithm *InstantiateTransitions* has been explained, and finally in Section 3.4.4 the complete definition was given. Later in Section 6.1 we have formalized the TFSM transitions as five-tuples in XASM which are added to a universe *Transitions*.

The differences between the informal version in Section 3.4.4 and the XASM counterpart are relatively small. We use dynamic functions instead of variables, and the outer explicit loop can be skipped, since XASM loop implicitly. For the application of the selector functions, the AXasm interpreter *A_INTERP* is used, and TFSM transitions are created by adding them to the relation *Transition*.

The previous XASM definitions are refined such that for each instantiated transition the node triggering its creation is remembered. In the condition of the transition, this *create-node* or *context-node* can be accessed as *self* in the the condition. The reason for this refinement is, that like this, all terms in a Montages, both the action rules and the conditions on transitions refer to the same self-object, if they are evaluated. Like this, a higher level of decoupling among different Montages is achieved.

As a consequence, in the refined version TFSM transitions are six-tuples, rather than five-tuples. Adding a transition

- from node/state pair (sn/ss),
- being created in under condition c,
- targeting to (tn/ts), and
- being created by node cn

is done by the following update.

```
  Transitions((sn, ss, cn, c, tn, ts)) := true
```

8.4.3 The Transition Instantiation

Algorithm *InstantiateTransitions* instantiates each MVL-transition with a number of TFSM-transitions.

In the so called *decoration phase* of *MontagesSemantics* the MVL-transitions of each Montage are instantiated for all instances of that Montage. The ASM *DecorateWithTransitions* instantiates for all nodes n all transitions t being part of its Montage *mtg0*. Formally, this is done by a number of let and do-forall constructs as follows.

```
do forall c in CharacteristicSymbols:
  let mtg0 = c.getMontage in
```

```
    let trans0 = mtg0.getTransisions in
      do forall n in $c$:
        do forall t in list trans0:
          ...
```

The actual instantiation of the MVL-transition t is done by an external function *InstantiateTransitions*. The arguments passed are twice the start-node n, and the source and target paths.

```
          ...
          if t =~ transition(&sp, &c, &tp) then
            let cond = &c in
              InstantiateTransition(n, &sp, n, &tp)
            endlet
          endif
        enddo
      enddo
    endlet
  endlet
enddo
```

The ASM *InstantiateTransitions* processes the start nodes and paths and creates the corresponding TFSM transitions. The derived function *getTransitions* is defined as follows:

```
  derived function getTransitions(Mtg) ==
    (if Mtg =~ montage(&, &, &, &Trans) then &Trans else undef)
```

The complete ASM *DecorateWithTransitions* is given as follows.

ASM 56:

```
asm DecorateWithTransitions
  updates universe Transitions
  accesses functions CharacteristicSymbols,
                     getMontage(_),
                     getTransitions(_)
  accesses constructors transition(_,_,_),
                        siblingPath(_,_,_),
                        globalPath(_,_),
                        statePath(_)
is
external function InstantiateTransitions(_,_,_,_)
do forall c in CharacteristicSymbols:
   let mtg0 = c.getMontage in
      let trans0 = mtg0.getTransisions in
            do forall n in $c$:
               do forall t in list trans0:
                   if t =~ transition(&sp, &c, &tp) then
                     let cond = &c in
                       InstantiateTransition(n, &sp, n, &tp)
                     endlet
                   endif
               enddo
            enddo
      endlet
   endlet
enddo
endasm
```

The ASM *InstantiateTransitions* has four arguments, updates universe *Transition*, and accesses the functions *cond* and *n* from ASM *DecorateWithTransitions*.

ASM 57:

```
asm InstantiateTransition(srcNode, srcPath, trgNode, trgPath)
  accesses function n, cond
  updates universe Transitions
  accesses constructors transition(_,_,_),
                        siblingPath(_,_,_),
                        globalPath(_,_),
                        statePath(_)
is
  ...
endasm
```

Sibling Paths

The cases where source or target paths are *sibling paths* have been discussed already in Section 3.3.3. If the source path *srcPath* (respectively target path *trgPath*) is a sibling path, the corresponding sibling of the source node *srcNode* (respectively target node *trgNode*) is calculated and assigned to *srcNode* (respectively *trgNode*) and the remaining path-component is assigned to *srcPath* (respectively *trgPath*). In contrast to the informal version given in Section 3.3.3, we use the $-feature to construct the syntax of the selector function. As in earlier sections, we abstract from *S1*- and *S2*- type selector functions.

```
...
elseif srcPath =~ siblingPath(&name, 1, &path) then
  srcNode := srcNode.$"S-"+&name"$
  srcPath := &path
elseif trgPath =~ siblingPath(&name, 1, &path) then
  trgNode := trgNode.$"S-"+&name"$
  trgPath := &path
...
```

With this rules, each time if either the source or target path is a sibling path, the corresponding sibling is calculated and the path simplified.

Global Paths

The processing of global paths has also been discussed before in Section 3.4.3. If *srcPath* (respectively *trgPath*) is a global path, *InstantiateTransitions* is called recursively for each instance of the universe denoted by the global path. Again the $-feature is used for the new formulation.

```
...
elseif srcPath =~ globalPath(&name, &path) then
  do forall n0 in $&name$
    InstantiateTransition(n0, &path, trgNode, trgPath)
  enddo
elseif trgPath =~ globalPath(&name, &path) then
  do forall n0 in $&name$
    InstantiateTransition(srcNode, srcPath, n0, &path)
  enddo
...
```

List Processing
Processing of lists has already been discussed in Section 3.4.1. If due to the processing of a sibling or global path either the source or target node is a list, *InstantiateTransitions* is recursively called for each element of the list.

```
...
elseif srcNode =~ [&hd | &tl] then
  ...
    do forall n0 in list srcNode
      InstantiateTransition(n0, srcPath, trgNode, trgPath)
    enddo
  ...
elseif trgNode =~ [&hd | &tl] then
  ...
    do forall n0 in list srcNode
      InstantiateTransition(n0, srcPath, trgNode, trgPath)
    enddo
  ...
```

There are two exceptions to these processing rules, reflecting transitions starting and ending at special ”LIST” boxes representing the whole list rather than its instances. For a detailed discussion we refer again to Section 3.4.1.

The first exception, concerning transitions departing from such boxes is as follows. If the source node *srcNode* is a list and at the same time, the source path *srcPath* is equal to a special path *statePath(”LIST”)* then the source of the transition is the ”T”-state of the last element in the list. The second exception covers transitions ending at the List-box. If the target node *trgNode* is a list and at the same time, the target path *trgPath* is equal to a special path *statePath(”LIST”)* then the target of the transition is the ”I”-state of the first element in the list. Those exceptions are reflected by the following refinement of the above rule fragment processing source and target node lists.

```
...
if     srcNode =~ [&hd | &tl] then
  if srcPath = statePath("LIST") then
     if &tl = [] then
       InstantiateTransition(&hd,     statePath("T"),
                             trgNode, trgPath)
     else
       InstantiateTransition(&tl,     statePath("LIST"),
                             trgNode, trgPath)
     endif
  else
    do forall n0 in list srcNode
       InstantiateTransition(n0,      srcPath,
                             trgNode, trgPath)
    enddo
  endif
elseif trgNode =~ [&hd | &tl] then
  if trgPath = statePath("LIST") then
       InstantiateTransition(srcNode, srcPath,
                             &hd,     statePath("I"))
  else
    do forall n0 in list srcNode
```

```
        InstantiateTransition(n0,      srcPath,
                                trgNode, trgPath)
    enddo
  endif
...
```

In order to guarantee a correct processing, the list rules must be applied first, before the sibling and global rules. In none of the described rules matches anymore, we have the guarantee, that both the source and target node are normal nodes of the syntax tree, and that both the source and target path are *state* paths. The components of the state paths are dispatched, and the corresponding entry into the transition relation is created. The node n and the condition *cond* are functions accessed from the *DecorateWithTransitions* ASM.

```
...
elseif srcPath =~ statePath(&srcState) then
  if trgPath =~ statePath(&trgState) then
    Transition((srcNode,
                &srcState,
                n,
                cond,
                trgNode,
                &trgState)) := true
endif
```

The above explained single rules for *InstantiateTransitions* give together the following complete XASM definition, corresponding to the informal algorithm in Section 3.4.4. It is interesting to see that the formal version is neither larger nor more complex than the informal one.

ASM 58:

```
asm InstantiateTransition(   srcNode, srcPath, trgNode, trgPath)
  accesses function          n, cond
  updates universe           Transitions
  accesses constructors      transition(_,_,_), siblingPath(_,_,_),
                             globalPath(_,_),   statePath(_)
is
if     srcNode =~ [&hd | &tl] then
  if srcPath = statePath("LIST") then
     if &tl = [] then
       InstantiateTransition(&hd,     statePath("T"),
                             trgNode, trgPath)
     else
       InstantiateTransition(&tl,     statePath("LIST"),
                             trgNode, trgPath)
     endif
  else
    do forall n0 in list srcNode
       InstantiateTransition(n0,      srcPath,
                             trgNode, trgPath)
    enddo
  endif
elseif trgNode =~ [&hd | &tl] then
  if trgPath = statePath("LIST") then
       InstantiateTransition(srcNode, srcPath,
                             &hd,     statePath("I"))
  else
    do forall n0 in list srcNode
       InstantiateTransition(n0,      srcPath,
                             trgNode, trgPath)
    enddo
  endif
elseif srcPath =~ siblingPath(&name, 1, &path) then
  srcNode := srcNode.$"S-"+&name"$
  srcPath := &path
elseif trgPath =~ siblingPath(&name, 1, &path) then
  trgNode := trgNode.$"S-"+&name"$
  trgPath := &path
elseif srcPath =~ globalPath(&name, &path) then
  do forall n0 in $&name$
    InstantiateTransition(n0, &path, trgNode, trgPath)
  enddo
elseif trgPath =~ globalPath(&name, &path) then
  do forall n0 in $&name$
    InstantiateTransition(srcNode, srcPath, n0, &path)
  enddo
elseif srcPath =~ statePath(&srcState) then
  if trgPath =~ statePath(&trgState) then
    Transitions((srcNode, &srcState,
                 n,
                 cond,
                 trgNode, &trgState)) := true
```

8.4.4 Implicit Transitions

In addition to the explicit transitions, there are implicit default transitions, linking list elements sequentially, and connecting ”I” and ”T” state of each *NoNode*-instance. The implicit transitions have already been discussed at the end of Section 3.4. The ASM *DecorateWithImplicitTransitions* generates these implicit transitions.

ASM 59:

```
asm DecorateWithImplicitTransitions
  accesses universe ListNode, NoNode
  updates universe Transitions
is
  external function InstantiateListTransitions(_)
  do forall l in ListNode:
    InstantiateListTransitions(l)
  enddo
  do forall n in NoNode:
    Transitions((n, "I", n,
                         default,
                         n,
                         "T")) := true
  enddo
endasm
```

The ASM *InstantiateListTransitions* creates the implicit transitions for lists.

ASM 60:

```
asm InstantiateListTransitions(l)
  updates universe Transitions
is
  if l =~ [&hd0 | [&hd1 | &tl]} then
    Transitions((&hd0, "T", &hd0,
                            default,
                            &hd1,
                            "I")) := true
    InstantiateListTransitions([&hd1 | &tl])
  endif
  return true
endif
```

8.4.5 The Decoration Phase

The XASM MontagesSemantics has been given till the state when the static semantics condition is checked. The next step is to decorate the parse tree with the states and transitions resulting in a TFSM. The following fragment of *MontagesSemantics* refines ASM 54.

ASM 61:

```
asm MontagesSemantics(prg, mtg)
  ...
is
  ...
  universe Transitions
  external functions DecorateWithStates,
                     DecorateWithTransitions,
                     DecorateWithImplicitTransitions
  ...
  constructors ..., decorate, execute, ...
...
elseif mode = decorate then
  DecorateWithStates
  DecorateWithTransitions
  DecorateWithImplicitTransitions
  mode := execute
...
endasm
```

8.4.6 Execution

The execution of the program is done in the execution phase. The following definition of ASM *Execute(Node, State)* refines the earlier nondeterministic version of *Execute*, ASM 32 of Section 6.1[1].

The state of the execution is hold by two functions, *CNode* denoting the current node of the syntax tree, where control of the execution is, and *CState*, the current state of this node being visited.

The firing of the current action is done by the following rule.

```
  A_INTERP(CNode.getAction(CState), CNode, CNode)
```

and the condition of a transition t is evaluated by providing to the self interpreter not only the values for *src* and *trg* but as well by feeding the create-object as context of the evaluation.

```
  t =~ (CNode, CState, &cn, &c, &tn, &ts) andthen        (let src = CNode in
          (let trg = &tn in
              A_INTERP(&c, &cn, &cn)))
```

Further we declare the self-interpreter *A_INTERP* as an access function, rather than an external function. With this choice, the characteristic/synonym symbols, universes and selector functions must not be included in the interface of *Execute*.

[1] Other earlier sections of this thesis relating to the *Execute* algorithm are Section 3.3.1, introducing the algorithm with an example and Section 6.4, showing how to apply partial evaluation to this algorithm.

ASM 62:

```
asm Execute(n,s)
  accesses functions getAction(_, _),
                     A_INTERP(_,_,_)
  accesses universe Transitions
is
  relation  fired
  functions CNode <- n, CState <- s
if not fired then
    A_INTERP(CNode.getAction(CState), CNode, CNode)
else
  choose t in Transitions:
           t =~ (CNode, CState, &cn, &c, &tn, &ts)
       and (let src = CNode in
            (let trg = &tn in
               A_INTERP(&c, &cn, &cn)))
    CNode  := &tn
    CState := &ts
  ifnone
    choose t in Transitions:
        t =~ (CNode, CState, &, default, &tn’, &ts’)
      CNode  := &tn’
      CState := &ts’
    endchoose
  endchoose
endif
endasm
```

As a last refinement of the ASM *MontagesSemantics* we can now give the fragment refining ASM 61 with the fragment for execution.

ASM 63:

```
asm MontagesSemantics(prg, mtg)
  ...
is
  ...
  external function Execute(_,_)
...
elseif mode = execute then
  Execute(root, "I")
endif
endasm
```

We have now given the complete definition of the semantics of Montages. In total the definition has a size of about 377 lines of XASM code, counting every line in the way we presented the algorithms, including lines with “end” constructs, and lines with closing brackets. An efficient implementation needs in addition the algorithms for partial evaluation and simplification of TFSM, which are about 268 lines of code, following the same conventions.

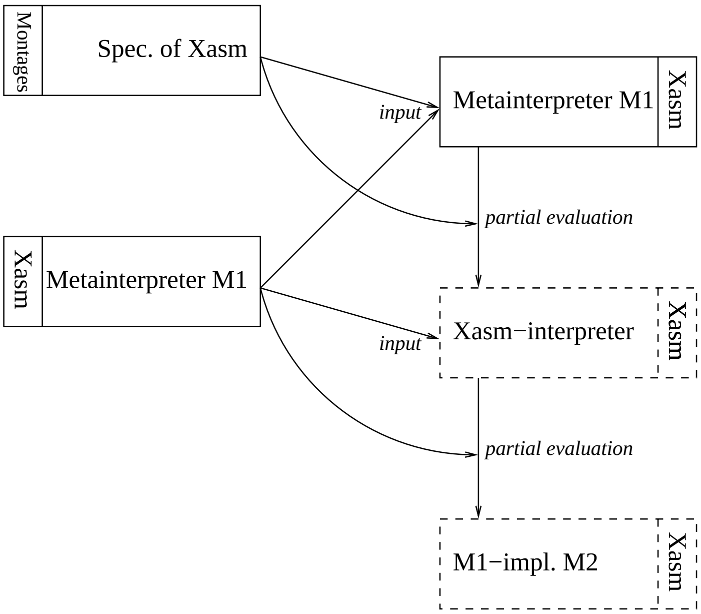


Fig. 42: Meta Bootstrapping of Montages System

8.5 Conclusions and Related Work

The given Montages meta-interpreter together with an XASM-semantics allows to *meta bootstrap* both the existing XASM language definition and meta-interpreter, as well as future versions of the XASM definition and meta-interpreter. Since the meta-interpreter corresponds to the definition of Montages, we are therefore able to meta-bootstrap future versions of both Montages and XASM with the presented process.

In Figure 42 we show what we understand under meta-bootstrapping (129), by applying the architecture presented in Figure 37 to the semantics of XASM and the Montages meta-interpreter. The input to the system are a Montages-specification of XASM, and the meta-interpreter $M1$. Please note that the same XASM-program $M1$ is both used as meta-interpreter, and as program serving as input to the partial-evaluation process. $M1$ is first specialized to an XASM-interpreter, and then to an implementation of $M1$, which we call $M2$. Meta-boots-trapping is done by tuning the specification, the partial evaluator, and the meta-interpreter such, that $M1$ equals $M2$ modulo pretty-printing. Like normal bootstrapping, this procedure cannot guarantee correctness, but allows to make the system more robust.

In Figure 35 the meta-bootstrapping has been visualized from a different perspective. The two cycles on the right are again shown in Figure 43, adapting them to the terminology of Figure 42. The meta-interpreter, being Montage's implementation and semantics, is developed in the left cycle on development platform XASM. In the right cycle, Montages is used as development platform to further develop the specification of XASM. If a new XASM-specification is released from the right cycle, the process of Figure 42 is used to bootstrap the existing meta-interpreter to the new version of XASM.

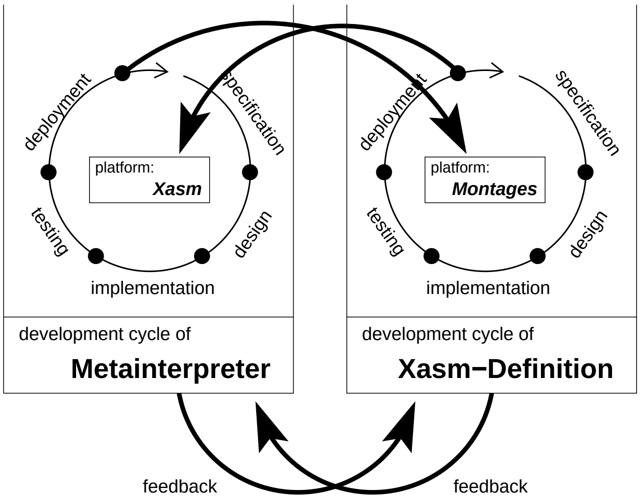


Fig. 43: The bootstrapping of XASM and Montages

Open problems and current areas of investigation are how to map object oriented XASM effectively into main-stream languages like C++ and Java, and how to port not only the interpreter/compiler from the old to the new architecture, but as well the graphical debugger and animation tool, which is currently generated for each described language (10).

Part III

Programming Language Concepts

In this part we use Montages to specify programming language concepts. We try to isolate each concept in a minimal example language. Each of these languages is tested carefully using Gem-Mex, and we invite the reader to use the prepared examples and the tool to get familiar with the methodology. The standard Gem-Mex distribution contains the sources.

The material is structured along two dimensions. The first is the already mentioned dimension of programming language concepts. We start with simple *expressions*, and then cover control statements like *if* and *while*, introduce the notions of *variables* and *updates*. Finally we show more advanced programming constructs like *procedure calls*, *exceptions*, and *classes*.

The second dimension is the dimension of applied specification patterns. Besides the Montages built in pattern of tree finite state machines, we use four identifiable patterns:

- **Declarator-Reification** A pattern common to most presented example language is to reuse tree-nodes being declarations as objects representing the type, variable, class, field, or method they are declaring. Attribution of the nodes is used to specify further properties, and dynamic fields are used to store the current value or state, e.g. the value of a variable or field, or the state of a class being initialized. Advantages of this pattern are *compactness* of the resulting model, since the existing nodes are reused, *ease of animation* of the specification, since the nodes correspond to areas in the program text which can be high-lighted, and *ease of specification* for features like scoping, overriding, and reloading of classes and modules, since different declaration-nodes with the same name can coexist. We call this pattern *Declarator-Reification* since the parse tree nodes being only declarations of objects like variables, procedures, classes, or modules are reificated into the very same objects.

- **Tree-Structural-Approach** A second major pattern is the use of the tree structure, by means of the universes, selector functions, the parent function, and the ASM *enclosed*, which have been defined in Section 5.3. As discussed in Section 5.3.3 the tree structure is used for both static *scope resolution* and *dynamic binding* to associate type, variable, and procedure (respectively class, field, and method) uses with the right declaration and to guide *abrupt control flow* through the program structure[2]. The advantages of this pattern are ease of animation, since the structure of the program text is used, as well as *simplicity of understanding* the idea to move up the tree, until a matching value is found. We call this pattern *Tree-Structural-Approach* since instead of traditional structural approaches, where constructors are used, in this pattern we use the structure of the tree. In contrast to the traditional structural approach, this allows to move not only down the tree, but as well up until the root is reached. Some technical aspects of this pattern, namely the ASMs *enclosing* and *lookUp* have been discussed already in Section 5.3.3

[2] The abrupt control flow features use this pattern in combination with the later discussed *frame-result-controlflow* pattern.

- **Field-Of-Object-Mapping** The third specification pattern is the use of one binary dynamic function *fieldOf* to model the mapping of an object's field to its value. Given object o and field f, the value of the field is given by the term

  $$f.\mathit{fieldOf}(o)$$

  Different language features are unified under this view, for instance

  - *global variables* are considered to be fields of a constant *Global*,
  - *local variables* are considered to be fields of an object being the current call incarnation,
  - *static fields* are considered to be fields of the class containing the field, and
  - *instance fields* are of course fields of the object instance.

  We decided to name this pattern *Field-Of-Object-Mapping* since it uses one mapping to unify several related features under the view of a object/field model.

- **Frame-Result-Controlflow** The fourth and last specification pattern is a special case of the *Tree-Structural-Approach* pattern, combined with a global variable *RESULT* which is used to return various results from non-sequential control flow. Examples for such results are

  - a return-value produced by a function/method call,
  - a target-label produced by a break or continue statement, or
  - an exception-object produced by a throw statement or error condition.

  All of this constructs have in common that their "results" are passed up the structure tree, and that there is only one such result at the time. Therefore a global variable *RESULT* can be used to model the current value of the result.

  The pattern works such that as soon as a result is generated, control is passed up the tree, rather than along the control-flow arrows. If the type of *RESULT* matches the frame-node, thus if

  - a return-value reaches a call-statement
  - a target-label matches a labeled-statement
  - an exception-object triggers a catch-statement

  then the *frameHandler* processes the result, and resets *RESULT* to *undef*, otherwise the control is passed further up the tree to the next least enclosing frame-node.

  Each frame-node only needs to check whether the type of *RESULT* matches its own kind, and otherwise it passes control further up the tree. Therefore, such a specification will not change depending on what are the other cases of non-sequential control flow. This allows us to give completely independent models and to compose them easily for a full fledged language. A more technical description of the frame-result-controlflow pattern is given in Section 14.1.

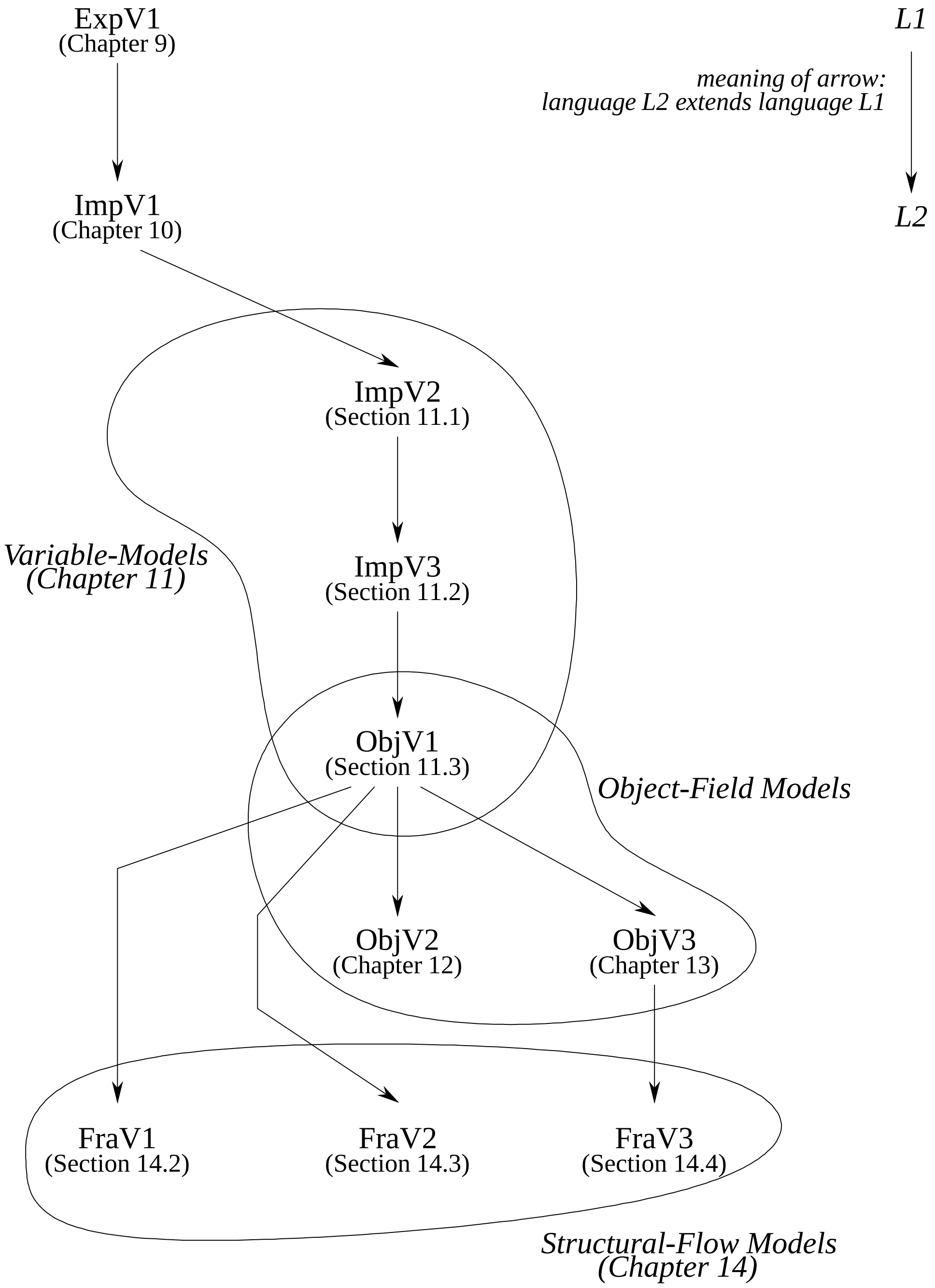


Fig. 44: The example languages of Part III

In Figure 44 the presented languages are depicted. The variable models of the first group are introducing stepwise the use of the first two pattern for reusable specifications of different kind of variables. The second group of *object-field models* shows how to specify object orientedness and recursive function-calls. In the third group different kinds of abrupt control flow are modeled with the *frame-result-controlflow* pattern. Each language and group is labeled by the chapter in which it is discussed.

The material is ordered such that each language can be formulated as an extension or refinement of its predecessor. An arrow from *L1* to language *L2* denotes that the definition of *L2* extends or refines the definition of *L1*. The *leave languages* of the resulting tree are specified such, that they can be easily combined to one big language with all introduced features. This is an indication that Montages allow to specify common language technology in a modular and composable way.

The language *ExpV1* is a simple expression language similar to the language introduced in Section 3. In contrast to its predecessor, this language features a rich choice of operators, as known from realistic programming languages. The remaining example languages are extensions of *ExpV1*, as denoted by the arrows in the figure. The first imperative language *ImpV1* extends *ExpV1* by introducing the concept of statements, blocks of sequential statements and conditional control flow. At this point we take advantage to give simple specifications of while and repeat loops, as well as a more advanced specification of the switch-statement. The concept of global variables is then introduced in example language *ImpV2*.

The purpose of languages *ImpV1* and *ImpV2* is to introduced features of a simple imperative language. In a series of refinements, the primitive, name based variable model of *ImpV2* is the further developed into the more sophisticated versions *ImpV3*, and finally *ObjV1*. Language *ObjV2* is an extension of *ObjV1* with classes and dynamically bound instance fields, and *ObjV3* is an extension of *ObjV1* with recursive procedure calls. The languages *FraV1* , *FraV2* , and *FraV3* feature iterative constructs, exception handling, and a refined model of procedure calls, respectively.

The presented example languages are an extract from a specification of sequential Java. The Java specification mainly differs from the languages presented here by a complex OO-type system, many exceptions and special cases, and a number of syntax problems. We have given the specification of the complete Java OO-type system as example in Appendix D.

9

Models of Expressions

In this chapter, we show an expression language *ExpV1*, where the intermediate results are computed during the execution of the program. The language works exactly like the example language $\mathcal{S}$ of Section 3.2 but features more operators and more different kinds of expressions. In addition, *ExpV1* has a simple type system, features lazy evaluation of disjunction and conjunction, and detects runtime errors such as division by zero. The grammar is given as follows, leaving away details on available unary and binary operators:

Gram. 10: *Program* ::= *exp*
exp = *lit* | *uExp* | *bExp* | *cExp*
lit = *Number* | *Boolean*
uExp ::= *“(” uOp exp “)”*
bExp ::= *“(” exp bOp exp “)”*
cExp ::= *“(” exp “?” exp “:” exp “)”*

In *ExpV1* only constant expressions such as the following can be formulated:

Ex. 1:

```
(((((3 - 2) * 7) > 2) and true) or false)
```

The result of executing this program is that ”true” is printed to the standard output.

9.1 Features of ExpV1

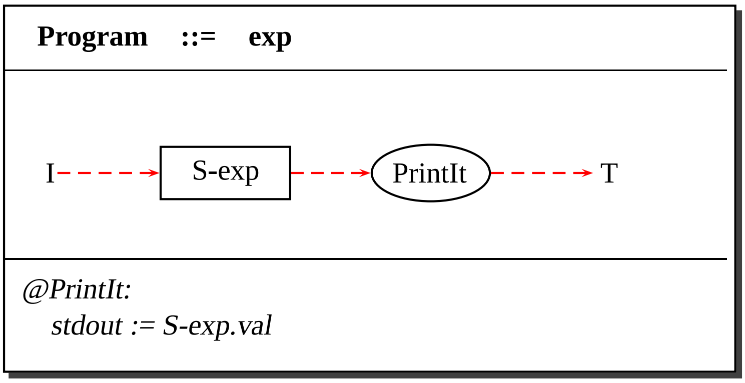
The start symbol of the language is *Program*, and each program consists of an expression, whose value is printed after the execution. The expressions are evaluated by storing the value of each subexpressions in an attribute *val*, which is modeled as a dynamic unary function.

Declarations
The signature of the global declarations consist of the single dynamic function *val(_)*, together with the derived function *defined(n) == (n != undef)* and the declaration of ASM *handleError(_,_)* which is used to handle run-time errors such as division by zero..

Decl. 1:

```
function val(_)
derived function defined(n) == ( n != undef)
external function handleError(_,_)
```

The Montage *Program* in Figure 45 specifies the semantics of the start-symbol of the ExpV1-language. The execution of such a program visits first the *exp*-component and then the *PrintIt*-state is visited. The *PrintIt*-action outputs the attribute *val* of the *exp*-component to the standard output *stdout*.



Program ::= exp

@PrintIt:
stdout := S-exp.val

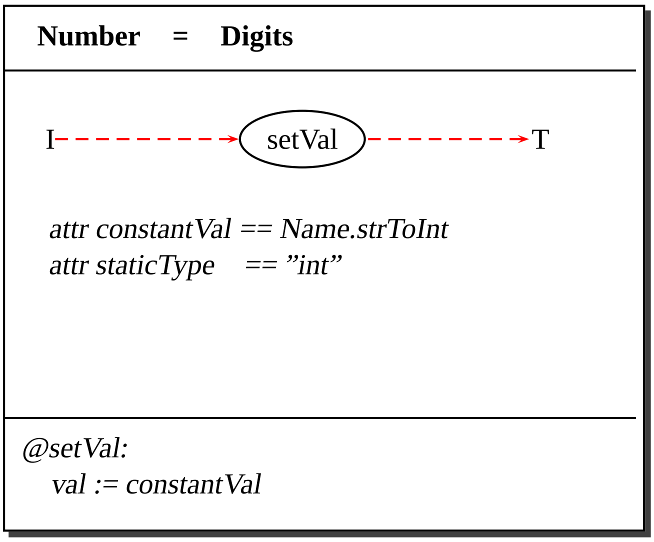
Fig. 45: Montage *Program* of language *ExpV1*

9.1.1 The Atomar Expression Constructs

The atomar expression constructs *Number* (Figure 46) and Boolean (Figure 47) use both a derived attribute *constantVal* to calculate their constant values. The definition of *constantVal* in the *Number*-Montage uses the built-in *Name*-attribute to get the parsed string-value of the *Digits*-token, and then applies the built-in *strToInt*-function to transform the string-value in an integer. The corresponding definition for the *Boolean*-Montage transforms the strings “true” or “false” in the corresponding elements *true* and *false*. The dynamic semantics of both constructs consists of the unique state *setVal* whose action updates the *val*-attribute to *constantVal*.

9.1.2 The Composed Expression Constructs

The unary expression *uExp* is specified in Figure 48. The components of a unary expressions are a unary operator *uop* and an expression. The local definitions of *uExp* contain the derived function *Apply(_,_)*

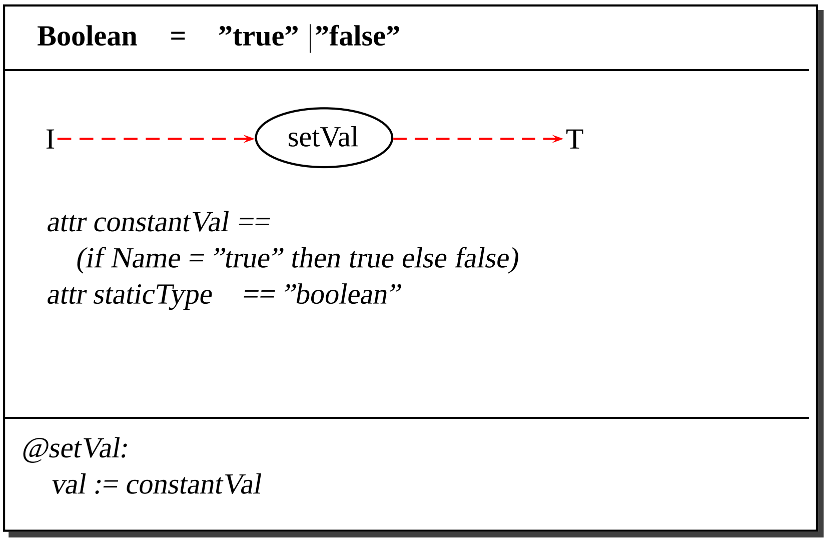


Number = Digits

attr constantVal == Name.strToInt
attr staticType == ”int”

@setVal:
val := constantVal

Fig. 46: Montage *Number* of language *ExpV1*



Boolean = ”true” |”false”

attr constantVal ==
(if Name = ”true” then true else false)
attr staticType == ”boolean”

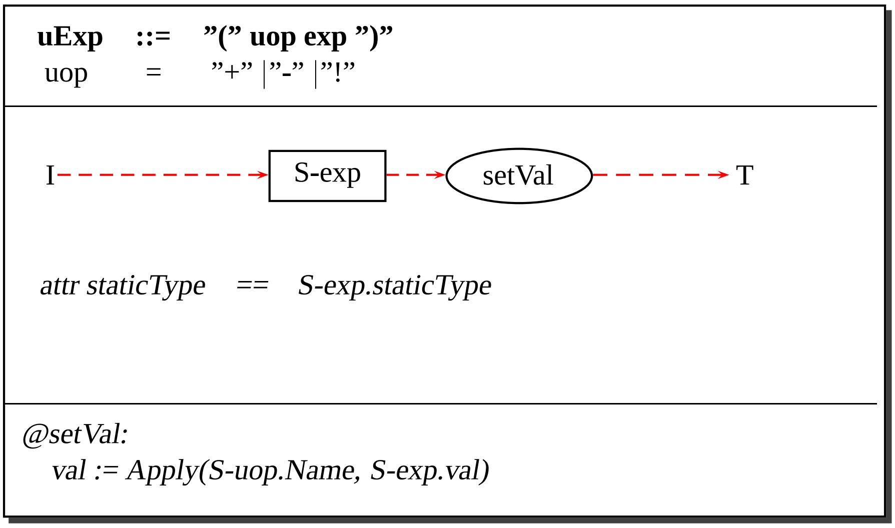
@setVal:
val := constantVal

Fig. 47: Montage *Boolean* of language *ExpV1*

Decl. 2:

```
derived function Apply(op, arg) ==
(if (arg = undef) then undef
else (if op = "+" then arg
else (if op = "-" then 0 - arg
else (if op = "!" then not arg
else undef
))))
```

which is used in the action *setVal* to calculate the result of the expression and to set the *val*-attribute to said result.



uExp **::=** **”(” uop exp ”)”**
uop = ”+” |”-” |”!”

attr staticType == S-exp.staticType

@setVal:
val := Apply(S-uop.Name, S-exp.val)

Fig. 48: Montage *uExp* of language *ExpV1*

Binary Expression

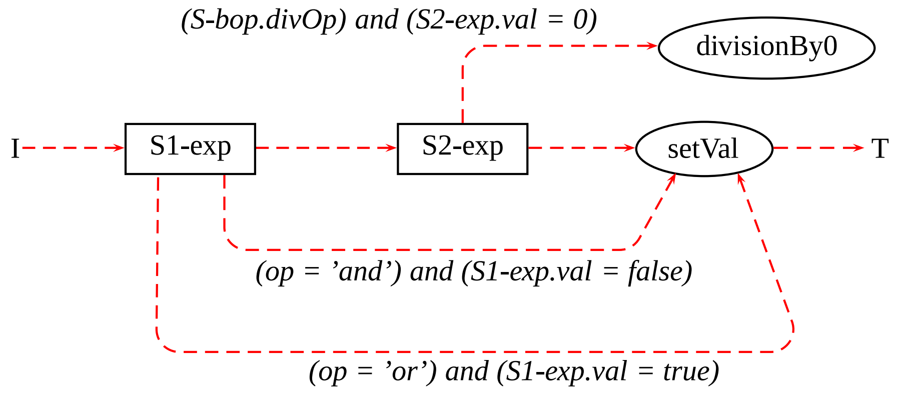
The binary expression Montage is shown in Figure 49. For standard operations, control flows through the two expressions, and then the *setVal*-action sets the *val*-attribute to the calculated value of the binary expression. The arguments to calculate the value are in the *val*-attributes of the left and right expression, respectively. This standard case is visualized in Figure 50 and corresponds exactly to the Sum Montage in Section 3.3.1, Figure 21.

Before we explain the other cases of control flow, we give the definition of the *Apply* function.

Decl. 3:

```
derived function Apply(op, arg1, arg2) ==
(if op = "and"                    then
        (if arg1 = false then false else arg2)
else (if op = "or"                    then
        (if arg1 = true  then true  else arg2)
else (if (arg1 = undef) or
         (arg2 = undef)    then undef
else (if op = "+"          then arg1 + arg2
else (if op = "-"          then arg1 - arg2
else (if op = "*"          then arg1 * arg2
else (if op = "/"          then arg1 / arg2
else (if op = "%"          then arg1 / arg2
```

bExp ::= ”(” exp bop exp ”)”
bop = arithOp |relOp |”and” |”or”
relOp = ”<” |”<=” |”>” |”>=” |”==” |”!=”
arithOp = divOp |”*” |”+” |”-”
divOp = ”/” |”%”

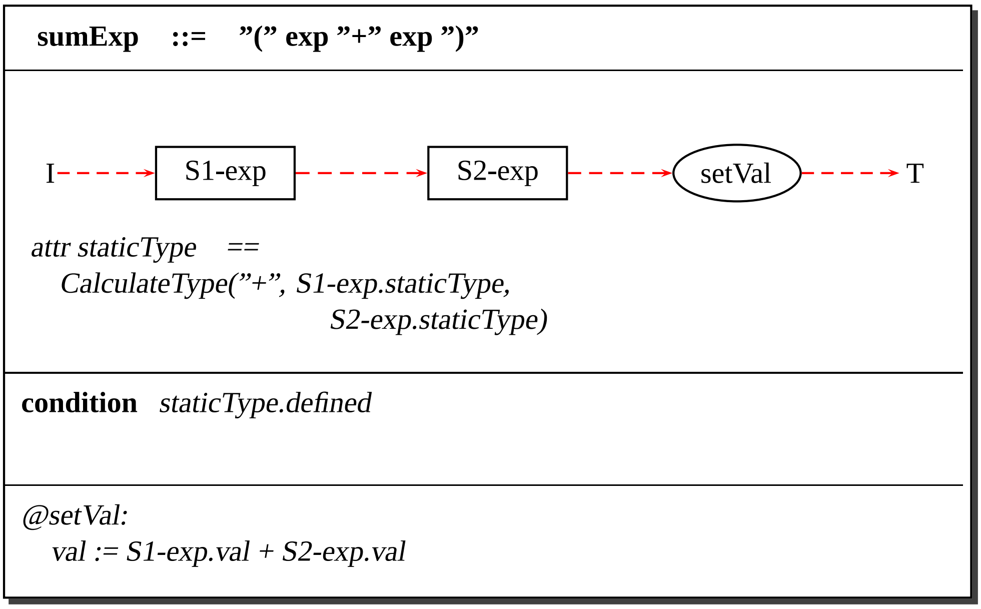

(S-bop.divOp) and (S2-exp.val = 0)
divisionBy0
I
S1-exp
S2-exp
setVal
T
(op = ’and’) and (S1-exp.val = false)
(op = ’or’) and (S1-exp.val = true)

attr op == S-bop.Name
attr staticType ==
CalculateType(op, S1-exp.staticType,S2-exp.staticType)

condition *staticType.defined*

@setVal:
val := Apply(S-bop.Name, S1-exp.val, S2-exp.val)
@divisionBy0:
handleError(”ArithmeticException”)

Fig. 49: Montage *bExp* of language *ExpV1*



sumExp ::= ”(” exp ”+” exp ”)”

attr staticType ==
CalculateType(”+”, S1-exp.staticType,
S2-exp.staticType)

condition *staticType.defined*

@setVal:
val := S1-exp.val + S2-exp.val

Fig. 50: Montage *sumExp* of language *ExpV1*

```
else (if op = "=="          then arg1 = arg2
else (if op = "!="          then arg1 != arg2
else (if op = "<"           then arg1 < arg2
else (if op = ">"           then arg1 > arg2
else (if op = "<="          then arg1 <= arg2
else (if op = ">="          then arg1 >= arg2
else  undef
)))))))))))))
```

Lazy evaluation of conjunction

In the flow specification of Figure 49 we see several control arrows, in addition to the described standard way. The first of them, departing from the *S1-exp* component directly to the *setVal*-action is labeled with the condition

(op = “and”) and (S1-exp.val = false)

This arrow guarantees that for the *and* operation the second argument is only evaluated if the first argument evaluates to true. This behavior is called “lazy evaluation” of conjunction, and is important, if the evaluation of the second argument has side-effects.

Lazy evaluation of disjunction

Similarly, the flow arrow labeled with

(op = “or”) and (S1-exp.val = true)

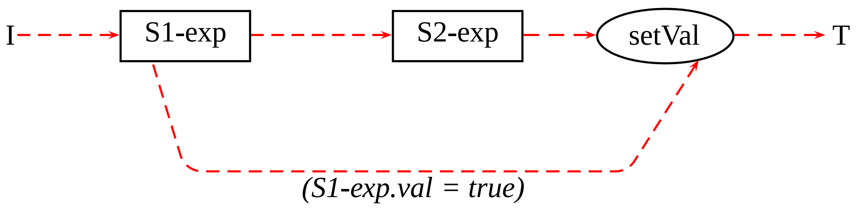
specifies lazy evaluation of disjunction.

Division by zero

The arrow departing from the second expression to the *divisionBy0*-action catches the case when the operand is a division, and the second expression evaluates to zero. The action *divisionBy0* calls the ASM *handleError*. In this language the definition of *handleError* simply prints error messages to the standard output. If in a later stage, the same Montage is reused in connection with exception handling, the definition of ASM *handleError* can be refined to a rule triggering a “division by 0” exception.

Coming back to the concept of partial evaluation, as discussed in Section 5.5 it is interesting and instructive to look at specialized Montages resulting from considering the binary operators to be static and to partially evaluate all expressions with this information. Examples for Montages resulting from such a specialization of Montage *bExp* are Montage *sumExp* (Figure 50), Montage *orExp* (Figure 51), and Montage *divExp* (Figure 52).

orExp ::= ”(” exp ”or” exp ”)”



```
attr staticType ==
   CalculateType(”or”, S1-exp.staticType,
                              S2-exp.staticType)
```

condition *staticType.defined*

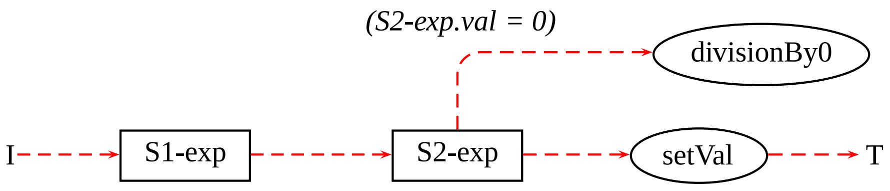
```
@setVal:
   if S1-exp.val then
      val := true
   else
      val := S2-exp.val
   endif
```

Fig. 51: Montage *orExp* of language *ExpV1*

Conditional Expression

The conditional expression *cExp* is specified in Figure 53. The control enters initially the first expression, and if it evaluates to true *true*, control flows along the upper arrow to the second expression; otherwise control flows along the lower arrow to the third expression. From those expressions control flows in the *setVal*-action. This action updates the attribute *val*.

divExp ::= ”(” exp ”/” exp ”)”



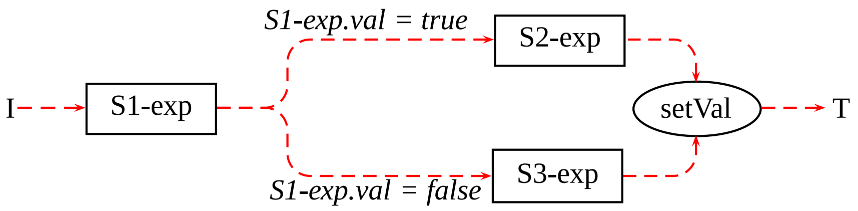
attr staticType ==
CalculateType(”/”, S1-exp.staticType,
S2-exp.staticType)

condition *staticType.defined*

@setVal:
val := S1-exp.val / S2-exp.val
@divisionBy0:
handleError(”ArithmeticException”)

Fig. 52: Montage *divExp* of language *ExpV1*

cExp ::= ”(” exp ”?” exp ”:” exp ”)”



attr staticType == lcst(S2-exp.staticType, S3-exp.staticType)

condition *staticType.defined AND S1-exp.staticType*
= ”boolean”

@setVal:
val := (if S1-exp.val then S2-exp.val else S3-exp.val)

Fig. 53: Montage *cExp* of language *ExpV1*

i: introduced	**r:** refined u: used	ExpV1	ImpV1	ImpV2	ImpV3	ObjV1	ObjV2	ObjV3	FraV1	FraV2	FraV3
Concept	Description										
Program	start symbol of each language	**i**	**r**	u	u	u	**r**	**r**	**r**	u	**r**
exp	synonym for expressions	**i**	**r**	**r**	u	u	**r**	**r**	**r**	u	**r**
lit	synonym for litterals	**i**	u	u	u	u	u	u	u	u	u
Number	the number litteral	**i**	u	u	u	u	u	u	u	u	u
Boolean	the Boolean litteral	**i**	u	u	u	u	u	u	u	u	u
uExp	unary expression	**i**	u	u	u	u	u	u	u	u	u
bExp	binary expression	**i**	u	u	u	u	u	u	u	u	u
cExp	conditional expression	**i**	u	u	u	u	u	u	u	u	u

Fig. 54: Roaster of ExpV1 features and their introduction (i), refinement (r), and use (u) in the different languages

9.2 Reuse of ExpV1 Features

Figure 54 displays the so called ”feature roaster” of *ExpV1* showing which languages are reusing and refining the *ExpV1*-features. In the first column, the symbols of the features are listed. After a short description, there is one column per language, and for each feature it is marked whether it is

(i) introduced,

(r) refined, or

(u) used

by a language. The column of *ExpV1* shows of course an (i) for each feature, since *ExpV1* is the first language we define. The remaining columns show that all features with exception of *exp* and *Program* are used without refining them by all other languages. The symbol *exp* is a synonym for expressions, and is of course refined each time a new expression is introduced. The symbol *Program* is only used to make each example language a testable, complete language. It is therefore different for each language.

The feature roaster is shown for each example language in order to visualize the high level of exact reuse and modularity of our specifications.

10

Models of Control Flow Statements

In this chapter we introduce the concept of *statements* and their sequential execution. We start with an example language *ImpV1* featuring a simple *print* statement, and the *if-then-else* statement.

The basic concept of a statement is a program construct that can be *executed*, and through its execution it has effects or changes the state, while, in contrast, an expression is a program construct that can be *evaluated*, and through its evaluation delivers a result. Thus programming languages without state, e.g. pure functional languages do not feature statements. On the other hand in most imperative and object oriented languages the evaluation of expressions may change the state as well, thus the evaluation of expressions in such languages delivers a result *and* changes the state. Montages is especially well suited for languages with such "un-pure" expression concepts.

i: introduced **r:** refined u: used		ExpV1	ImpV1	ImpV2	ImpV3	ObjV1	ObjV2	ObjV3	FraV1	FraV2	FraV3
Concept	Description										
Program	start symbol of each language	**i**	**r**	u	u	u	**r**	**r**	**r**	u	**r**
exp	synonym for expressions	**i**	**r**	**r**	u	u	**r**	**r**	**r**	u	**r**
lit, Number, Boolean, uExp, bExp, cExp											
stm	synonym for statements		**i**	**r**	**r**	u	u	**r**	**r**	**r**	**r**
block	sequential block of statements		**i**	u	**r**	u	u	u	u	u	u
printStm	the print statement		**i**	u	u	u	u	u	u	u	u
ifStm	the if statement		**i**	u	**r**	u	u	u	u	u	u

Fig. 55: Roaster of ImpV1 features and their introduction (i), refinement (r), and use (u) in the different languages

In Section 10.1 we show the grammar, an example program, the Montages, and the feature roaster of language *ImpV1*. A number of additional control statements are shown in Section 10.2: switch, while, repeat, and for statements. Later in Chapter 14 the *while* and *repeat* statements will be refined with versions allowing for *break* and *continue*

10.1 The Example Language ImpV1

We use again a small example language to explain the new constructs. The language is called *ImpV1* and its grammar is

Gram. 11:

Program	=	*block*
block	::=	“{” *stm* “}”
stm	=	“;” | *printStm* | *ifStm* | *block*
printStm	::=	“*print*” *exp* “;”
ifStm	::=	“*if*” *exp block*
		[“*else*” *block*]
exp	=	**(see Gram. 10)**

where the definitions for the expressions are inherited from ExpV1 in Chapter 9, Grammar 10 and the Montages in Figures 46 through 53.

The Block Statement

The Montage for the *block* statement of *ImpV1* is given in Figure 56. The list of *stm*-components is represented graphically by the special list-box. By default the components of the list are connected sequentially by flow arrows. Thus the control flow enters the list at the first element and traverses it sequentially.

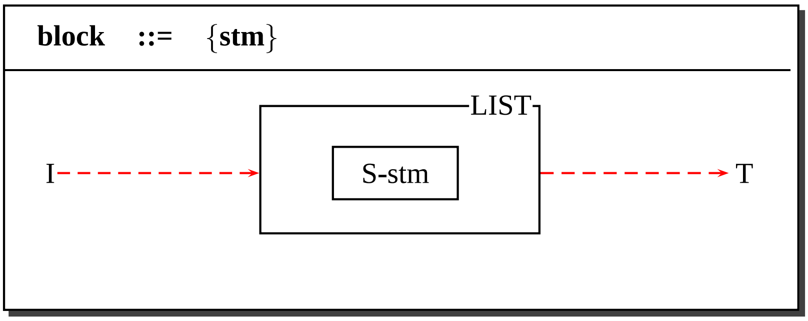
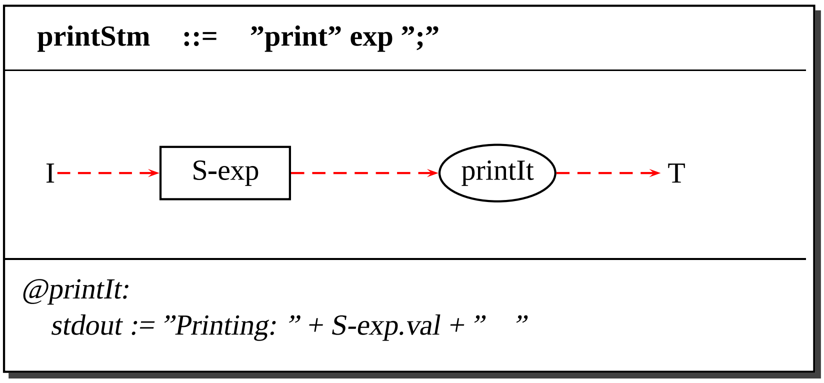


Fig. 56: Montage *block* of language *ImpV1*

The Print Statement

The print statement is specified by the *printStm*-Montage (Figure 57). The dynamic semantics of the print statement evaluates first the *exp*-component, and then in the *printIt* state the *val*-attribute of the *exp*-component is sent to standard output.



printStm ::= ”print” exp ”;”

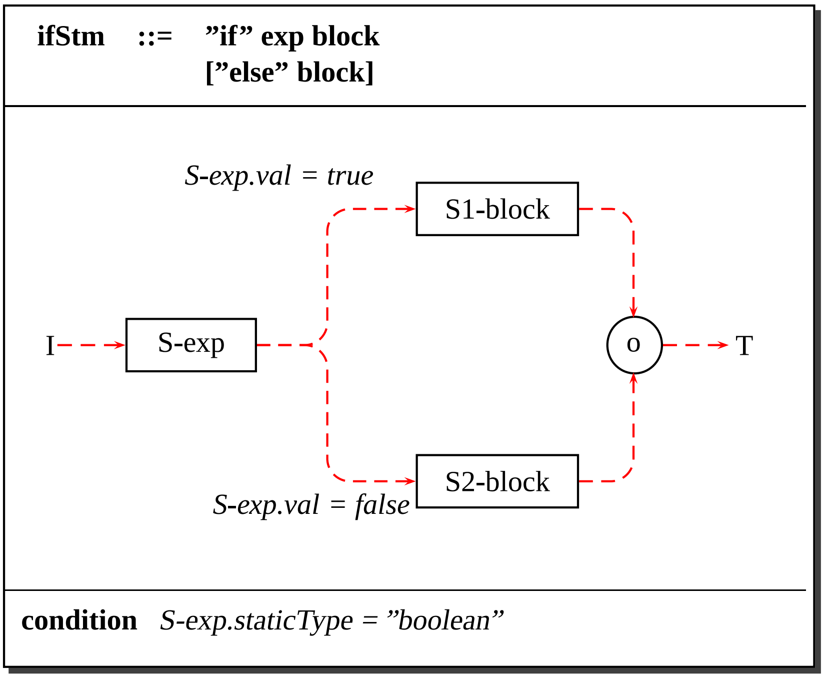
@printIt:
stdout := ”Printing: ” + S-exp.val + ” ”

Fig. 57: Montage *printStm* of language *ImpV1*

The If-Then-Else Statement

The if-statement of our example language is specified in Figure 58. In order to avoid the usual “dangling-if” problem the if-syntax of *ImpV1* forces the user to give each time a block included in curly brackets. The else-part is made optional.

The control-flow specification is similar to that found in the *cExp*-Montage, Figure 53. The control enters the if construct at the *exp*-component, and then, depending whether the expression evaluates to true or false, control flows to the first or the second block.



ifStm ::= ”if” exp block
[”else” block]

condition *S-exp.staticType = ”boolean”*

Fig. 58: Montage *ifStm* of language *ImpV1*

10.2 Additional Control Statements

The following statements are sketched in order to demonstrate the compactness and ease of readability of Montages of different control flow statements.

The While and Repeat Statements

In Figure 59 we present a simplified version of the while-statement of Section 3.1, Figure 10, where the domain specific action is left away, and *ExpV1* style typing is added. This Montages is closely related to the ”repeat...until”- or ”do..while”-statement shown in Figure 60. Comparing the two Montages we see how a subtle difference in the semantics of while and repeat is visually documented.

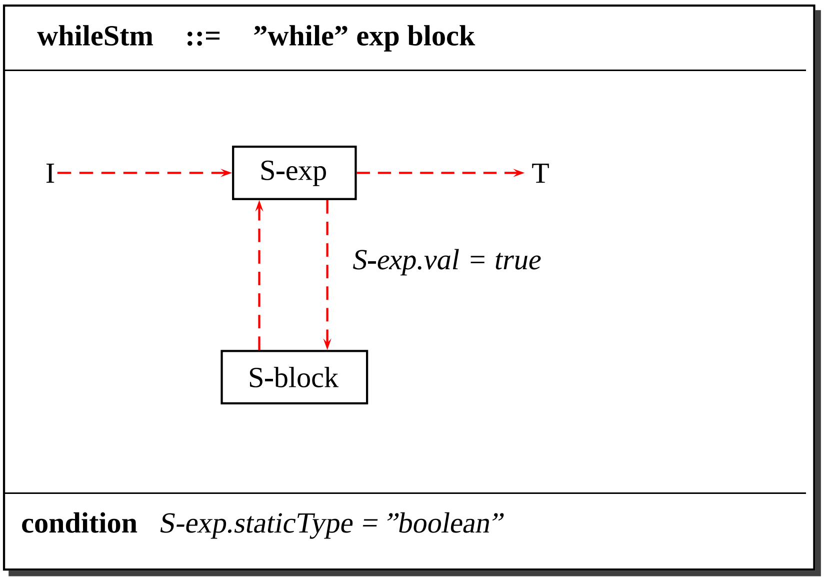

whileStm ::= ”while” exp block
I
S-exp
T
S-exp.val = true
S-block
condition S-exp.staticType = ”boolean”

Fig. 59: Montage *whileStm* of language *ImpV1*

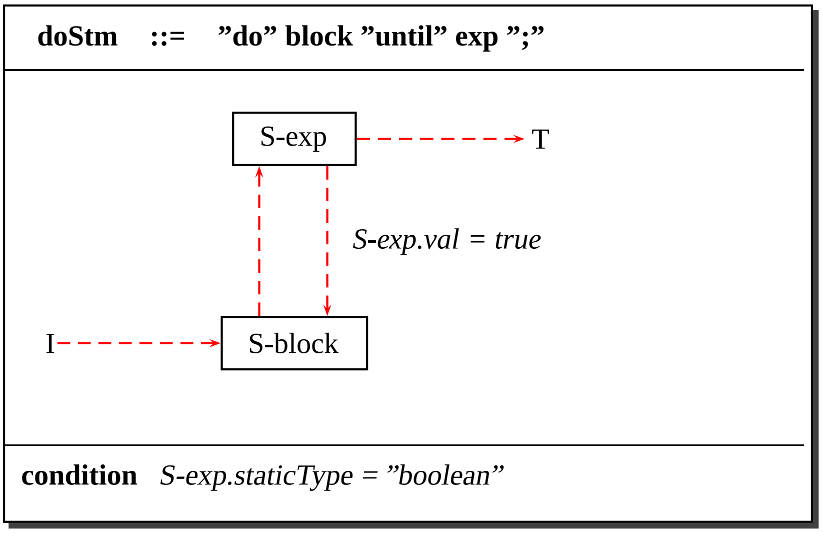

doStm ::= ”do” block ”until” exp ”;”
S-exp
T
S-exp.val = true
I
S-block
condition S-exp.staticType = ”boolean”

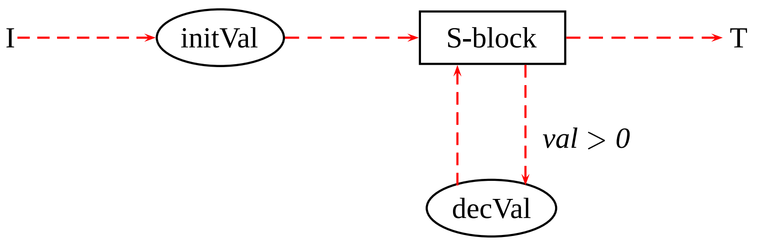
Fig. 60: Montage *doStm* of language *ImpV1*

A Simple For Statement

In Figure 61 a very simple for-statement is shown. Two integer expressions are given and then the block is repeated x times, where x is the difference between the two expressions. The val-attribute is used to remember how many times the loop has already been executed.

This example is given to show how easy a new iteration construct can be specified, and how near the specification techniques for the semantics are to common programming techniques. The way how the var-field is used to count the repetitions is very similar to the way a programmer would solve the same problem.

forStm ::= ”for” exp ”to” exp block



condition *(S1-exp.staticType = ”int”)*
andthen (S2-exp.staticType = ”int”)

```
@initVal:
  if S1-exp.val > S2-exp.val then
    val := S1-exp.val - S2-exp.val
  else
    val := S2-exp.val - S1-exp.val
  endif

@decVal:
  val := val - 1
```

Fig. 61: Montage *forStm* of language *ImpV1*

The Switch Statement

The switch statement is a kind of more powerful conditional-statement. Depending on the value of an expression, the statement ”switches” to one of different statements marked by labels. The statements following the selected statement are executed as well, a behavior called ”fall through”. The following EBNF productions extend Grammar 11 with a switch-statement.

Gram. 12: (refines Grammar 11)

stm = . . . | *switchStm*
switchStm ::= *“switch” exp* “{” { *switchLabelOrStm* } “}” “;”
switchLabelOrStm = *stm* | *defaultLabel* | *caseLabel*
defaultLabel ::= *“default” “:”*
caseLabel ::= *“case” Number*

In Figure 62, 63, and 64 the Montages for *switchStm*, *defaultLabel*, and *caseLabel* are given.

The components of the *switchStm*-Montage are an expression, the *exp*-component, and a list of components being statements, or labels. Some control arrows in this Montage use the *src* and *trg* functions, which denote in arrow-labels the origin and target nodes of the arrow. Further two arrows go not to the list-node, but the node inside the list-node. These arrows denote a family of arrows, one to each component of the list.

The control flows first through the *exp*-component. From there, a family of flow-arrows labeled *trg.hasLabel(src.val)* leads to the components in the list. Such a label evaluates to true, only if the target of the corresponding arrow is a *caseLabel* and if that label has a constant value equivalent to the just evaluated *exp*-component. If the control cannot flow along any of these arrows, it flows to the *default*-action. From there sources another family of arrows leading to the components of the list. The flow condition *trg.isDefault* on these arrows leads control directly to the default label in the list. If there is no such label, control flows to the *T*-action. If any of the discussed arrows led control into the list, all remaining components of the list are executed sequentially. This property is called “fall-through” and typically it is expected to use in most cases an explicit “jump”[1] to break out of the switch without falling through all the remaining cases. In our little language, jumping out is not possible.

[1] Break or continue.

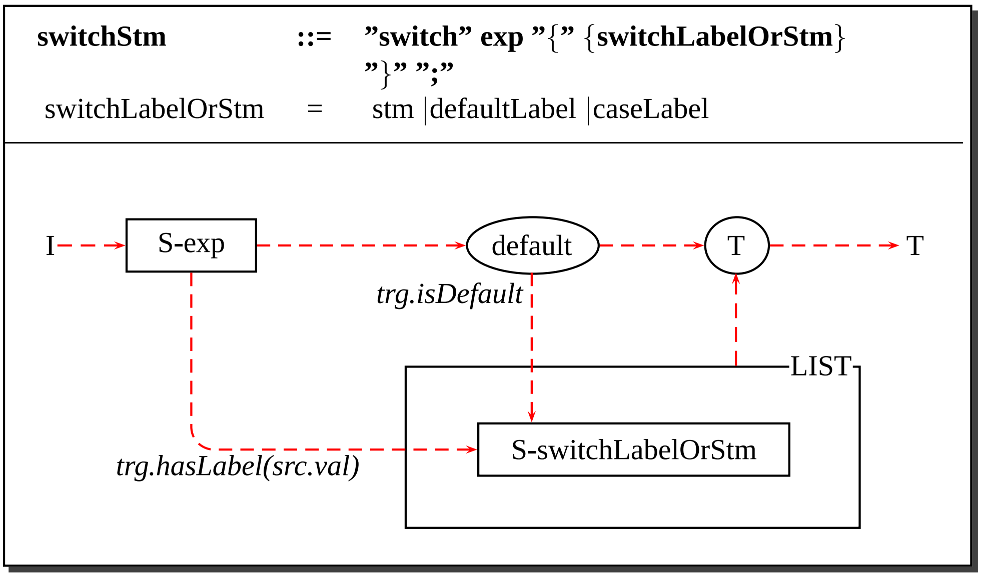

switchStm ::= ”switch” exp ”{” {switchLabelOrStm} ”}” ”;”
switchLabelOrStm = stm |defaultLabel |caseLabel
I
S-exp
default
T
T
trg.isDefault
LIST
S-switchLabelOrStm
trg.hasLabel(src.val)

Fig. 62: Montage *switchStm* of language *ImpV1*

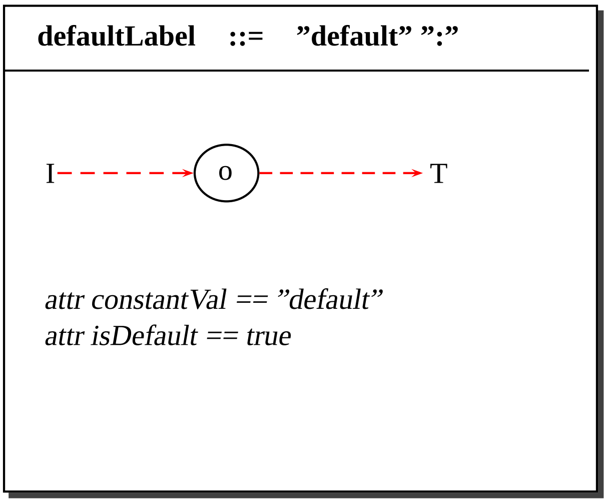
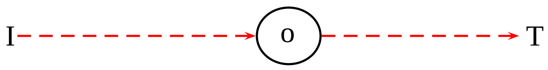

defaultLabel ::= ”default” ”:”
I
o
T
attr constantVal == ”default”
attr isDefault == true

Fig. 63: Montage *defaultLabel* of language *ImpV1*

caseLabel ::= ”case” Number ”:”



attr constantVal == S-Number.constantVal
attr staticType == S-Number.staticType
attr hasLabel(l) == constantVal = l

condition *constantVal.defined*

Fig. 64: Montage *caseLabel* of language *ImpV1*

11

Models of Variable Use, Assignment, and Declaration

Unlike a mathematical variable, which serves as placeholder for values, a variable in imperative and object oriented programming languages is a kind of *box* which is used to store a value. The value stored in the box is called the *value of the variable*. The action to exchange the content of the box is called *variable update* or *variable assignment*. After a variable x has been updated with value v, the content, or value of x remains v until the next update of x. In expressions a variable can be *used* like a constant.

Modeling variables in XASM can be done in a number of different ways. The simplest, but most inflexible choice is to model each variable as a 0-ary dynamic function. This solution has already been explained in the introduction of Part II where we as well discussed the advantages of using this model in combination with partial evaluation. In Section 11.1 we present a full example language with global variables *ImpV2* based on this solution.

The disadvantage of this first solution is that two incarnations of a variable named ”x” cannot coexist, since the name of the variable is used as it's identity. A pattern to solve this problem is the *Declarator-Reification* patter, which uses the declaration of a variable as its identity. Combining this pattern with the *Tree-Structural-Approach* pattern allows then to easily introduce several nested scopes. This solution to variable use and declaration is presented in form of the example language *ImpV3* in Section 11.2. This language features nested blocks of statements with nested scopes of variable names.

The advantages of this second model are ease of animation and ease of specification. Further it may be an advantage that parameterized XASM are not needed for this kind of model. In general, PXasm are used in the rules and declarations if a special kind of production code has to result, and they are not needed for an abstract model serving as prototype and documentation of the language.

Finally, in Section 11.3 the variable model is refined using the *Field-Of-Object-Mapping* pattern. The declarations of the variables are interpreted to be fields of a constant element *Global*. In addition we extend the specification of the assignment construct such that it can model both assignments to simple variables, as well as assignments to variables calculated by expressions. Such assignable expressions are called *use*, and if they are evaluated they evaluate not only their value, or *right value*, but as well the variable, or left value. The same pattern is then used in the next two chapters to model an object oriented example language and recursive procedure calls.

11.1 ImpV2: A Simple Name Based Variable Model

In this Section we define the example language *ImpV2* by extending *ImpV1* with simple, name based models for variable update and use.

Using the symbol *asgnStm* for the variable-update, and the symbol *use* for the variable-use in expressions we extend the grammar rules *stm* and *exp* as follows, reusing the other definitions of Grammar 11. The ... notation in synonym productions denotes that all choices of the predecessor language are reused, and extended with some additional synonyms[1].

Gram. 13: (refines Grammar 11)

stm = ... | *asgnStm*
exp = ... | *use*
asgnStm ::= *id* "=" *exp*
use = *id*

An overview on the features and their reuse/refinement is given in *ImpV2*'s feature roaster, Figure 65. The two new constructs *use* and *assign* are going to be refined twice in *ImpV3* and in *ObjV1*.

Declarations

Variables in *ImpV2* must not be declared, and each used or assigned variable is directly modeled as an 0-ary dynamic XASM function which is initialized as 0. The PXasm declaration of those functions for all used variables is given as follows.

Decl. 4:

```
(for all s in String:
      (exists a in asgnStm: a.S-id.Name = s) or
      (exists u in use: u.Name = s)
  function $s$ <- 0
)
```

[1] As we mentioned in the introduction, we would need to extend Montages with inheritance mechanisms to formalize the notion of "reused" and "extended" but have not done this.

i: introduced	**r:** refined u: used	ExpV1	ImpV1	ImpV2	ImpV3	ObjV1	ObjV2	ObjV3	FraV1	FraV2	FraV3
Concept	Description										
Program	start symbol of each language	**i**	**r**	u	u	u	**r**	**r**	**r**	u	**r**
exp	synonym for expressions	**i**	**r**	**r**	u	u	**r**	**r**	**r**	u	**r**
use	the use expression			**i**	**r**	**r**	u	u	u	u	u
..., lit, Number, Boolean, uExp, bExp, cExp											
stm	synonym for statements		**i**	**r**	**r**	u	u	**r**	**r**	**r**	**r**
block	sequential block of statements		**i**	u	**r**	u	u	u	u	u	u
asgnStm	the assignment statement			**i**	**r**	**r**	u	u	u	u	u
..., printStm, ifStm											

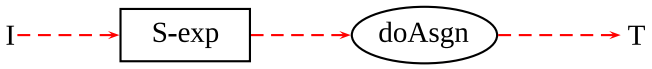
Fig. 65: Roaster of ImpV2 features and their introduction (i), refinement (r), and use (u) in the different languages

Assignment Statement

The specification of syntax and semantics of *asgnStm* is shown in Figure 66. The attribute *signature* is introduced for readability, and denotes the identifier representing the variable to be updated.

The control flows through the *asgnStm* by first evaluating the expression, and then triggering the *doAsgn*-action, doing the update by updating the function named after the string value of *signature*. The $ operator is used to refer to the 0-ary function corresponding to the value of *signature*.

asgnStm ::= id ”=” exp ”;”



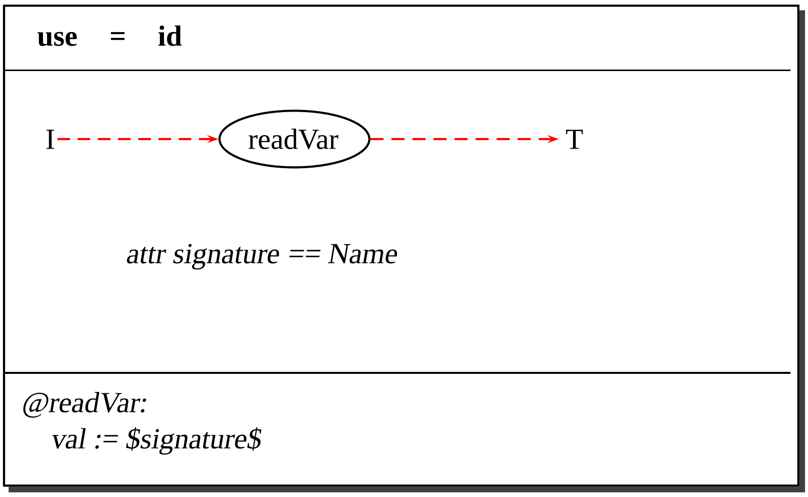
attr signature == S-id.Name

@doAsgn:
$signature$:= S-exp.val

Fig. 66: Montage *asgnStm* of language *ImpV2*

Use Expression

The *use*-Montage in Figure 67 consists mainly of the *readVar*-action, which sets the *val*-attribute of the *use*-expression to the value of the 0-ary function whose signature corresponds to the value of the *signature*-attribute.



```
attr signature == Name
```

```
@readVar:
  val := $signature$
```

Fig. 67: Montage *use* of language *ImpV2*

11.2 ImpV3: A Refined Tree Based Variable Model

In this section we define the language *ImpV3* featuring a refined tree based model for variable *declaration*, *use*, and *update*. Variable must be declared prior to their use. The feature-roaster in Figure 68 shows that in addition *block* and the block-statements *bstm* are introduced and reused without further refinement by all following languages. The grammar of *ImpV3* is given by extending and refining the definitions of Grammar 13.

Gram. 14: (refines Grammar 13)

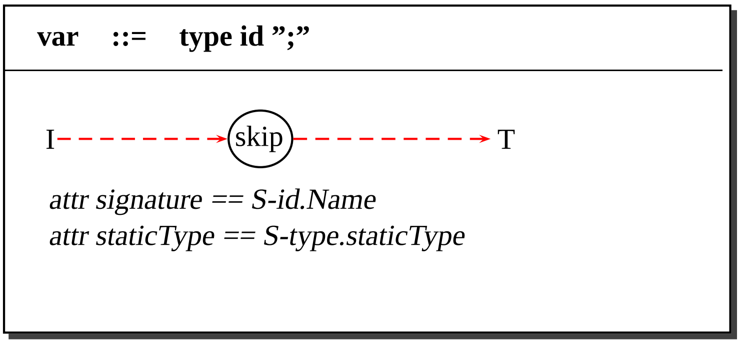
stm = ... | *block*
block ::= *"{" {bstm } "}"*
bstm = *var* | *stm*
var ::= *type id ";"*
type = *"int"* | *"boolean"*

A declaration consists of the keyword *var*, the *type* and the *name* of the variable. Variables are represented by the node being their declaration in the program. Blocks can contain variable declarations and can be nested, e.g. a block can contain another block. The nesting of blocks defines so called *scopes* or name spaces.

The var-Montages (Figure 69) and the type-Montage (Figure 70) contain only attribute definitions. In the var-Montage, for instance, the signature-attribute returns the name of the variable, and the staticType-attribute returns the static type of the type-nodes. These attributes are used for basic type checks in *ImpV3*-programs. The dynamic semantic of var does nothing, a situation which is here explicitly specified with a state "skip" having no action associated. This "skip" behavior is the default behavior of a Montages if no states and arrows are given.

i: introduced **r:** refined u: used											
Concept	Description	ExpV1	ImpV1	ImpV2	ImpV3	ObjV1	ObjV2	ObjV3	FraV1	FraV2	FraV3
Program	start symbol of each language	**i**	**r**	u	u	u	**r**	**r**	**r**	u	**r**
exp	synonym for expressions	**i**	**r**	**r**	u	u	**r**	**r**	**r**	u	**r**
use	the use expression			**i**	**r**	**r**	**r**	u	u	u	u
lit, Number, Boolean, uExp, bExp, cExp											
stm	synonym for statements		**i**	**r**	**r**	u	u	**r**	**r**	**r**	**r**
asgnStm	the assignment statement			**i**	**r**	**r**	**r**	u	u	u	u
block	list of block statements				**i**	u	u	u	u	u	u
bstm	statement or variable declaration				**i**	u	u	u	u	u	u
printStm, ifStm											
var	variable declaration				**i**	u	u	u	u	u	u
type	types of the language				**i**	u	**r**	**r**	u	u	u

Fig. 68: Roaster of ImpV3 features and their introduction (i), refinement (r), and use (u) in the different languages


var ::= type id ";"

attr signature == S-id.Name
attr staticType == S-type.staticType

Fig. 69: Montage *var* of language *ImpV3*

type = "int" |"boolean"

attr staticType == Name

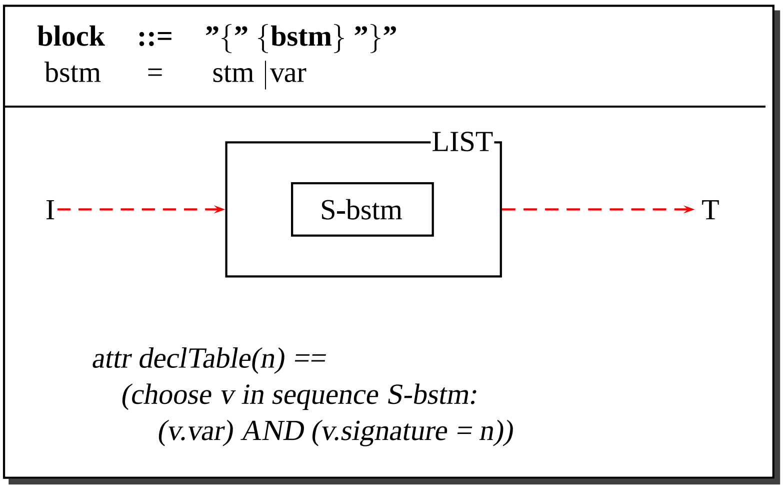
Fig. 70: Montage *type* of language *ImpV3*

As mentioned, the block-statement may contain not only statements, but as well variable declarations. The *block*-Montage in Figure 71 links the execution of the mixed statement and variable list sequentially. The unary attribute *declTable(_)*

Attr. 1:

```
attr declTable(n, id) ==
  (choose v in sequence n.S-bstm:
     (v.var) AND (v.signature = id))
```

returns a *var*-component of the *bstm*-list, whose *signature* equals the argument of *declTable(_)*; is no such component exists, it returns *undef*.



block ::= ”{” {**bstm**} ”}”
bstm = stm |var

attr declTable(n) ==
(choose v in sequence S-bstm:
(v.var) AND (v.signature = n))

Fig. 71: Montage *block* of language *ImpV3*

The attribute *declTable* is used by the function *lookUp(_, _)* which has been introduced in Section 5.3.3, as ASM 21, and which uses the ASM 20, *enclosing(_, _)*. The ASM *enclosing* in turn relies on an appropriate definition of *Scope*, being a set of Montages-names serving as scopes. For our Grammar 14 the correct definition of *Scope* is

Decl. 5:

```
derived function Scope == {‘‘block’’}
```

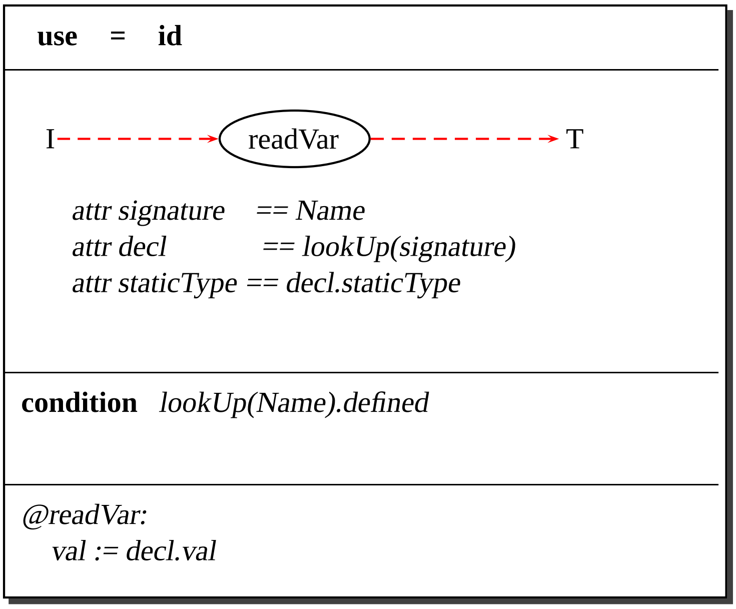
the set consisting of the single string element “block”.

The new versions of the *use* and *asgnStm* Montages in Figure 72 and 73 both contain the attribute definition

Attr. 2:

```
attr decl == lookUp(signature)
```

for accessing the identity of the variable. Read and write accesses to the variable are then done by updating and reading the unary dynamic function *val(_)*. With other words expressions and variable declarations in the abstract syntax tree are interpreted as objects whose value is given by the attribute *val*. The difference is that expressions values are only implicitly updated during their evaluation, and variable values are explicitly updated using an assignment statement.



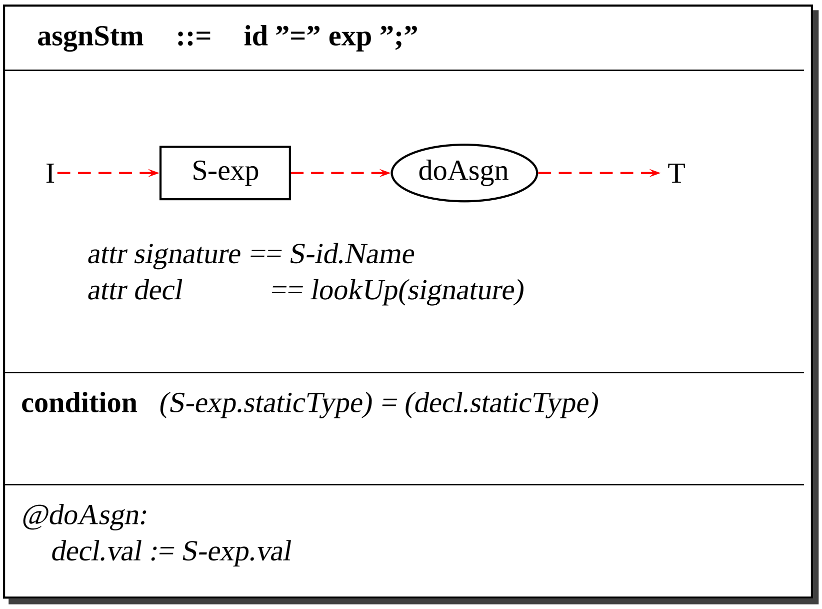
use = id

attr signature == Name
attr decl == lookUp(signature)
attr staticType == decl.staticType

condition *lookUp(Name).defined*

@readVar:
val := decl.val

Fig. 72: Montage *use* of language *ImpV3*



asgnStm ::= id "=" exp ";"

attr signature == S-id.Name
attr decl == lookUp(signature)

condition *(S-exp.staticType) = (decl.staticType)*

@doAsgn:
decl.val := S-exp.val

Fig. 73: Montage *asgnStm* of language *ImpV3*

11.3 ObjV1: Interpreting Variables as Fields of Objects

A further refinement of the model in the last section is given by language *ObjV1* which uses directly the *Field-Of-Object-Mapping* pattern, modeling the global variables, e.g. the reification of their declarations as fields of a constant *Global*. The grammar remains unchanged. The new declarations for *val, fieldOf*, and *Global* are given as follows

Decl. 6:

```
function fieldOf(_,_)

constructor Global

derived function val(n) == n.fieldOf(Global)
```

Through the redefinition of val as field of the constant *Global* we can reuse the existing *use* and *asgnStm* Montages (Figures 72 and 73) without any change. During the whole specification process we found that there are many instances of exact reuse in Montages, and therefore we have neglected more advanced reuse features such as inheritance.

To enable exact reuse in later languages we introduce now two equivalent, refined definitions of *use* and *asgnStm*. The new definition of the use-Montage (Figure 74) is semantically equivalent to the old one, but defines explicitly two attributes *lObject* and *lField*. These attributes serve as *interface* for accessing *left values* of expressions. The right-value is given by the already given definition of the val-attribute.

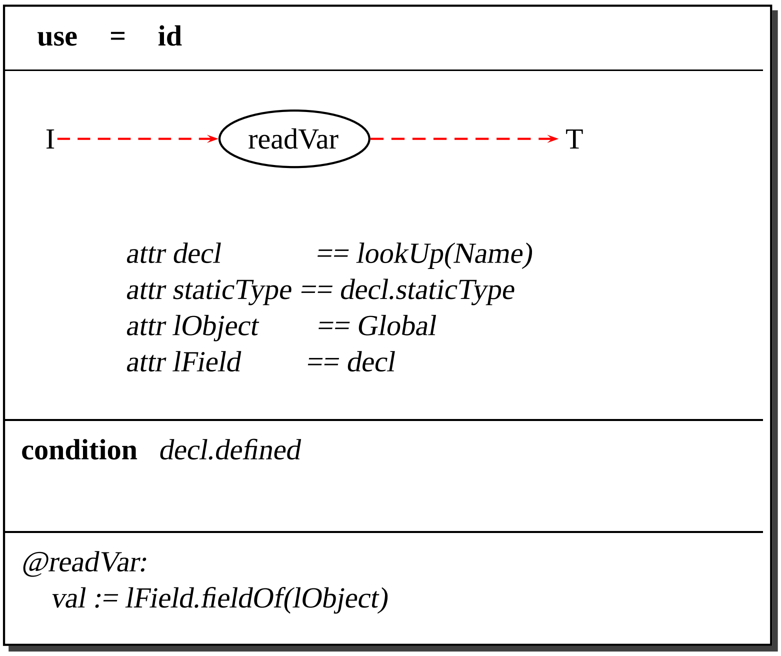
The refined specification of the assignment Montage is given in Figure 75. This version of the assignment works with arbitrary complex use-expressions on the left, as long as the evaluation of this expression results in defining its *lObject* and *lField* attributes. The action

ASM 64:

```
let o = S-use.lObject
    f = S-use.lField
in
  f.fieldOf(o) := S-exp.val
endlet
```

is then generically working for assignments to global variables, local variables, and instance variables. In the feature roaster of Figure 68 we see that the refined versions of *use* and *asgnStm* are reused as they are in all remaining languages, with exception of *ObjV2*, which is a successor of *ObjV1*, but not a predecessor for the other languages.

In the next two Chapters we show other applications of the *Field-Of-Object-Mapping* pattern, one for modeling classes, instances, and instance fields, and one for modeling procedures, recursive-calls, parameters, and variables.



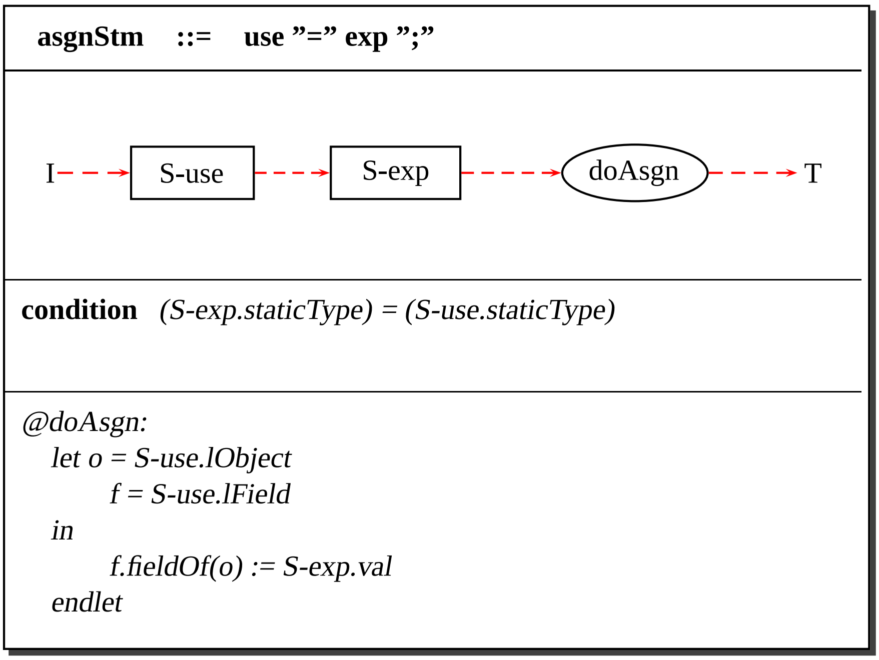
use = id

attr decl == lookUp(Name)
attr staticType == decl.staticType
attr lObject == Global
attr lField == decl

condition *decl.defined*

```
@readVar:
  val := lField.fieldOf(lObject)
```

Fig. 74: Montage *use* of language *ObjV1*



asgnStm ::= use "=" exp ";"

condition *(S-exp.staticType) = (S-use.staticType)*

```
@doAsgn:
  let o = S-use.lObject
      f = S-use.lField
  in
      f.fieldOf(o) := S-exp.val
  endlet
```

Fig. 75: Montage *asgnStm* of language *ObjV1*

12

Classes, Instances, Instance Fields

In this chapter we present the language *ObjV2* a simple “object oriented” language, featuring classes, inheritance, instance-fields, and their dynamic binding. Many main-stream languages like Java feature only dynamic binding of methods, and instance-fields are statically bound; our choice to present a language without methods but dynamically bound instance fields allows us to present key features of object oriented languages in a minimal setting.

To specify *ObjV2* we extend *ObjV1*, by refining two out of 17 existing Montages and adding six new Montages. Four Montages are introduced to build the syntax for class and field declaration, two to define the new kind of types. The *use*- and a *asgnStm*-Montages are refined in order to take into consideration the differences of variable and field accesses and updates. Finally we define two new expressions, the *newExp* for creating objects, and a *cast* for casting the dynamic type of an object, in order to allow access to the overridden fields of its super-classes.

The grammar of *ObjV2* is given as follows.

Gram. 15: (refines Grammar 14)

Program	::=	{ *classDeclaration* } *body*
classDeclaration	::=	*“class” id* [*“extends” superId*]
		“{*”* { *fieldDeclaration* } *“*}*”*
superId	=	*id*
fieldDeclaration	::=	*type id “;”*
type	=	*primitiveType* | *typeRef*
primitiveType	=	*“int”* | *“boolean”*
typeRef	=	*id*
...		
exp	=	... | *newExp* | *cast*

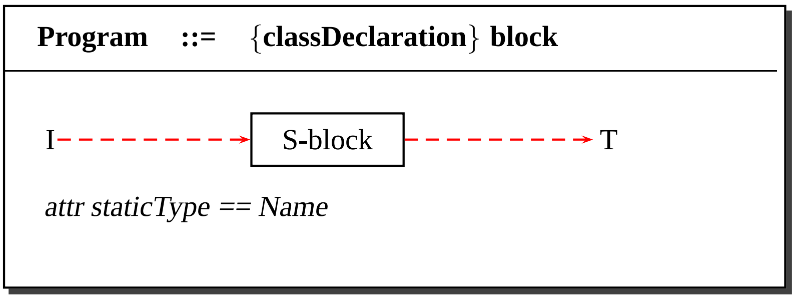
i: introduced	**r:** refined u: used	ExpV1	ImpV1	ImpV2	ImpV3	ObjV1	ObjV2	ObjV3	FraV1	FraV2	FraV3
Concept	Description										
Program	start symbol of each language	**i**	**r**	u	u	u	**r**	**r**	**r**	u	**r**
exp	synonym for expressions	**i**	**r**	**r**	u	u	**r**	**r**	**r**	u	**r**
use	the use expression			**i**	**r**	u	u	u	u	u	u
lit, Number, Boolean, uExp, bExp, cExp											
stm	synonym for statements		**i**	**r**	**r**	u	u	**r**	**r**	**r**	**r**
printStm, ifStm, asgnStm, body, block, bstm, var											
type	types of the language				**i**	u	**r**	**r**	u	u	u
classDeclaration	declaration of OO class						**i**				
fieldDeclaration	declaration of object fields						**i**				
primitiveType	synonym of primitive types						**i**				
typeRef	references to class types						**i**				
newExp	expression for creation of new obj.						**i**				
cast	casting of dynamic object type						**i**				

Fig. 76: Roaster of ObjV2 features and their introduction (i), refinement (r), and use (u) in the different languages

newExp ::= *"new" typeRef*
cast ::= *"cast" "(" typeRef "," use ")"*

12.1 ObjV2 Programs

The start symbol *Program* of *ObjV2* is given in Figure 77. The attribute *declTable(_)* of this Montage maps identifiers to the corresponding class declaration node, which are modeling the classes. The control enters directly the block of statements, the list of class declarations needs not to be visited.



Program ::= {classDeclaration} block

attr staticType == Name

Fig. 77: Montage *Program* of language *ObjV2*

The possible scopes used by *lookUp* and *enclosing* are now including *Program* in addition to the *block*. Therefore Declaration 5 is refined as follows.

Decl. 7:
```
derived function Scope == {``block'', ``Program''}
```

12.2 Primitive and Reference Type

In language *ImpV3* we introduced built-in types, namely integers and booleans. We defined the attribute *staticType* for expressions and introduced simple type checks. In object-oriented languages the definition of classes or *reference types*, allows the user to introduce new types. The existing built in types are called *primitive types*, since the values of these types have no internal structure. In *ObjV2* there exist the primitive, built-in types integer and boolean, and the user-defined classes.

The existence of different kind of types rises the question how they can be treated in a uniform way, in order to make type checking and variable declarations simple. In *ObjV2* we model all types as elements, the primitive types are represented by the string-values corresponding to their name, and the reference types are represented by their declaration-node in the syntax tree. The *type*-production has two synonyms, *primitiveType* as specified in Figure 78, and *typeRef* as specified in Figure 79. The attribute *staticType* of the first points to the name of the primitive types, and the *staticType* definition of the second points to the corresponding class-declaration, which is retrieved using the *lookUp* function.

```
primitiveType   =   "int" |"boolean"
------------------------------------
    attr staticType == Name
```

Fig. 78: Montage *primitiveType* of language *ObjV2*

```
typeRef   =   id
------------------------------------
  attr signature    == Name
  attr staticType == lookUp(signature)
------------------------------------
condition   staticType.defined AND (staticType.classDeclaration)
```

Fig. 79: Montage *typeRef* of language *ObjV2*

Type references are specified in Figure 79. Their static semantics guarantees that their *staticType* attribute refers to a class declaration. The definition of *staticType* of type references uses the *lookUp* function introduced earlier.

```
classDeclaration   ::=   "class" id ["extends" superId]
                         "{"
                         {fieldDeclaration}
                         "}"
 superId            =    typeRef
-----------------------------------------------------------
   attr signature == S-id.Name
   attr superType == S-superId.staticType
   attr declTable(n) ==
      (choose f in sequence S-fieldDeclaration:
          f.signature = n)
   attr fieldTable(n) ==
      (if declTable(n).defined then declTable(n)
       else (if superType = undef then undef
                else superType.fieldTable(n)))
```

Fig. 80: Montage *classDeclaration* of language *ObjV2*

12.3 Classes and Subtyping

Classes are specified in Figure 80. The first component of a class is an identifier, denoting the name of the class. This name is accessible as attribute *signature*. The second component is an optional type reference to the super type of the class. The attribute *superType* of a class refers directly to the static type of the type reference to the super type, e.g. to the class declaration of the super type.

Again based on the definition of the attribute *superType*, we can now define the sub-typing relation *subtypeOf(_,_)*.j The term *subtypeOf(a,b)* or *a.subtypeOf(b)* evaluates to *true* if either a and b are equal, or if *a.superType* is defined and this super type is a subtype of the second argument.

Decl. 8:

```
derived function subtypeOf(t1, t2) ==
  (t1 = t2) OR
      (t1.superType.defined AND
       t1.superType.subtypeOf(t2))
```

Finally, the last component of a class is a list of field declarations. Each field, as specified in Figure 81 has two attributes, the *signature* attribute referring to the field's name, and the *staticType* attribute referring to the field's type. Coming back to the class declaration, there are two attributes to refer to the fields, both taking the field-name as argument. The first, *declTable(_)*, returns a field-declaration node from the class's list of field-declarations, if one of these declarations matches a given field-name, otherwise it returns *undef*.

The attribute *fieldTable(_)* is collecting field declarations from the class and its super-classes. It tries to find a field-declaration using the previously defined *declTable*. If there is no field found in the *declTable* of the class itself, the field table of the super-type is evaluated, if a super-type exists. Otherwise *undef* is returned.

fieldDeclaration ::= type id ";"

attr signature == S-id.Name
attr staticType == S-type.staticType

Fig. 81: Montage *fieldDeclaration* of language *ObjV2*

12.4 Object Creation and Dynamic Types

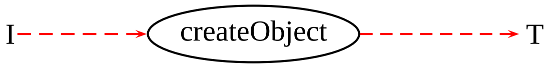
As mentioned earlier, we model objects as ASM-elements. A universe *ObjectID(_)* of all elements being objects in the specified language is introduced, and a dynamic function *dynamicType(_)* is used to keep track of the type of the created objects.

Decl. 9:

```
univers ObjectID
function dynamicType(_)
```

In the *newExp*-Montage (Figure 82) the specification of the object creation construct is given. The "createObject"-action creates a new member *o* of the *ObjectID* universe, sets the dynamic type of *o* to the static type of the new-clause, and sets the *val*-attribute of the new-clause to *o*.

newExp ::= "new" typeRef



attr staticType == S-typeRef.staticType

```
@createObject:
  extend ObjectID with o
    o.dynamicType := staticType
    val := o
  endextend
```

Fig. 82: Montage *newExp* of language *ObjV2*

12.5 Instance Fields

The instance fields of objects in *ObjV2* are modeled as the field-declarator-nodes, being linked to the dynamic type of the object via the *fieldTable* attribute.

The values of such fields are modeled using the dynamic function *fieldOf(_, _)*. Once the field-declarator node *lField* is known of an object *lObject*, the value of that field is read as the following expression

$$lField.fieldOf(lObject)$$

and it is set to a new value v by the following update

```
  lField.fieldOf(lObject) := v
```

12.6 Dynamic Binding

Which field of an object is read or written is determined dynamically, depending on the dynamic type of an object o, being determined by expression *o.dynamicType*. Given field-name f, and object o, the field is determined by

$$o.dynamicType.fieldTable(f)$$

In the following each Montage of an assignable expression, e.g. an expression that can be on the left-hand-side of an assignment, has attribute definitions *lObject*, denoting the so called *left object*, and *lField*, the *left field*. Assigning a value v to such an assignable expression e is done by

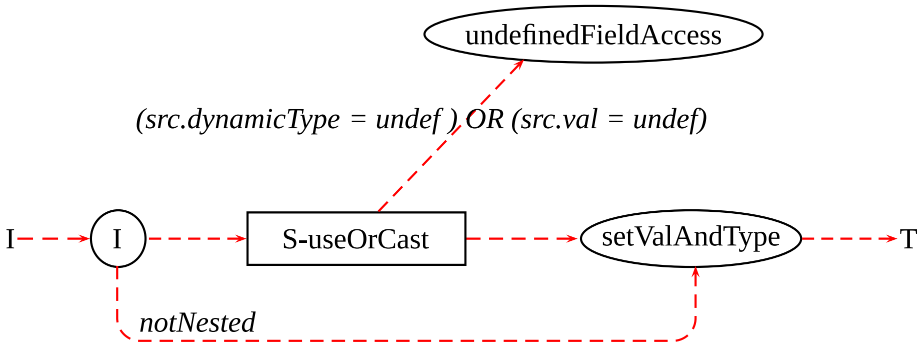
$$e.lField.fieldOf(e.lObject) := v$$

The use construct
The specification of the *use*-construct, which serves for variable uses and field accesses and as left part of assignments, is complicated since it covers both simple variable accesses and the above sketched accesses to object fields. The complete specification is given in Figure 83. To simplify the explanations, we deduce by partial evaluation two specialized versions of the use-Montages, one for simple variable accesses and one for instance-field accesses.

In the case of a simple variable use, the *useOrCast*-component and the "." are not present and the attribute *notNested* evaluates to *true*. The specialized Montage for this case is called *useVar* and is given in Figure 84. Control flows directly to the *setValAndType* action. This action sets the *val*-attribute to the value of the referenced variable. The value of variables is stored as a field of left-object *Global*, and the left-field is looked up by *lookUp(signature)*. The action uses the term *lField.fieldOf(lObject)* to read the value of the variable.

The case of field access is visualized by the Montage *useField* in Figure 85, which is again obtained by specializing the *use*-Montage. The attribute *nestedUse* points directly to the *useOrCast*-component, which is always present in this case. Control flows first into the *useOrCast* component, being either again a use, or alternatively a cast. If after the evaluation of this component either it's dynamic type is undefined, or there results no value, control flows into the

use ::= [useOrCast ”.”] id

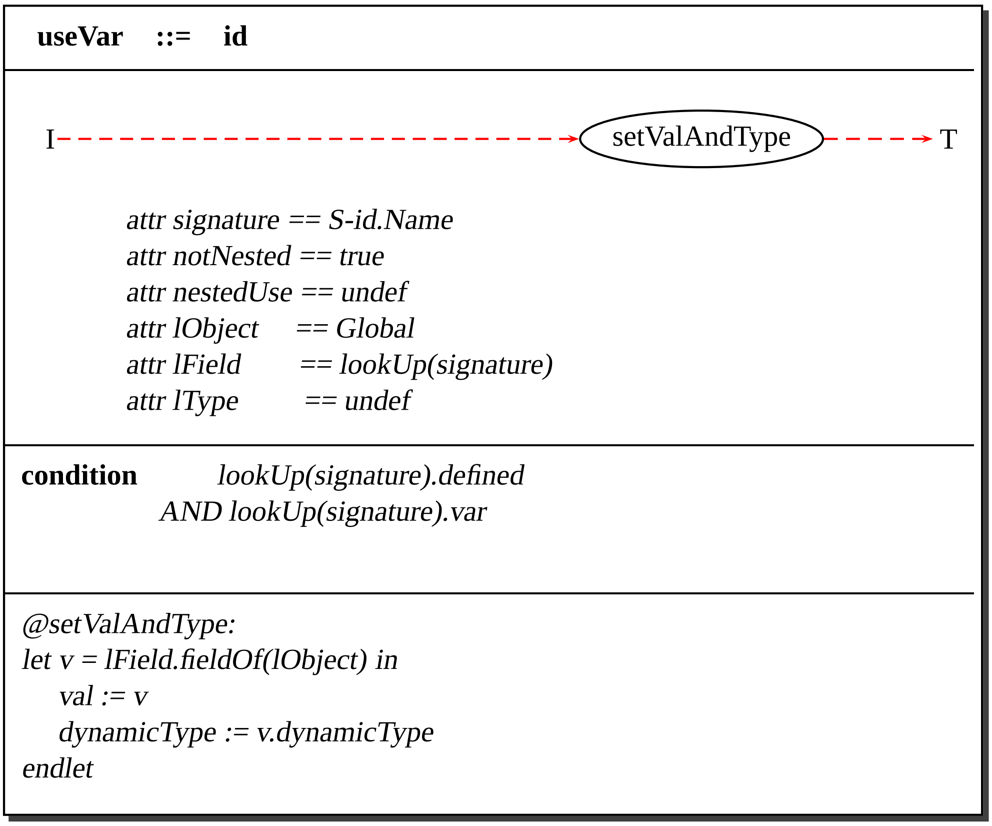


```
attr signature == S-id.Name
attr notNested == S-useOrCast.NoNode
attr nestedUse == S-useOrCast
attr lObject    ==
(if notNested then   Global
                     else   nestedUse.val)
attr lField     ==
(if notNested then   lookUp(signature)
                     else   lType.fieldTable(signature))
attr lType      == nestedUse.dynamicType
```

```
condition   (if notNested then       lookUp(signature).defined
                                    AND lookUp(signature).var
             else true)
```

```
@setValAndType:
let v = lField.fieldOf(lObject) in
    val := v
    if not notNested then
        dynamicType := v.dynamicType
    endif
endlet

@undefinedFieldAccess:
   handleError(”Access of undefined field.”)
```

Fig. 83: Montage *use* of language *ObjV2*

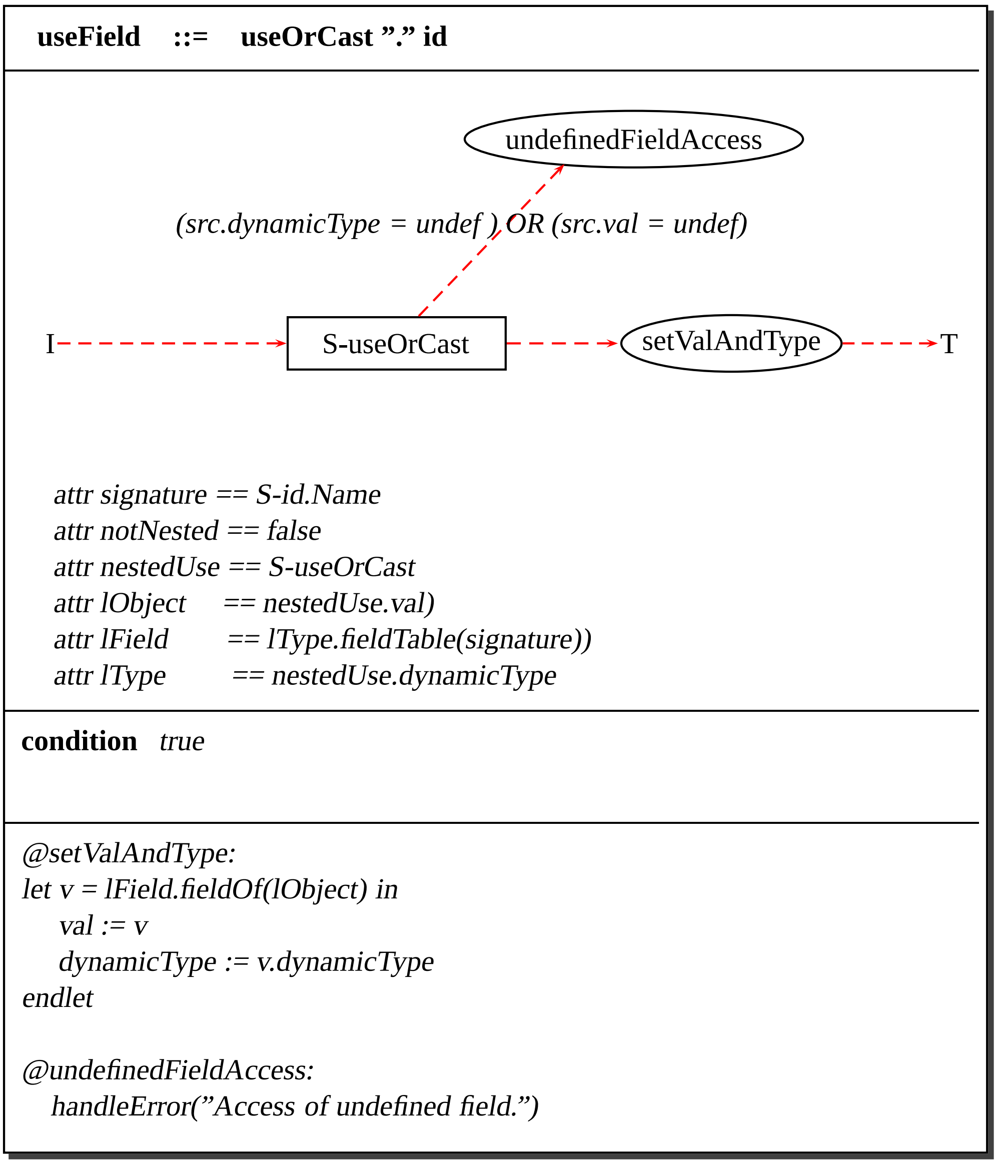


```
attr signature == S-id.Name
attr notNested == true
attr nestedUse == undef
attr lObject   == Global
attr lField    == lookUp(signature)
attr lType     == undef
```

```
condition   lookUp(signature).defined
        AND lookUp(signature).var
```

```
@setValAndType:
let v = lField.fieldOf(lObject) in
   val := v
   dynamicType := v.dynamicType
endlet
```

Fig. 84: Montage *useVar* of language *ObjV2*



useField ::= useOrCast "." id

```
attr signature == S-id.Name
attr notNested == false
attr nestedUse == S-useOrCast
attr lObject    == nestedUse.val)
attr lField     == lType.fieldTable(signature))
attr lType      == nestedUse.dynamicType
```

condition *true*

```
@setValAndType:
let v = lField.fieldOf(lObject) in
    val := v
    dynamicType := v.dynamicType
endlet

@undefinedFieldAccess:
   handleError("Access of undefined field.")
```

Fig. 85: Montage *useField* of language *ObjV2*

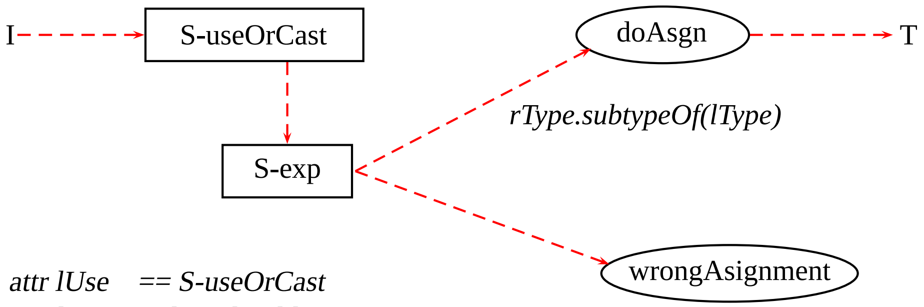
undefinedFieldAccess-action, triggering a "Access of undefined field"-error. We do not further specify how such an error is handled. If the error does not occur control flows into the *setValAndType*-action. *lObject* is the object, and *lField* the field to be accessed. As mentioned at the beginning of this section, *lField* is looked up in the field table of the dynamic type of the accessed object. To increase readability, the attribute *lType* is introduced, denoting the above used dynamic type of the field access.

The assignment statement

The *asgnStm*-Montage is given in Figure 86. As we can see, it is not needed to differentiate between variable and field use in this Montage. Further it is possible to assign values both to the above described *use* Montages, respectively it's special cases *useVar* and *useField*, and the later described *cast* Montage. This property is achieved by using the definitions of *lObject* and *lField* attributes as interface for left values, as discussed earlier in Section 11.3.

First control flows through the *exp*-component, resulting in the evaluation of its *val*-attribute, and then into the use or cast component, resulting in the evaluation of their *lObject* and *lField* attributes. Then the assignment is done in action *doAsgn*, or, if the types of left and right side are not assignable, the control flows to action *wrongAssignment*. The exact definition for assignability in *ObjV2* is that the dynamic type of the expression is assignable to the static type of the field or variable we are assigning to. In detail, the field or variable to which we assign is *lUse.lField*, thus the type of the left side, *lType* is defined as *lUse.lField.staticType*. The attribute *rType* denotes the dynamic type of the *exp*-component, if defined, otherwise the static type. The condition for a correct assignment is that all instances of the *rType* are instances of the *lType*, with other words, the *rType* must be a subtype of the *lType*. This condition is given as label of the control-arrow from the "S-exp"-box to the "doAsgn"-oval. In the case of correct dynamic types, the same action as in the *ObjV1* version of *asgnStm* (Figure 75) is triggered, assigning to the *lField* of the *lObject* of the left-hand-side the value of the right-hand-side expression.

asgnStm ::= useOrCast ”=” exp ”;”



```
attr lUse    == S-useOrCast
attr lType == lUse.lField.staticType
attr rType ==
(if S-exp.dynamicType.defined then
            S-exp.dynamicType
   else S-exp.staticType)
```

```
@doAsgn:
   let o = lUse.lObject
         f = lUse.lField
   in
      f.fieldOf(o) := S-exp.val
   endlet

@wrongAsignment:
   handleError(
”This asignment is not valid, due to ”+
”dynamic type missmatch.”)
```

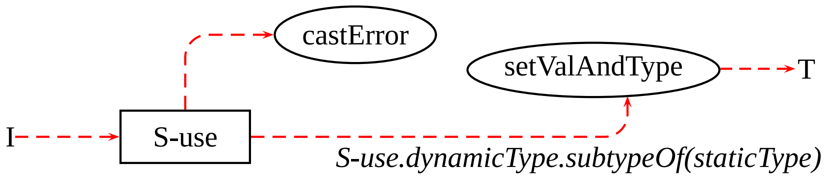
Fig. 86: Montage *asgnStm* of language *ObjV2*

12.7 Type Casting

With the type-casting expression it is possible to change the dynamic type of an object to one of it's super-types. This is needed for instance, if a field of a subtype hides the definition of a field of a super-type. Hiding in this sense happens if the names of these fields are equal. Using the cast expression, the hidden field of the super-type can be read or written.

The specification of the cast-expression is given in Figure 87. The dynamic type check in this Montage ensures, that no field accesses happen on null objects, and that assignments are type correct with respect to the static type of the variable of the field which one is assigning to. The values of attributes *lObject* and *lField* are copied from the corresponding fields of the *use*-component.

cast ::= ”cast” ”(” typeRef ”,” use ”)”



```
attr staticType   == S-typeRef.staticTyp
attr lObject      == S-use.lObject
attr lField       == S-use.lField
```

```
@setValAndType:
   val           := S-use.val
   dynamicType := staticType

@castError:
   handleError(”CastError”)
```

Fig. 87: Montage *cast* of language *ObjV2*

13

Procedures, Recursive-Calls, Parameters, Variables

In this chapter we introduce example language *ObjV3*, featuring function calls, recursion, and call-by-value parameters, as well as local variables. The language is defined by extending and refining the definitions of *ObjV1* (Section 11.3), the grammar is given as follows.

Gram. 16: (refines Grammar 14)

Program	::=	{ *functionDecl* } *block*
exp	=	... | *call*
stm	=	... | *returnStm*
functionDecl	::=	*“function” id “(” { var } “)”*
		“:” type body
call	::=	*id “(” [actualParam*
		{ *“,” actualParam* } *] “)”*
actualParam	=	*exp*
returnStm	::=	*“return” exp “;”*

13.1 ObjV3 Programs

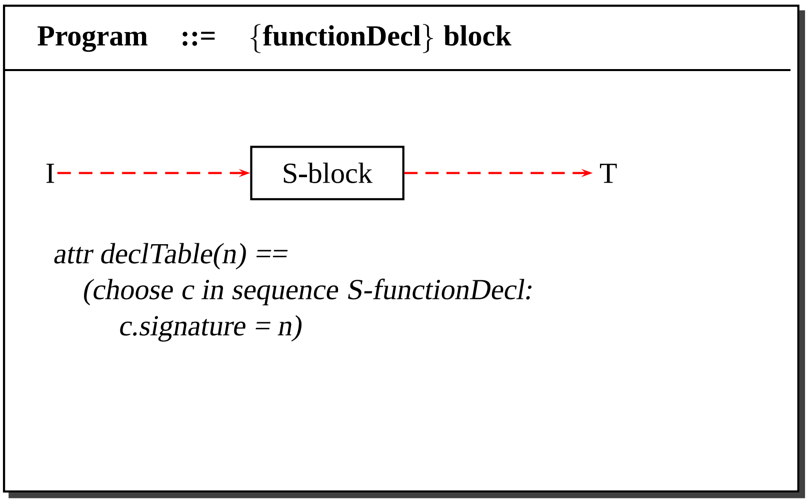
The start-symbol of the grammar, *Program*, produces a list of function declarations and a block. The execution of an *ObjV3* program is done by executing the block. This behavior is given in Figure 89. The same specification defines as well the declaration table for accessing the functions, allowing to access function f from any point n in the program as

$$n.\mathit{enclosing}(\{\text{“}\mathit{Program}\text{”}\}).\mathit{declTable}(f)$$

i: introduced **r:** refined u: used

Concept	Description	ExpV1	ImpV1	ImpV2	ImpV3	ObjV1	ObjV2	ObjV3	FraV1	FraV2	FraV3
Program	start symbol of each language	**i**	**r**	u	u	u	**r**	**r**	**r**	u	**r**
exp	synonym for expressions	**i**	**r**	**r**	u	u	**r**	**r**	**r**	u	**r**
use	the use expression			**i**	**r**	u	u	u	u	u	u
lit, Number, Boolean, uExp, bExp, cExp											
stm	synonym for statements		**i**	**r**	**r**	u	u	**r**	**r**	**r**	**r**
printStm, ifStm, asgnStm, body, block, bstm, var											
type	types of the language				**i**	u	**r**	**r**	u	u	u
functionDecl	procedure dec.aration							**i**			**r**
call	procedure call							**i**			**r**
actualParam	actual parameter of call							**i**			u
returnStm	return statment							**i**			**r**

Fig. 88: Roaster of ObjV3 features and their introduction (i), refinement (r), and use (u) in the different languages


Program ::= {functionDecl} block

attr declTable(n) ==
(choose c in sequence S-functionDecl:
c.signature = n)

Fig. 89: Montage *Program* of language *ObjV3*

13.2 Call Incarnations

The semantics of the function calls is based on modeling *function-call incarnations* as elements of universe *INCARNATION*. After creation, the current call incarnation is assigned to the dynamic function *Incarnation*. The new current incarnation is linked to the previous one by the dynamic function *lastInc*.

Decl. 10:

```
universe INCARNATION
function Incarnation
function lastInc(_)
```

The most simple semantics based on this model calls a function by executing

```
extend INCARNATION with i
  i.lastInc   := Incarnation
  Incarnation := i
endextend
```

and returns from the call by restoring the old value of *Incarnation*.

```
Incarnation := Incarnation.lastInc
```

From these actions we omitted the details how the call-statement is found, once the called function terminated, how the parameters are passed, and how the result is returned.

Before we come to these details we continue to investigate the properties of languages with recursive calls. In contrast to languages without recursion, expressions may have different values in different function-call incarnations, and therefore, the definition of the attribute *val* is refined to the derived function

Decl. 11:

```
derived function val(n) == n.fieldOf(Incarnation)
```

which stores and retrieves values of a program-expression e as the value of field n of object *Incarnation*, where n is the AST-node representing e, and *Incarnation* is the previously introduced current incarnation. Like this expressions have distinct values in distinct function-call incarnations and at the same time, the old *val* syntax can be used to calculate expressions within the current incarnation.

On the other hand, the val-attribute cannot be used to pass information from one function-call incarnation to the next one, e.g. for passing formal parameter and returning call-results. This will be done by using a simple variable *RESULT* which is just a 0-ary dynamic ASM function.

13.3 Semantics of Call and Return

As mentioned there are two points when information must be passed across incarnations, once when call is triggered and the formal parameters of the function declaration must be actualized, and once when the result of the terminating call is returned.

For passing information from one incarnation to another we use a simple 0-ary dynamic function called *RESULT*. The RESULT function is used in the current example language and in following languages whenever information is passed along the control flow.

In the Montage for the *call* construct (Figure 90) we can see the action *prepareCall* which executes both the above outlined rule for creating a new call incarnation, and which sets the *RESULT* to the actual parameters. As last component the current call-node *self* is assigned to the field *ReturnPoint* of the newly created function-call incarnation.

Then the *call*-Montage sends control to a function-declaration. The control flows to the function declaration being denoted by the *decl*-attribute of the *call*-Montage. If control entered the function-declaration Montage (Figure 91) the actual parameters are passed to the formal ones, and the *RESULT*-function is reset to *undef*.

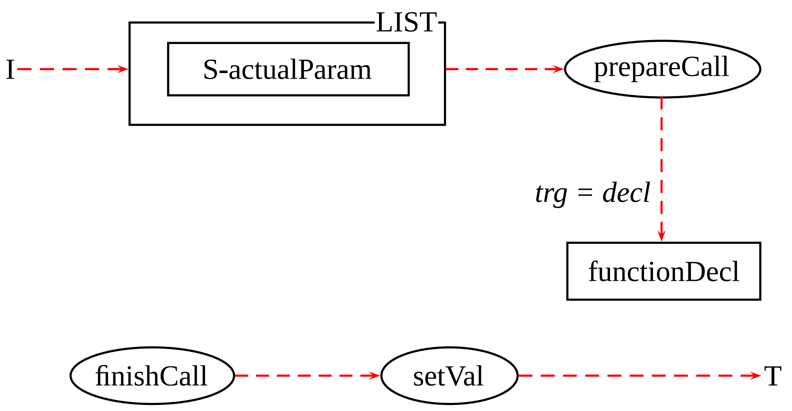
If in the body of the function declaration a return statement (Montage in Figure 92) is reached, the *RESULT*-function is set to the value of the returned expression, and control is send to the *finishCall*-action of the *call*-instance being stored in the field *ReturnPoint* of the current incarnation.

The XASM declarations for the described processes are given as follows.

Decl. 12:

```
function RESULT
constructor ReturnPoint

external function PassParameters(_,_)

derived function Scope == {"block", "functionDecl"}
```

```
call          ::=  id ”(” [actualParam {”,” actualParam}] ”)”
actualParam    =   exp
```

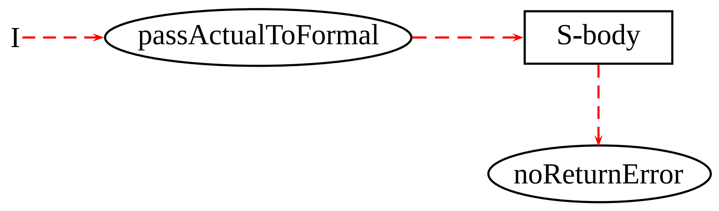


```
attr signature == S-id.Name
attr decl == enclosing({”Program”}).declTable(signature)
attr staticType == decl.staticType
```

```
@prepareCall:
  RESULT :=   S-actualParam.combineActualParams
  extend INCARNATION with i
    ReturnPoint.fieldOf(i) := self
    i.lastInc := Incarnation
    Incarnation := i
  endextend

@finishCall:
  Incarnation := Incarnation.lastInc

@setVal:
    val := RESULT
    RESULT := undef
```

Fig. 90: Montage *call* of language *ObjV3*

functionDecl ::= ”function” id ”(” {var} ”)” ”:” type body



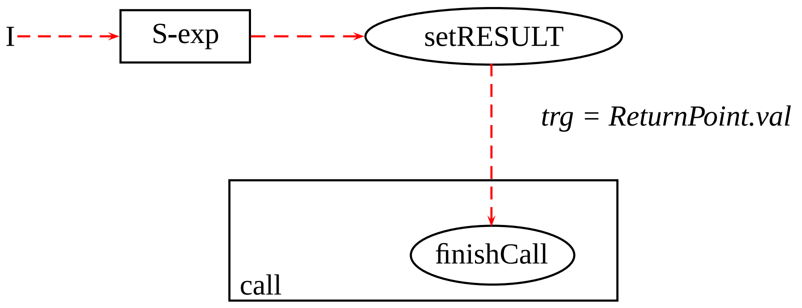
attr staticType == S-type.staticType
attr signature == S-id.Name
attr declTable(pStr) == (choose p in sequence S-var :
p.signature = pStr)

@passActualToFormal:
val := PassParameters(RESULT, S-var)
RESULT := undef

@noReturnError:
handleError(”Exiting without return error”)

Fig. 91: Montage *functionDecl* of language *ObjV3*

returnStm ::= ”return” exp ”;”



@setRESULT:
RESULT := S-exp.val

Fig. 92: Montage *returnStm* of language *ObjV3*

13.4 Actualizing Formal Parameters

Before we can pass the actual parameters via the *RESULT* function, we need to transform the list of expressions of the call-syntax into the list of the actual values of these expressions. This is done by the following derived function.

Decl. 13:

```
derived function combineActualParams(al) ==
  (if al =~ [&hd | &tl] then
    [&hd.val | &tl.combineActualParams]
   else
    al
  )
```

In the prepare-action of the call-Montage the resulting list of assigned to the RESULT function, and the new incarnation is created. Then control flows into the corresponding functionDecl-node where the list is retrieved from RESULT and passed together with the list *S-Var* of formal parameter declarations to the ASM *PassParameters* which is given in the following. The algorithm traverses the list of values and the list of parameter declarations in parallel and sets the *val* attribute of each parameter in the second list to the corresponding value in the first list.

ASM 65: PassParameters.xasm

```
asm PassParameters(a, f)
-- a is sequence of values, f sequence of parameter instances
 updates function val(_)
is
 function a0 <- a, f0 <- f
if  a0 =~ [&ahd | &atl] then
   if f0 =~ [&fhd | &ftl] then
      &fhd.val := &ahd
      a0 := &atl
      f0 := &ftl
   else return "length mismatch of actual and formal parameters"
   endif
else
   return true
endif
endasm
-- a is sequence of values, f sequence of parameter instances
```

14

Models of Abrupt Control

In this chapter we introduce example languages *FraV1* (Section 14.2), *FraV2* (Section 14.3), and *FraV3* (Section 14.4) featuring iteration constructs, exception handline, and a revised version of recursive function calls. All of this languages use the concept of frames which is explained in the next section.

A main result of this thesis is the fact the here presented specifications are compositional and provide the same degree of modularity for abrupt control flow features as the normal Montages transitions provide for sequential, regular control flow.

14.1 The Concept of Frames

For the specification of *FraV1*, and its relatives *FraV2* (exception handling) and *FraV2* (procedure calls) we use the frame-result-controlflow or short *frame pattern* introduced in the introduction to Part III for modeling abrupt control flow. Abrupt control flow is a term for all kind of non-sequential control flow such as breaking out of a loop, throwing an exception, or calling a procedure. A frame is a node in the syntax tree which is relevant for abrupt control.

By defining the set of universe names *Frame* to contain all symbols relevant to abrupt flow, we can jump to the least enclosing frame using the earlier introduced *enclosing* function. The information relevant for controlling abrupt control flow is passed via the *RESULT* function, and each frame has an action *frameHandler* which handles the information, if it is relevant for the frame, and otherwise passes the information further up to the next enclosing frame.

In Figure 93 an abstract Montage *framePattern* visualizes the principle how abrupt control flow is specified with frames. The normal, sequential control flow enters the Montage at the I-edge, and triggers normal processing of the components of the Montage, such as the abstract *body* component, and then leaves the Montages via the T-edge.

Within the body, control follows the sequential transitions, until a statement initiating abrupt control is reached. As an abstract example we show the *abruptPattern*-Montage in Figure 94. The *setRESULT*-action of this Montage updates *RESULT* with the information needed to control the abrupt control, and then sends control to the *FrameHandler*-action of the least enclosing frame, leading us back to Figure 93.

From the reached *FrameHandler*-state depart two transition. The first is followed if the *RESULT is relevant* and can be processed by this Montage[1] In this case, the abrupt processing is done, *RESULT* is reset to *undef*, and control is led back into the regular sequential flow. If the *RESULT* is not relevant for the Montage, the control is sent further up to the *FrameHandler*-action of the next enclosing frame.

Since this pattern works for all kinds of abrupt control flow and a certain frame can pass arbitrary information to the next enclosing frame, such definitions are compositional and allow the same degree of modularity for abrupt control flow as the normal transitions do for sequential control flow. In Appendix C a non-compositional model of abrupt control flow is shown.

In the following frames are applied to iteration constructs, where the instances of the abrupt pattern are continue and break statements, and where the instances of the frame pattern are the different kinds of loops and the labeled statement. In the next chapter we show exception handling, where the abrupt pattern is used for the throw statement, and the frame pattern is used for try, catch, and finally clauses. As a third example in Chapter 14.4 we reformulate recursive calls using the abrupt pattern for the return-statement and the frame

[1] As an example, an exception would be a relevant result to the frame handler of an exception-construct, but a continue would not be a relevant result for the same construct.

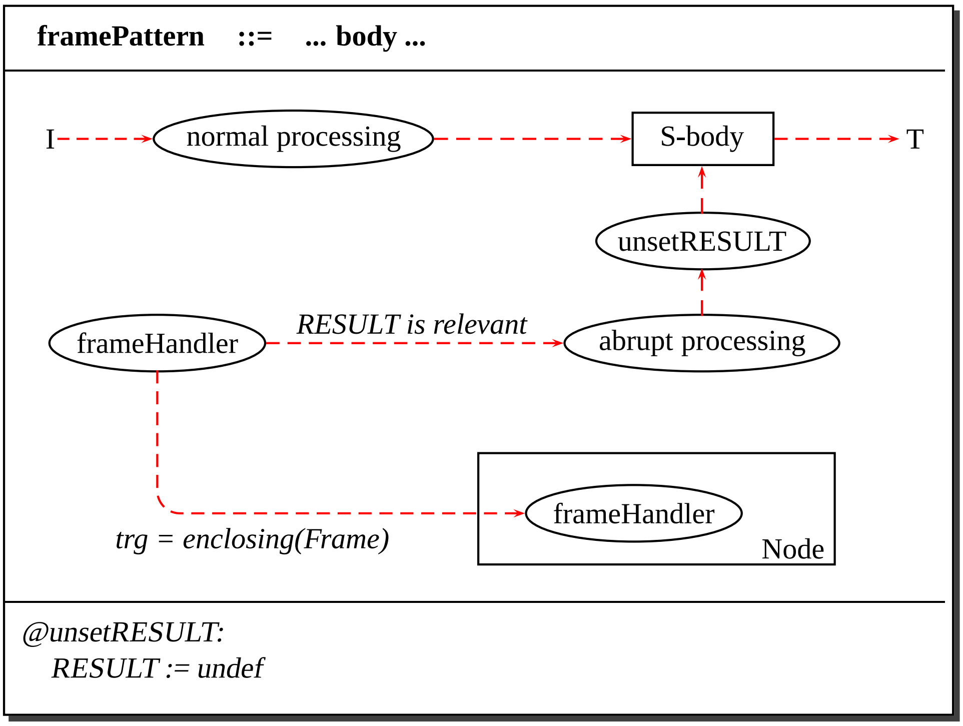


Fig. 93: Montage *framePattern* of language *FraV1*

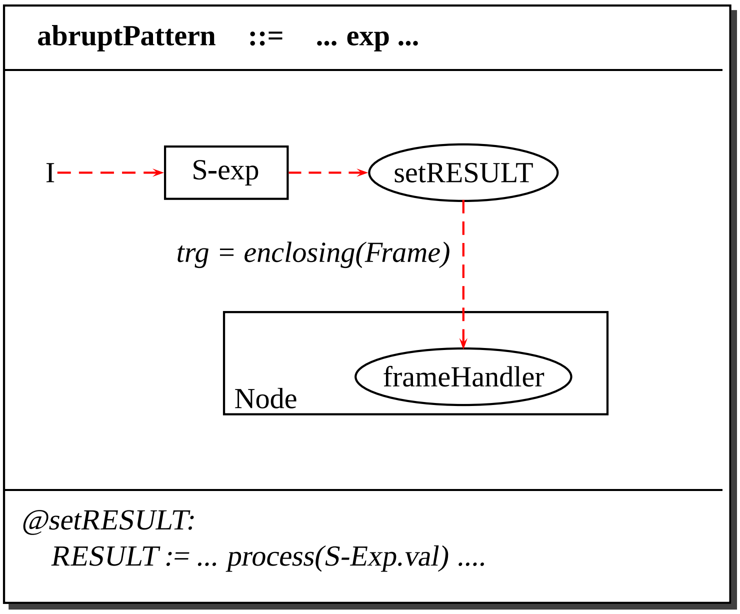

abruptPattern ::= ... exp ...
I
S-exp
setRESULT
trg = enclosing(Frame)
Node
frameHandler
@setRESULT:
RESULT := ... process(S-Exp.val) ....

Fig. 94: Montage *abruptPattern* of language *FraV1*

pattern for the function call and declaration.

14.2 FraV1: Models of Iteration Constructs

FraV1 features while, repeat, continue, break, and labeled statements. *FraV1* extends the earlier while example in Section 3.1 and the control statements of Section 10.2 with *continue* and *break* mechanisms. A first model for reaching targets of break and continue statement directly has already been shown in Section 5.3.3. In contrast, the here presented model uses the frame-pattern and is compositional with other kinds of abrupt control flow.

The grammar of FraV1 is defined as extension and refinement of the *ObjV1* grammar.

Gram. 17: (refines Grammar 14)

```
stm             =    ... | continueStm | breakStm |
                     iterationStm | labeledStm
iterationStm    =    whileStm | doStm
continueStm     ::=  “continue” [ labelId ] “;”
breakStm        ::=  “break” [ labelId ] “;”
labelId         =    id
whileStm        ::=  “while” exp body
doStm           ::=  “do” body “while” exp “;”
labeledStm      ::=  labelId “:” iterationStm
```

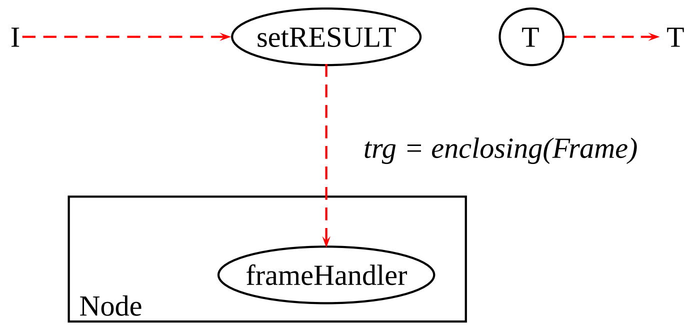
The exact definition of the *Frame* constant together with the declaration of break and continue constructors is given as follows.

Decl. 14:

```
derived function Frame ==
  {"whileStm", "doStm", "labeledStm"}
constructors break(_), continue(_)
```

The Montages of *FraV1* are mostly direct instantiations of the abrupt and frame patterns explained above. The labeled break and continue (Figures 95 and 96) follow the abrupt pattern and set the *RESULT* to the corresponding constructor term. If this term has the label *undef* it is catched by the while and do statements (Figures 97 and 98) which are both following the frame pattern. In both Montages we see how the frame-handler sends continue-results back inside the loop, and break-results to a program point after the loop. If the *RESULT* term has a label l, it is catched by the least enclosing instance of Montage *labeledStatement* (Figure 99), another instance of the frame pattern. This Montage analyzes at the *frameHandler*-action whether the label in the *RESULT* matches its own label. If there is a match, the labeled break/continue constructor terms are replaced by their un-labeled versions and control is send to the *frameHandler*-action of the statement after the label. The static semantics of *labeledStm* guarantees that this statement is a frame and therefore has a *frameHandler*-action. The un-labeled break and continue are then catched by a while or do, as mentioned above.

continueStm ::= ”continue” [labelId] ”;”

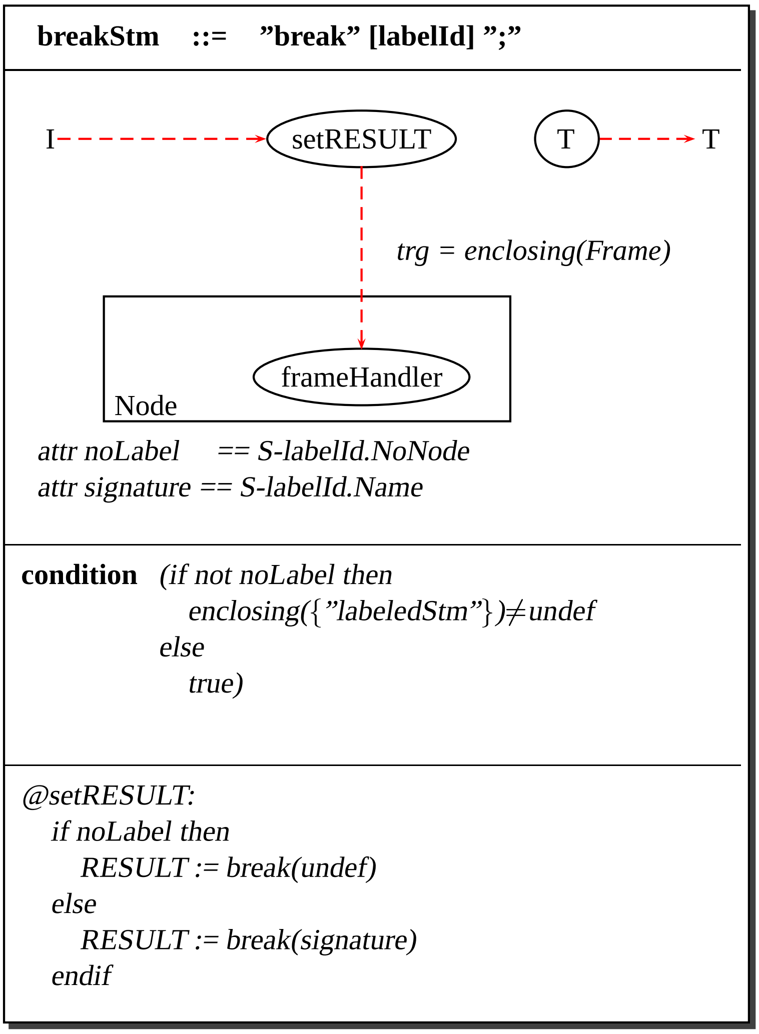


attr signature == S-labelId.Name

```
condition (if not noLabel then
              enclosing({”labeledStm”}) $\neq$ undef
          else
              true)
```

```
@setRESULT:
    if noLabel then
        RESULT := continue(undef)
    else
        RESULT := continue(signature)
    endif
```

Fig. 95: Montage *continueStm* of language *FraV1*



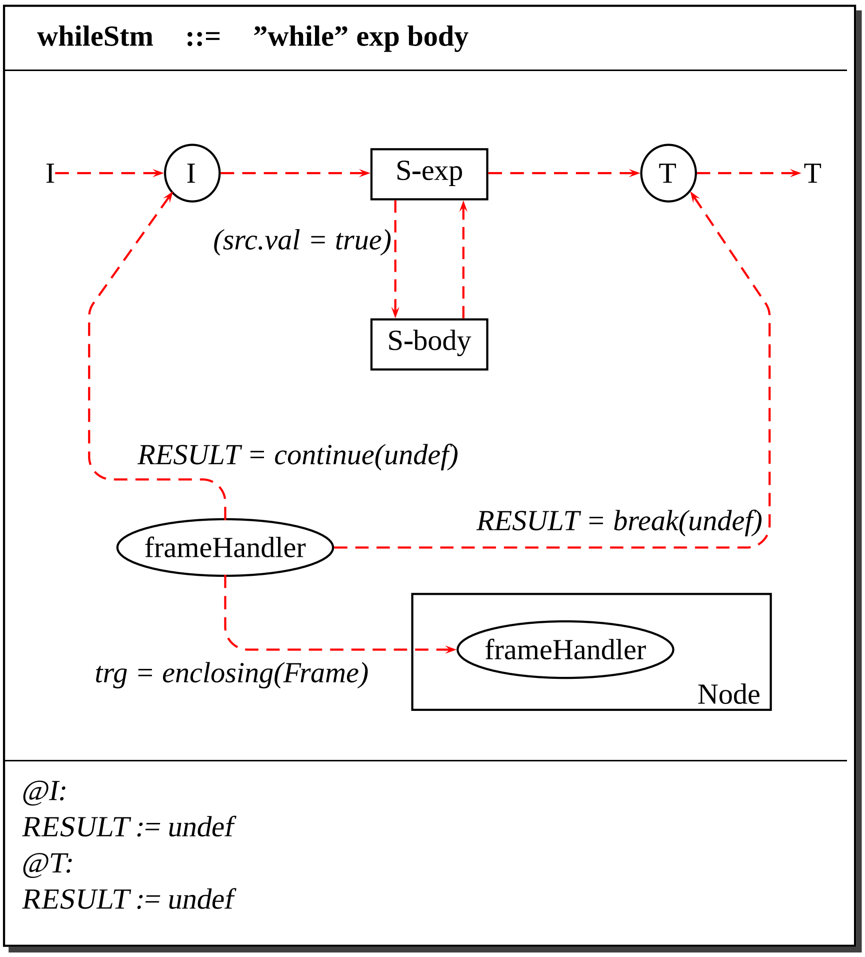
breakStm ::= ”break” [labelId] ”;”

```
attr noLabel     == S-labelId.NoNode
attr signature == S-labelId.Name
```

```
condition   (if not noLabel then
               enclosing({”labeledStm”})≠undef
            else
               true)
```

```
@setRESULT:
   if noLabel then
      RESULT := break(undef)
   else
      RESULT := break(signature)
   endif
```

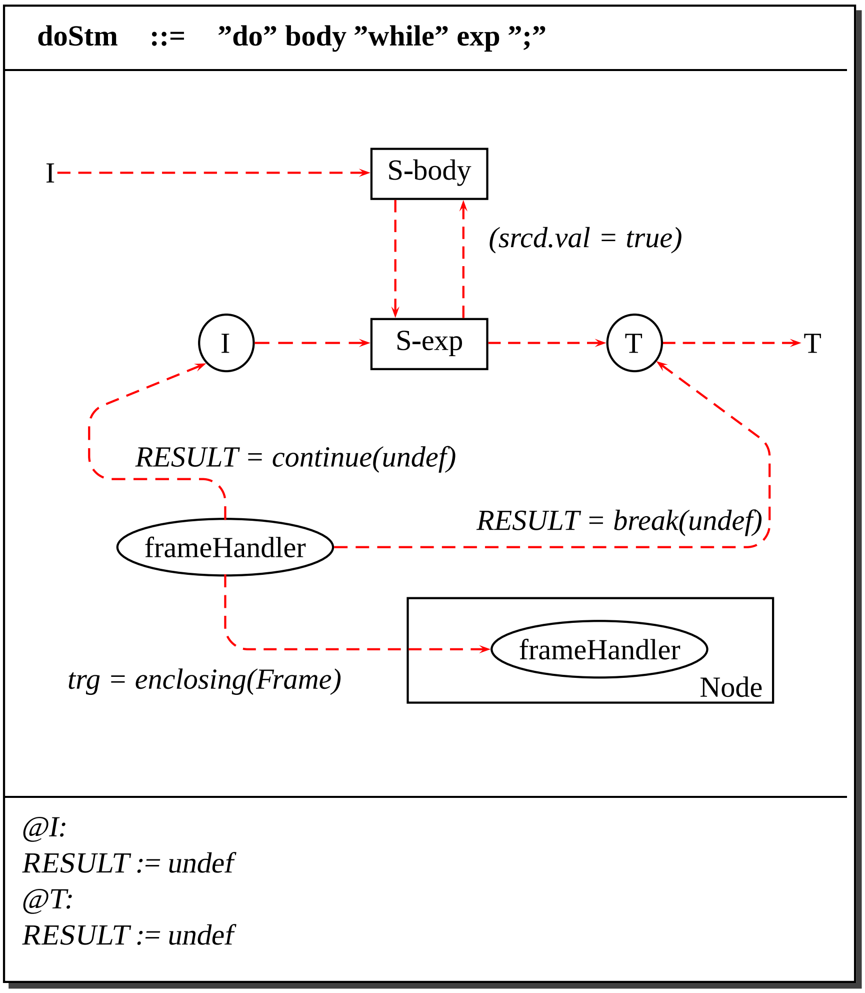
Fig. 96: Montage *breakStm* of language *FraV1*



whileStm ::= "while" exp body

@I:
RESULT := undef
@T:
RESULT := undef

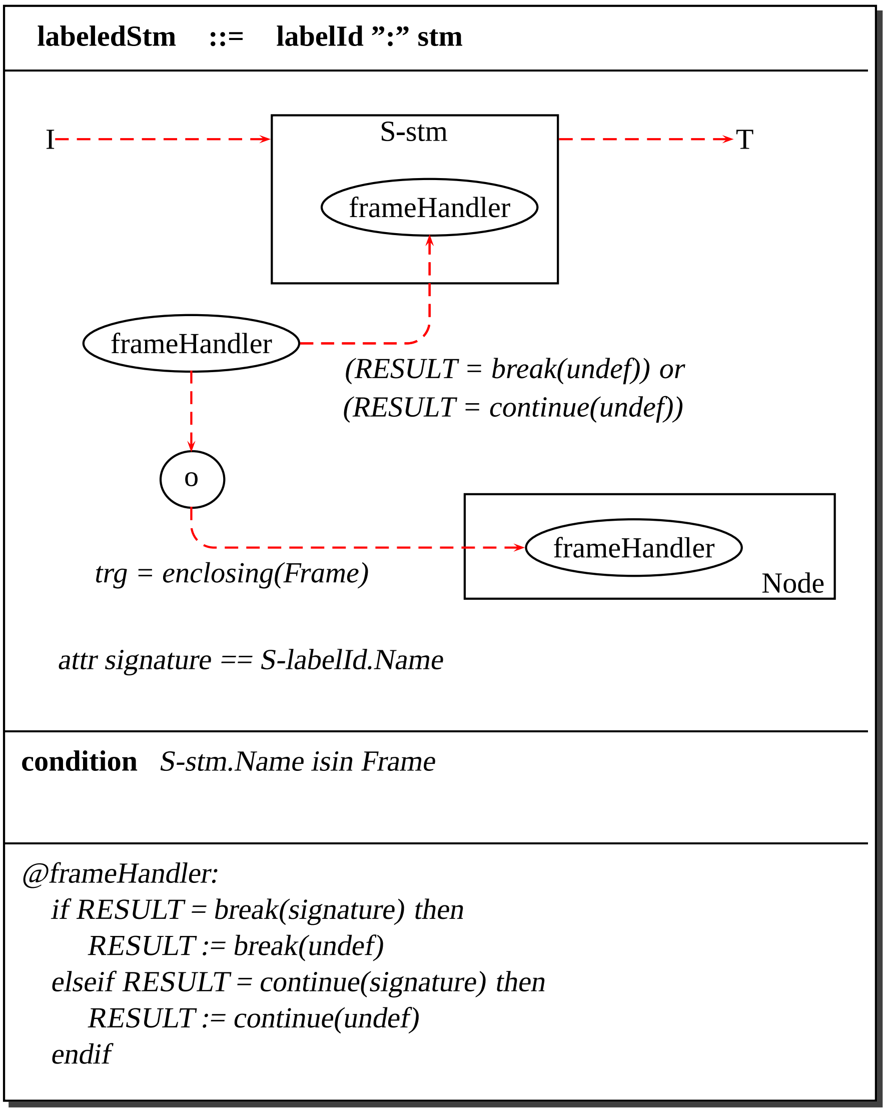
Fig. 97: Montage *whileStm* of language *FraV1*



doStm ::= ”do” body ”while” exp ”;”

@I:
RESULT := undef
@T:
RESULT := undef

Fig. 98: Montage *doStm* of language *FraV1*



labeledStm ::= labelId ”:” stm

I
S-stm
T
frameHandler
frameHandler
(RESULT = break(undef)) or
(RESULT = continue(undef))
o
trg = enclosing(Frame)
frameHandler
Node
attr signature == S-labelId.Name

condition *S-stm.Name isin Frame*

```
@frameHandler:
  if RESULT = break(signature) then
     RESULT := break(undef)
  elseif RESULT = continue(signature) then
     RESULT := continue(undef)
  endif
```

Fig. 99: Montage *labeledStm* of language *FraV1*

i: introduced	**r:** refined u: used	ExpV1	ImpV1	ImpV2	ImpV3	ObjV1	ObjV2	ObjV3	FraV1	FraV2	FraV3
Concept	Description										
Program	start symbol of each language	**i**	**r**	u	u	u	**r**	**r**	**r**	u	**r**
exp	synonym for expressions	**i**	**r**	**r**	u	u	**r**	**r**	**r**	u	**r**
use	the use expression			**i**	**r**	u	u	u	u	u	u
lit, Number, Boolean, uExp, bExp, cExp											
stm	synonym for statements		**i**	**r**	**r**	u	u	**r**	**r**	**r**	**r**
printStm, ifStm, asgnStm, body, block, bstm, var											
type	types of the language				**i**	u	**r**	**r**	u	u	u
throwStm	exception throwing									**i**	
tryCatchFinally	finally part of exception catch									**i**	
tryCatchClause	try part of exception catch									**i**	
catch	single exception catch clause									**i**	

Fig. 100: Roaster of FraV2 features and their introduction (i), refinement (r), and use (u) in the different languages

14.3 FraV2: Models of Exceptions

Example language *FraV2* features exception throws and try-catch-finally constructs. It is directly formulated as an extension and refinement of *ObjV1*.

Gram. 18: (refines Grammar 14)

stm	=	... | *throwStm* | *tryCatchFinallyStm*
throwStm	::=	*"throw" exp ";"*
tryCatchFinallyStm	::=	*tryCatchClause ["finally" block]*
tryCatchClause	::=	*"try" block* { *catch* }
catch	::=	*"catch" "(" exp ")" block*

The semantics of *FraV2* is basically given using the frame pattern of Section 14.1, and therefore the given Montages can be freely combined with other languages based on the frame pattern. The exact definition of *Frame* and the declaration of the constructor *exception* are given as follows.

Decl. 15:

```
derived function Frame ==
 {"tryCatchClause", "tryCatchFinallyStm", "catch"}
constructor exception(_)
```

Exceptions in *FraV2* are triggered using the *throwStm* construct (Figure 101), an instance of the abrupt-pattern. In our simplified setting the information within the *exception(_)* constructor are arbitrary values, and exception catching (Figure 102) is based on equality of the exception information and the value in the catch clause. In object oriented languages, exceptions are typically instances of a special exception-class, and catching is done by checking for types, rather than values. The presented Montages have been applied to this situation as well, in fact they are taken directly from the specification of exception handling in Java.

The following three Montages *catch*, *tryCatchFinallyStm*, and *tryCatchClause* (Figures 102, 103, and 104) are refining the frame pattern by introducing a second action *execFinally*, which is used to guarantee that the control executes the block after the “finally” keyword in *tryCatchFinallyStm* even if any exception or other abrupt control has been triggered.

Assume normal control enters the *tryCatchFinallyStm* (Figure 103), which leads directly into the *tryCatchClause* (Figure 104), and then into the *block*. If no abrupt control is triggered in the block, the *tryCatchClause* is then left, control flows back in the *tryCatchFinanllyStm*-Montage, and then the block after the “finally”-keyword is entered. If again no abrupt control is triggered, *tryCatchFinallyStm* is terminated normally. There are now two possible places where abrupt control can be triggered, in the block of the *tryCatchClause*, and in the block after the “finally”. We call the first block *try-block* and the second block *finally-block*, and we assume that the triggered abrupt control is an exception throw.

If an exception is thrown in the try-block, control is send to the frame handler of the *tryCatchClause* and the list of catches is entered. Each catch-clause (Figure 102) checks after its *o*-state whether the value of the exception matches its catch-value.

$$RESULT = exception(S\text{-}exp.val)$$

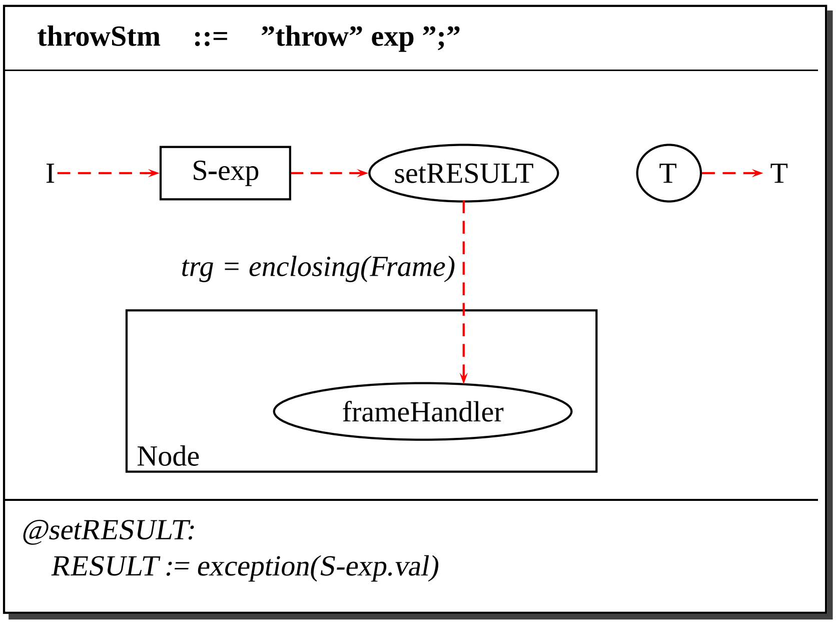
If not, control is sent to the next catch-clause in the list, and if none of the clauses catches the exception, control leaves the list of catch-clauses, exits the *tryCatchClause*, executes the finally-block, and since *RESULT* is still set to the unmatched exception, control is passed up to the frame handler of the least enclosing frame.

If the catch clause matches the exception, control is sent to the *resetRESULT*-state, *RESULT* is set to *undef* the block of the catch is executed, and control is sent out of the list to the finally-block. For this purpose the action *execFinally* is introduced, which sends control straight up to the finally-block. Thus after the block of the catch is executed, control is sent to the *execFinally*-action of the least enclosing frame.

Besides this main scenario, there are three more subtle cases, which result from abrupt-control triggered in the finally-block, the expression or block of a catch clause. We discuss here the case of exceptions triggered in these places.

- If an exception is triggered in the finally-block, the frame handler of the *tryCatchFinallyStm* sends control to the least enclosing frame.

- If an exception is triggered in the expression or block of a catch clause, the newly triggered exception must not be catched by the catch-list of the enclosing *tryCatchClause*-frame, but control must be sent to the finally-block directly. Therefore the frame handler of the catch sends control to the *execFinally*-action of the enclosing frame.



throwStm ::= ”throw” exp ”;”

@setRESULT:
RESULT := exception(S-exp.val)

Fig. 101: Montage *throwStm* of language *FraV2*

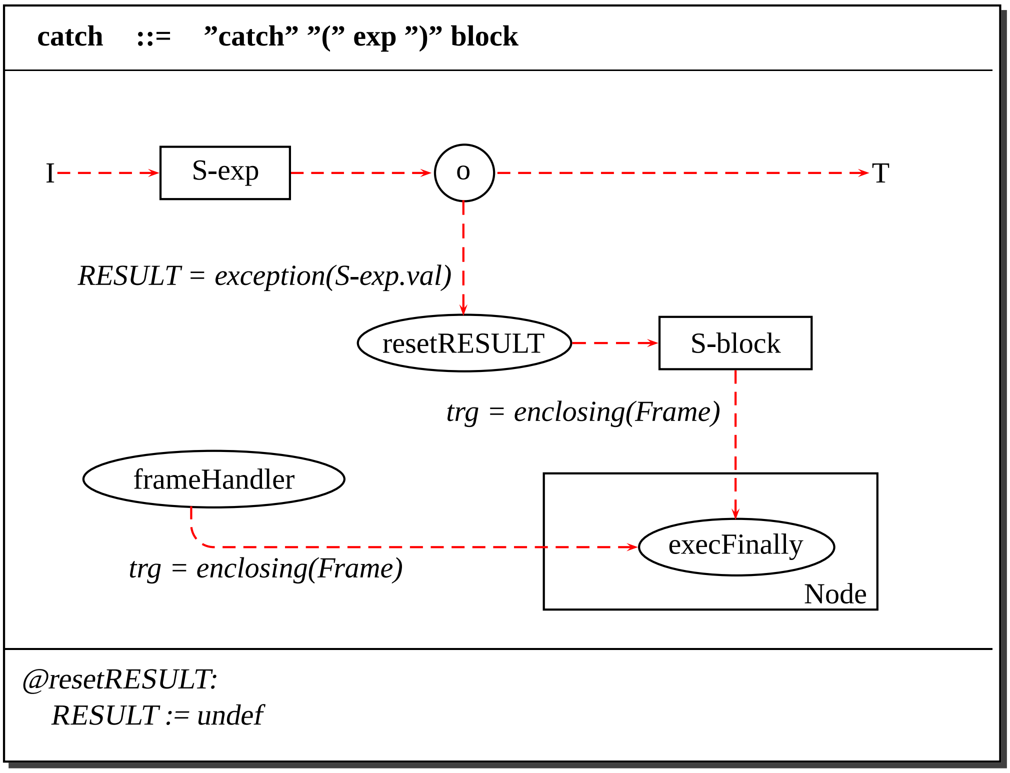


Fig. 102: Montage *catch* of language *FraV2*

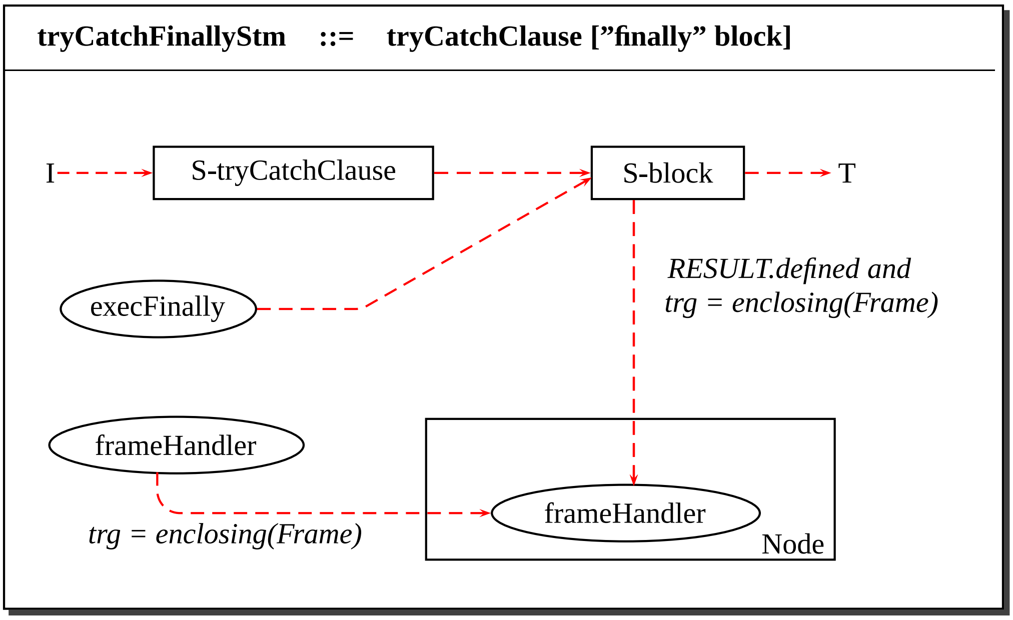


Fig. 103: Montage *tryCatchFinallyStm* of language *FraV2*

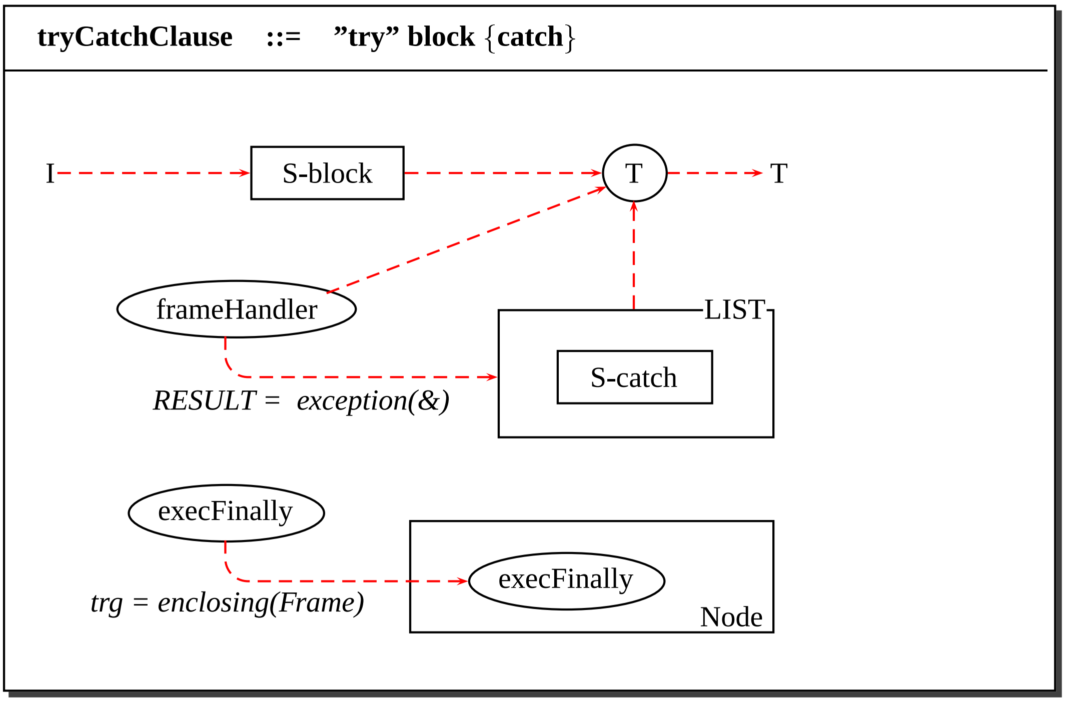


Fig. 104: Montage *tryCatchClause* of language *FraV2*

i: introduced	**r:** refined u: used	ExpV1	ImpV1	ImpV2	ImpV3	ObjV1	ObjV2	ObjV3	FraV1	FraV2	FraV3
Concept	Description										
Program	start symbol of each language	**i**	**r**	u	u	u	**r**	**r**	**r**	u	**r**
exp	synonym for expressions	**i**	**r**	**r**	u	u	**r**	**r**	**r**	u	**r**
use	the use expression			**i**	**r**	u	u	u	u	u	u
lit, Number, Boolean, uExp, bExp, cExp											
stm	synonym for statements		**i**	**r**	**r**	u	u	**r**	**r**	**r**	**r**
printStm, ifStm, asgnStm, body, block, bstm, var											
type	types of the language				**i**	u	**r**	**r**	u	u	u
functionDecl	procedure dec.aration							**i**			**r**
call	procedure call							**i**			**r**
actualParam	actual parameter of call							**i**			u
returnStm	return statment							**i**			**r**

Fig. 105: Roaster of FraV3 features and their introduction (i), refinement (r), and use (u) in the different languages

14.4 FraV3: Procedure Calls Revisited

The example language *FraV3* is a revised, frame-pattern version of *ObjV3* which can be composed with the definitions of *FraV1* and *FraV2*. The declaration of the frame-universe consists only of “functionDecl”, and the constructor *callResult(_)* is needed to wrap the call results, similar how the exception values or break/continue labels have been wrapped in the last two chapters.

Decl. 16:
```
function RESULT
derived function Frame ==
 {"functionDecl''}
constructor callResult(_), ReturnPoint
```

The given Montages work like the ones of *ObjV3* in Chapter 13, with the following differences.

- In the *returnStm*-Montage (Figure 106) the result v is not directly assigned to *RESULT*, but as constructor term *callResult(v)*.

  Further control is not sent directly to the caller, but to the *frameHandler* of the least enclosing frame.

- In the *call*-Montage (Figure 107) the *frameHandler*-action is introduced, and sends control only to the *setVal*-action if the returned result is a *callResult*-term. Otherwise it sends control to the least enclosing frame. The *finishCall*-action has been removed, its work is taken over by the frame-handler in the function-declaration.

  In addition the *setVal*-action must unwrap the result from the *callResult*-term.

- Finally in the functionDecl-Montage (Figure 108) a *frameHandler*-action is added, which resets the incarnation to the last one, and sends control to the caller which is stored as value of the *ReturnPoint*-constant.

A subtle change to the previous specification in *ObjV3* is that the call-node where one has to return is no more stored as *ReturnPoint*-field of the new incarnation, but as *ReturnPoint*-field of the old incarnation.

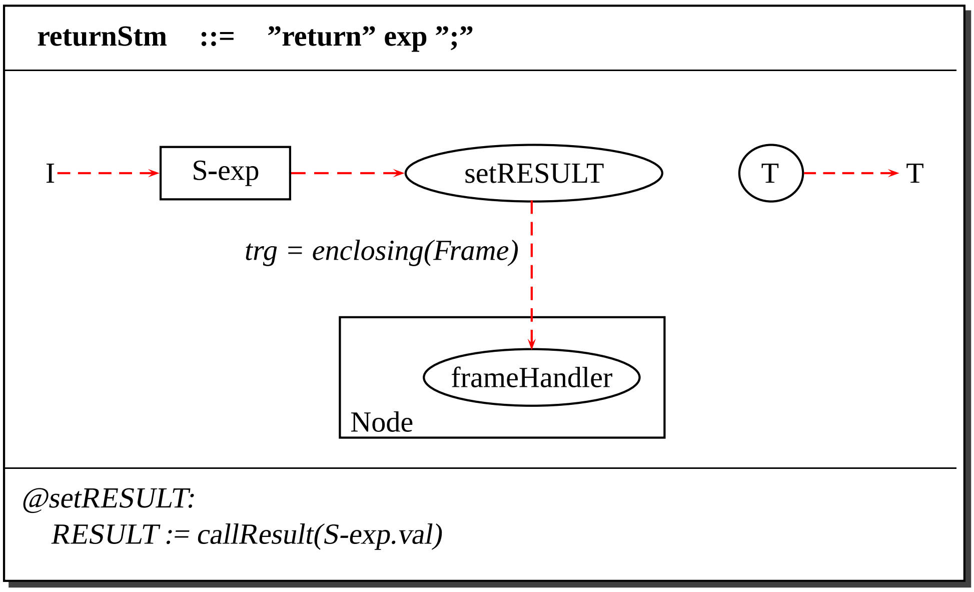
This change may seem unnecessary, but it turned out to be the only choice due to the following situation. Since we want to allow any kind of abrupt control flow exiting a call correctly, we need to reset the incarnation in the frame-handler of the function declaration. All other choices are not correct:

- if the incarnation is reset in the frame handler of the call, wrong behavior results from abrupt control triggered in the actual parameters of the call[2]

- if the incarnation is reset in a special *finishCall*-action which is located between the frame-handler and the *setVal*-action of the *call*-Montage, we obtain the opposite error: abrupt control returning from the call, but not being a call-result is not triggering the reset of the incarnation and therefore leads to wrong behavior.

Since we therefore need to reset the incarnation in the frame-handler of the function-declaration, it is not possible to access the *ReturnPoint*-value on the new incarnation, which has been lost for ever by reseting the current incarnation to the old one. Therefore it is mandatory in this new situation to store the call-node to which we have to return in the old incarnation.

[2] In fact, if we assume a very generalized language design, where return-statements can be used as expressions, then we would need to further refine the semantics in order to avoid the error that a call-result issued by an actual parameter would be interpreted as result of the not yet called function. Our solution works perfectly if the only abrupt control we expect from the actual parameters are exceptions. Since this is the case in all main-stream language we know, we are not further refining the specification at this point.

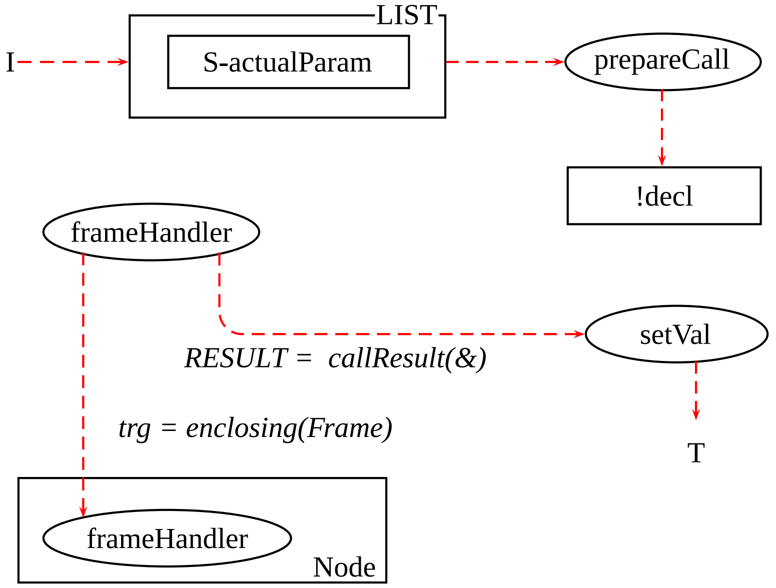


returnStm ::= ”return” exp ”;”

```
@setRESULT:
  RESULT := callResult(S-exp.val)
```

Fig. 106: Montage *returnStm* of language *FraV3*

call ::= **id ”(” [actualParam {”,” actualParam}] ”)”**
actualParam = exp

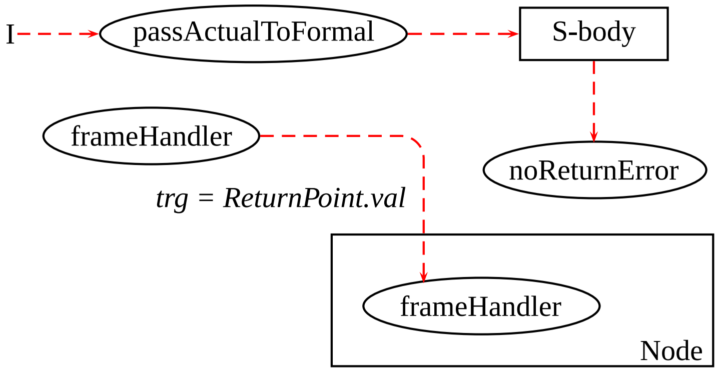


attr signature == S-id.Name
attr decl ==
enclosing({”Program”}).declTable(signature)
attr staticType == decl.staticType

```
@prepareCall:
   RESULT :=   S-actualParam.combineActualParams
   extend INCARNATION with i
      ReturnPoint.val := self
      i.lastInc := Incarnation
      Incarnation := i
   endextend

@setVal:
   if RESULT = callResult(&r) then
      val := &r
      RESULT := undef
   endif
```

Fig. 107: Montage *call* of language *FraV3*

functionDecl ::= ”function” id ”(” {var} ”)”
”:” type body



attr staticType == S-type.staticType
attr signature == S-id.Name
attr declTable(pStr) ==
(choose p in sequence S-var :
p.signature = pStr)

```
@passActualToFormal:
   val := PassParameters(RESULT, S-var)
   RESULT := undef

@frameHandler:
   Incarnation := Incarnation.lastInc

@noReturnError:
   handleError(”Exiting without return error”)
```

Fig. 108: Montage *functionDecl* of language *FraV3*

Part IV

Appendix

A

Kaiser’s Action Equations

Unpublished joint work with Samarjit Chakraborty.

Among the several mechanisms proposed for specifying programming environments, attribute grammar systems have been one of the most successful ones. The main reason for this lies in the fact that they can be written in a declarative style and are highly modular. However, by itself they are unsuitable for the specification of dynamic semantics. The work of Gail Kaiser on *action equations* (AE) (111; 112) addresses this problem by augmenting attribute grammars with mechanisms taken from *action routines* proposed by Medina-Mora in (151) for use in language based environments. In this appendix, action equations are described and compared with Montages.

A.1 Introduction

Action routines are based on semantic routines used in compiler generation systems such as Yacc, in which the semantics processing is written as a set of routines in either a conventional programming language or a special language devised for this purpose (3). Each node in the abstract syntax tree (AST) is associated with such actions and the execution of a construct is triggered by calling the corresponding action routine. In contrast to this, actions in AE are given by a set of rules similar in form to semantic equations of attribute grammars. Such equations are embedded into an event-driven architecture. Events occurring at any node of the AST activate the attached equations in the same sense in which in the action routines paradigm commands trigger the associated action routines. Equations which are not attached to any events correspond exactly to the semantic equations of attribute grammars. Equations in this framework can be of five types: assignments, constraints, conditionals, delays and propagates. Assignments and constraints are exactly similar in form, with the difference being that constraints are not attached to events and hence are active at all times. The propagate equations propagate an event from one node of the AST to another after evaluating the equations in that node. Thus the control flow is modeled by propagation of events from one node to the other.

This appendix reevaluates the problem of specifying dynamic semantics in an attribute grammar framework for language definitions in an environment generator, by comparing AEs with Montages.

Montages can be seen as a combination of Attribute Grammars and Action Routines. For giving the *actions*, Montages use Abstract State Machine (ASM) rules. There exist a number of case studies applying ASMs to the specification of programming languages. In the case of imperative and object oriented languages, these applications work in the same way as Action Routine specifications, but they have a formal semantics. Montages adapt and integrate the ASM framework for specifying dynamic semantics with attribute grammars, and a visual notation for specifying control-flow as state transitions in a hierarchical finite state machine (FSM).

In short the differences between AE and Montages can be summarized as follows. In AE, the semantic processing at each node of the abstract syntax tree (AST) is given by sets of *equations* which are attached to particular events. The triggering of an event at a node leads to a reevaluation of these equations. Montages on the other hand uses ASM rules to specify such semantic processing, which is strictly different from the concept of using equations.

As a second difference, control flow in AEs is specified by *propagating* an event from a source to a destination node, thereby activating the equations associated with this event in the destination node. In contrast to this, control flow in Montages is specified by state transitions in a finite state machine, which is described using graphical notation.

Section A.2 describes how control flow is specified using action equations. Section A.3 contains a description of a number of different control-structures specified using Montages which are found in any imperative or object-oriented language. These are compared to the corresponding specifications written using AE. In order to simplify comparison, we base the Montages direct on the abstract syntax definitions.

In one example (Example 6) we show a programming construct whose ASM-action cannot be given as AE equation and in other example (Example 3) we show that our visual notation makes it substantially easier to understand a specification. In the process of describing with Montages the control structures corresponding to AE examples in the literature, an error was discovered in Example 3 of Kaiser's article in ACM transactions on programming languages and systems (112). The same error would have been hard to overlook in the graphical Montages description.

A.2 Control Flow in Action Equations

As described above, the AE paradigm is based on the concept of attaching a set of equations with non-terminals of the grammar, and thereby with the instances of the non-terminals as the nodes of the AST. The occurrence of an event at a node of the AST leads to an evaluation of the equations attached to that particular event in that node. Events, like attributes in attribute grammars, can be either synthesized or inherited. The events associated with the left-hand non-terminal of a production, as shown below, are synthesized.

Example 1
production

```
    event_1 →
      equation_{1,1}
      . . .
      equation_{1,m}
   . . .
    event_p →
      equation_{p,1}
      . . .
      equation_{p,q}
```

Here $equation_{1,1}$ through $equation_{1,m}$ are attached to $event_1$, and similarly for the other events. Inherited events with their attached equations are associated with the right-hand non-terminals of a production. In (112) the left-hand non-terminal is referred to as the goal-symbol, the non-terminals on the right as the components of the goal symbol, and the context-free grammar notation is the same as that introduced in Example 1. Using this notation the inherited events are given as

Example 2
goal symbol ::=
  component$_1$**: type**
  . . .
  component$_n$**: type**

    event$_a$ **On component**$_1$ $\rightarrow$
      equations
    event$_b$ **On component**$_1$ $\rightarrow$
      equations
    . . .

    event$_z$ **On component**$_n$ $\rightarrow$
      equations

The *On* keyword is used to denote that the inherited event is associated with the named component. It was also mentioned that the *propagate* equation is used to propagate an event from a source to a destination node of the AST. This has the effect of activating the equations at the destination node attached to the named event. Formally the equation is stated as

Propagate event To destination

Using these equations at each step of the computation, set of equations is dynamically determined and activated. The reevaluation of these equations results in the redefinition of a number of attributes. This redefinition of attributes is used for side-effects. The next Section shows the AE specifications for common control constructs and compares these with Montages specifications for the same constructs. Throughout the Section, sequential control flow is modeled with two kind of events, *Execute* and *Continue*.

A.3 Examples of Control Structures

Example 3
As first example how to model dynamic semantics with AE we take the if statement, as it is described in (112). The *ifStm* has two children, the condition-part being an expression, and the thenpart, being a statement.

```
ifStm ::= condpart: EXPRESSION
          thenpart:  STATEMENT
```

When the *Execute* event occurs at an instance of *ifStm*, the *Execute* is propagated to the *condpart*.

```
Execute ->
  Propagate Execute To condpart
```

ifStm ::= condpart: EXPRESSION
thenpart: STATEMENT

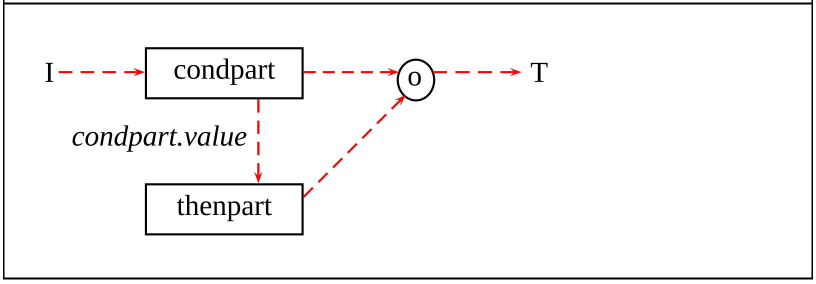


Fig. 109: The ifStm Montage

After any semantics processing involving the *condpart* are completed (including, for example, the setting of its *value* attribute), then the *condpart* propagates the *Continue* event to itself. A *Continue* on the *condpart* activates the following pair.

```
Continue On condpart ->
 If condpart.value
  Then Propagate Execute To thenpart
  Else Propagate Continue To self
```

If the *value*-attribute evaluates to true, *Execute* is propagated to the *thenpart*. If not, the if statement has completed execution, and *Continue* is propagated to itself.

After the *thenpart* terminates, the *Continue* is correspondingly propagated to the ifStm.

```
Continue On thenpart ->
  Propagate Continue To self
```

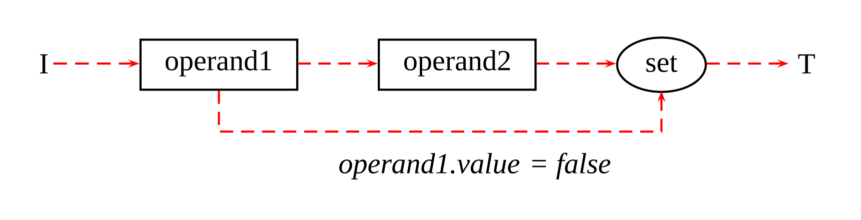
Figure 109 we see how the same mechanism is given in terms of a FSM. It the *ifStm* is executed, the first visited state is the *condpart*. The semantics processing involving the *condpart* is given by the related FSM, whose actions set for instance its *value* attribute. The *condpart* has then two outgoing control edges along which the processing of the *ifStm* continues. One of the edges is labeled by

$$condpart.value$$

and the other has no label. In such cases, the non-labeled edge is assumed to represent the else-case, e.g. the case when all labels of other edges evaluate to true. Consequently, if the *condpart.value* is true, control continues to the *thenpart*, otherwise control leaves the *ifStm* through the terminal T. When the semantic processing of the *thenpart* terminates, control leaves the *ifStm* along the unique outgoing arrow.

The advantage to have an explicit visual representation of the control flow is that it is much easer to understand and validate the semantics of a construct like the *ifStm*. This is even indicated by the fact that while we entered the above

boolAnd ::= operand1: EXPRESSION
operand2: EXPRESSION



@set:
if operand1.value then
value := operand2.value
else
value := false
endif

Fig. 110: The boolAnd Montage

example we found that the "Continue On thenpart" rule is missing in (112). This rule corresponds to the unique outgoing arrow from the *thenpart*, and it the user would forget this arrow it would be immediately clear that something is missing.

Example 4

The following AE description gives the semantics of a lazy evaluated boolean and as available for instance in Pascal. The second operand must not be evaluated, if the first operand evaluates to false. This is important for the semantics, since expressions may have side effects. After the evaluation of the operands, the value is equal to the value of *operand2*, if the value of *operand1* is true, otherwise it is equal to false.

```
boolAnd ::= operand1: EXPRESSION
            operand2: EXPRESSION

  Execute ->
    Propagate Execute To operand1

  Continue On operand1 ->
    If operand1.value
    Then Propagate Execute To operand2
    Else Propagate Continue To self

  Continue On operand2 ->
    Propagate Continue To self

  Continue ->
    If operand1.value
    Then value := operand2.value
    Else value := false
```

loop **::=** **initialization: STATEMENT**
condition: EXPRESSION
body: STATEMENT
reinitialization: STATEMENT

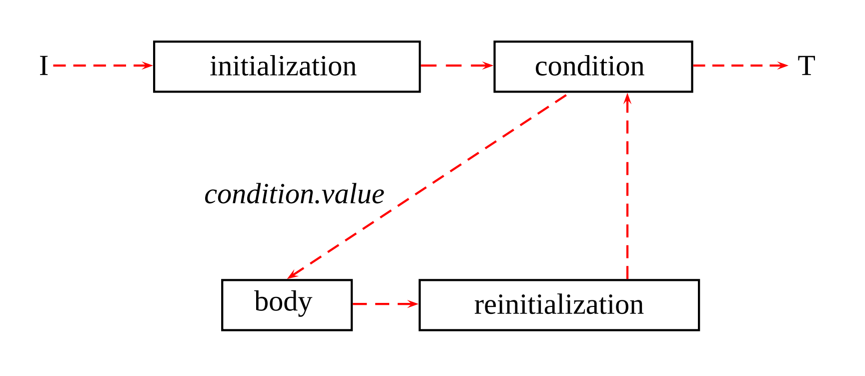


Fig. 111: The loop Montage

In Figure 110 we see the equivalent Montage. While the form of the value calculation remains the same, the visualization of the control flow shortens the length of the textual elements considerably.

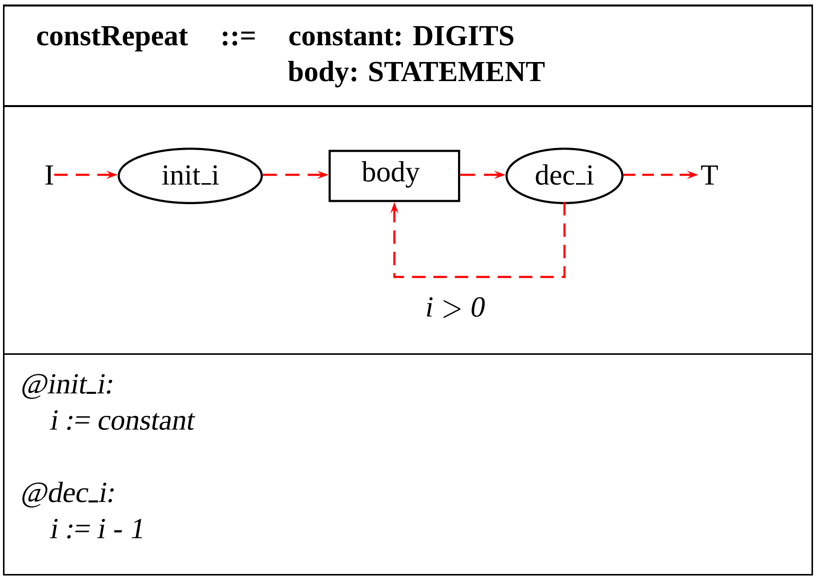
Example 5

Another example is the following loop construct. After initialization, the control loops until the condition evaluates to false. In each cycle, the reinitialization is executed. While in Figure 111 the cyclic control structure is explicitly visible, in the following AE description it is encoded using the events.

```
loop ::= initialization:   STATEMENT
         condition:        EXPRESSION
         body:             STATEMENT
         reinitialization: STATEMENT

  Execute ->
    Propagate Execute To initialization

  Continue On initialization, reinitialization ->
    Propagate Execute To condition

  Continue On condition ->
    If condition.value
    Then Propagate Execute To body
    Else Propagate Continue To self

  Continue On Body ->
    Propagate Execute To reinitialization
```

Example 6

In a last example we consider a simple construct that repeats a statement n-times, where n is a constant, positive integer.



constRepeat ::= constant: DIGITS
body: STATEMENT

@init_i:
i := constant

@dec_i:
i := i - 1

Fig. 112: The constRepeat Montage

```
constRepeat ::= constant: DIGITS
                body:     STATEMENT
```

In a Montages specification we would introduce an attribute i, initialize it with *constant*, and after each time we executed the *body* we decrease the value of i by one. If after this i is still larger than 0, the body is reevaluated, else *constRepeat* terminates. In Figure 112 the complete Montage is given, using the name *init_i* and *dec_i* for the two states doing the initialization and the decreasing.

Naively one would model this in a similar way with AEs:

```
constRepeat ::= constant: DIGITS
                body:     STATEMENT
  Execute ->
    i := constant
    Propagate Execute To body

  Continue On body ->
    if (i - 1) > 0 then
       Propagate Execute To body
       i := i - 1
    else
       Propagate Continue To self
```

But using the AE framework, the formalization of

$$x = x - 1$$

is not possible with one equation. There is an intrinsic circular dependency in such an equation and the try to evaluate it would not lead to a solution.

The only possible solution is to introduce a help-attribute h, and to activate in a first step the equation

$$h = x - 1$$

and then in a next step to activate the equation

$$x = h$$

In order to introduce an intermediate step, one needs to introduce a new event *helpEvent*. Using this the complete AE solution is:

```
constRepeat ::= constant: DIGITS
                body:     STATEMENT
  Execute ->
    i := constant
    Propagate Execute To body

  Continue On body ->
    if (i - 1) > 0 then
       h := i - 1
       Propagate helpEvent To self
    else
       Propagate Continue To self

  helpEvent ->
     i := h
     Propagate Execute To body
```

This solution introduces an additional complexity which makes the designer's task more tedious and specifications more verbose, respectively. In this respect, being Montages based on ASM, which is a Dynamic Abstract Data Type framework, presents the advantage that one can express directly the following update

$$x := x - 1$$

requesting that the original value of the 0-ary function x can be discarded and replaced by a new one without an intermediate step, i.e. by means of a non homomorphic transformation of the algebra modeling the state before the modification.

A.4 Conclusions

This appendix compared two different paradigms which extend the attribute grammar framework in different ways, for the specification of dynamic semantics in a programming environment generator. Most of the previous work on environment generators were more concerned with the generation of a language-based editing system. The design of the AE paradigm followed this line, the main focus being incremental semantic processing during editing. In contrast to this, the Montages framework is concerned with the rapid prototyping of a language and focuses on issues like ease of specification.

It is understandable that the event oriented view is helpful and probably even necessary for the specification of a system which has to do some interactive processing. Apart from the *Execute* and the *Continue* events of AE described in this paper which models the control flow, other events arising from the functionality required in an editor include events like *Create*, *Delete*, *Clip*, etc. Although an editor is currently not generated in the Gem-Mex tool-suite for Montages, we do not foresee any difficulties in doing so.

The event-based framework of AE can result in triggering a set of rules from different nodes of the AST. As a result of this equations in different nodes can be active at the same time. Such a system is highly distributed and well suited for situations other than dynamic semantics of sequential languages. In this paper we consider only the application of the event-mechanism to situations with a single sequential tread of control. For these situations we are able to present the sequential control flow in terms of FSMs. For distributed situations FSMs would have to be replaced with PetriNets or StateCharts.

B

Mapping Automata

Joint work with Jörn Janneck, published as technical report (101)

In this appendix we describe *Mapping Automaton* (MA), a variant of Gurevich's Abstract State Machines (GASM). The motivation for this work is threefold. First we want to make the MA view explicit in a formal way. Second the MA and the mapping from GASM to MA serve as implementation base for a GASM interpreter written in Java (100). And finally the definition of MA simplifies the syntactic aspect as well as the structure of a state by removing the concept of 'signature'.

Removing signature and the induced structure from the specification language and the state, respectively, makes state and specification completely orthogonal, only connected by an interpretation of the basic syntactic constants. These constants play the role of syntax (vocabulary), which are independent from the structure of the semantics (objects, and the interpretation of σ).

In effect, any specification may be interpreted in any state (that has certain basic properties, such as being 'big' enough to allow sufficiently many objects to be allocated), which in turn means that different specifications may be interpreted on the same state.

We believe that this will allow us to compose specifications much easier than was possible in GASM, an interesting aspect of this improved compositionality possibly being the easy integration of object-based constructs into the concept with a view of making it a practical specification and prototyping method in such environments (99).

B.1 Introduction

The motivation for MA starts with Gurevich's claim that in dynamic situations, it is convenient to view a state as a kind of memory that maps locations to values (82). A location is a pair of an r-ary function name and an r-tuple of elements. Such a memory is partitioned in different areas each consisting of the locations belonging to one function. We believe that it is often more appropriate to view a state as a collection of objects, each associated with a mapping from attributes to values. In this view the notions of attribute, value, and object are unified. This allows to model a large number of commonly used data structures, e.g. records with pointer attributes, arrays with dynamic length, stacks, or hashtables.

For the moment we restrict our interest to completely untyped object systems. Such systems can be modeled with a Tarski structure having only one binary function, encoding the objects and their associated mapping. We fix the name of this function to σ. *Mapping Automaton* (MA), is a name for the combination of the above explained object-view on state with GASM whose vocabulary contains only the binary σ and a set of static constants. We define and investigate MA as a mathematical object, by adopting the definition of GASM over mapping-structures to the MA view, i.e. the σ function is made part of the formal definition of MA states. Finally we give a formal mapping from GASM to MA.

In the next section, the used static structures are described, then MA are defined formally. In Section B.4 the definition of transition rules is adopted to MA. In the last section of this chapter the mapping from GASM to MA is formalized.

B.2 Static structures

Before we present MA as describing the dynamic transition from one state to the next, we first make precise our notion of state. For MA, this notion is completely independent of any syntactical concepts and indeed of the existence of any MA defined for it.

B.2.1 Abstract structure of the state

Our intuitive concept of state is that of a structure between objects of a set. This set, the set of all admissible objects that may ever occur in the computation to be modeled, we will subsequently call our *universe* $\mathcal{U}$. We will not make any assumptions about its nature, except that it be big enough (cf. section B.4.5 for details on this) and contain a special element $\perp$. We will refer to the elements of $\mathcal{U}$ as *objects*.

Given such as universe we can now define our concept of state as follows: Intuitively, we may think of a state as a mapping σ, that assigns each element of $\mathcal{U}$ a unary function over $\mathcal{U}$. Many common data structures can be directly conceptualized in this way: records (mapping field names to field values), arrays (indices to values), hash-tables (keys to values), etc. Of course, higher arities may be modeled by successive application of unary functions or with tuples.[1]

Alternatively, and equivalently, a state may be regarded as a mapping of pairs of objects to objects, i.e. as a two dimensional square table with objects as entries. Formally,

Def. 28: State space.. *Given a universe $\mathcal{U}$, we define the state space of $\mathcal{U}$ to be*

$$\Sigma = \mathcal{U} \times \mathcal{U}$$

Note that the equation

$$(\mathcal{U} \times \mathcal{U}) \longrightarrow \mathcal{U} = \mathcal{U} \longrightarrow \mathcal{U}^{\mathcal{U}}$$

supports the alternative views of the state as either a square table populated by objects or a mapping of objects to mappings.

Since these are two equivalent manners of speaking, we will freely alternate between these two conceptions of a state, talking about a mapping associated with an object, or equivalently refer to an object as being an index to a row in the state table (assuming here and in the following that a row corresponds to a mapping).

B.2.2 Locations and updates

The structure of such a state is changed in one atomic action by a set of pointwise *updates*, which specify a *location* to be set to a new *value*. However, MA locations are somewhat simpler than those in GASM, since they basically specify a place in the two-dimensional position in the state table, i.e. they are a pair of objects.

Def. 29: Location and update.. *Given a universe $\mathcal{U}$, a location is a pair in $\mathcal{U}$, the set of all locations is $\Lambda = \mathcal{U} \times \mathcal{U}$. An update is a pair consisting of a location and an element in $\mathcal{U}$, the set of all updates is thus defined as $\mathbf{U} = \Lambda \times \mathcal{U}$.*

Applying a set of such updates results in a new state, with the entries in the square table changed to the values given in the update set:

Def. 30: Application of update set.. *Given a state $\sigma \in \Sigma$ and an update set $\mathbf{u} \subset \mathbf{U}$, applying $\mathbf{u}$ to σ yields the successor state σ' – symbolically $\sigma \xrightarrow{\mathbf{u}} \sigma'$ – that is defined as follows:*

$$\sigma' \ a \ b = \begin{cases} v & ((a,b),v) \in \mathbf{u} \\ \sigma \ a \ b & \textit{otherwise} \end{cases}$$

[1] See also the discussion in section B.5.2 for more details.

Clearly, the above definition only yields a well-defined function if the update set contains at most one new value for a given location. This condition is called *consistency*.

Def. 31: **Consistency..** *An update set* $\mathbf{u}$ *is called consistent, iff*

$$\forall(\lambda_1, v_1), (\lambda_2, v_2) \in \mathbf{u} : \lambda_1 = \lambda_2 \Longrightarrow v_1 = v_2$$

In the following, we assume an update set to be consistent. Since there are several possible ways of defining the effects of the application of inconsistent update sets, each with its respective merits and drawbacks, we will not commit ourselves to one particular version and choose to leave this point open for further discussion.

B.3 Mapping automata

Mapping Automata (MA) describe the evolution of a state as defined above. Although its structure differs slightly from GASM, where it is an algebra of a given signature, the evolution is still described by a rule, that computes an update set for a given state and the application of this update set to the state it was computed for, resulting in the successor state.

Formally, we define MA as follows:

Def. 32: **Mapping automaton..** *A mapping automaton is a pair* $(\mathbf{C}, \mathbf{R})$, *with* $\mathbf{C} = \{c_i\}$ *a set of* constant symbols *and* $\mathbf{R}$ *a* rule.

The constant symbols c_i are similar in function to the signature in GASM in that they serve as anchor points for interpretation and also term evaluation, as will be seen below.[2]

Such an MA is related to some state universe by an *interpretation* as follows:

Def. 33: **Interpretation..** *Given a universe* $\mathcal{U}$ *and a mapping automaton* $\mathcal{M} = (\mathbf{C}, \mathbf{R})$, *we call a function* $\mathcal{I} : \mathbf{C} \longrightarrow \mathcal{U}$ *an interpretation of* $\mathcal{M}$.

Without going into the details of how such a rule may be described (this will be the task of section B.4, this is what it *does*: Given an interpretation, it computes an update set from some state. Formally,

Def. 34: **Rule..** *Given an MA and an interpretation of its constant symbols, its rule* $\mathbf{R}$ *maps states to update sets:*

$$\mathbf{R} : \Sigma \longrightarrow \mathbf{U}$$

[2] In fact, as will become clear in section B.4, these symbols not only serve as constants, but also as the namespace for quantified and other variables. However, since the interpretation $\mathcal{I}$ is never updated during the execution of an MA, and since even when some variable binding shadows a constant in the scope of a rule, this at least is not destructively modified in its scope, we will stick to this name.

Now we can make precise the ’dynamics’ of an MA, by defining a *run* starting from some state σ:

Def. 35: Run.. *A run of an MA* $(\mathbf{C}, \mathbf{R})$ *starting from some initial state* σ *is a sequence* $(\sigma_i)_{i \in \mathbb{N}}$ *such that*

- $\sigma_0 = \sigma$
- $\sigma_i \overset{\mathbf{R}(\sigma_i)}{\longrightarrow} \sigma_{i+1}$

Of course, a run terminates iff ex k such that $\sigma_i = \sigma_{i+1}$ for all $i > k$.

B.4 A rule language and its denotation

In the following we will suggest a notation for MA rules, which parallels the one suggested for GASM in (82). Following (82), we will give the denotation of each construction in our notation in terms of the update set that it represents given an interpretation and a state – according to definition 34. First, however, we will develop the notion of *term*, which are basic constituents in most rule constructs.

B.4.1 Terms

Terms are some kind of syntactic structure that we use to refer to objects of the universe. Some objects of the universe we can refer to directly using constant symbols and an interpretation of them. For others we form compound terms and use the state. Therefore, we will define the evaluation in a given state $\sigma \in \Sigma$ and under some interpretation $\mathcal{I}$.

MA terms are very simple structures:[3] They are either constant symbols, or pairs of terms. The latter can be intuitively thought of as signifying the application of the mapping that is bound to the value of the first term to the value of the second - which is the intuition that is responsible for the name of mapping automata.[4] Since we also need a basic predicate testing for the equality (i.e. identity) of two objects, this is also a term.

Def. 36: Terms.. *Let* $\mathbf{C}$ *be a set on constant symbols. Then the set of all terms* $\mathcal{T}_{\mathbf{C}}$ *of* $\mathbf{C}$ *is defined to be the smallest set such that*

- $\mathcal{C} \subset \mathcal{T}_{\mathbf{C}}$
- $\mathbf{s}, \mathbf{t} \in \mathcal{T}_{\mathbf{C}} \Longrightarrow \langle \mathbf{s}\ \mathbf{t} \rangle \in \mathcal{T}_{\mathbf{C}}$

[3] However, see. section B.4.3.2 for an extension that complicates things somewhat.

[4] Making application left-associative, one can write the term $\langle\langle a\ b\rangle\ c\rangle$ in the more familiar for $a\ b\ c$.

- $\mathrm{s}, \mathbf{t} \in \mathcal{T}_{\mathbf{C}} \Longrightarrow \mathrm{s} = \mathbf{t} \in \mathcal{T}_{\mathbf{C}}$

They are assigned a value in a given state in a most straightforward way: constants are mapped to their interpretation, while pairs are evaluated by applying the map associated with the first element to the value of the second, or, equivalently, simply applying the state σ to the pair of values of the two terms. The identity test is $\perp$ if the two terms to not yield the same object. If they do, however, this test must produce some other element, which we will call $\top$ here, but which has no special significance other than being different from $\perp$.

Def. 37: Term evaluation.. *Given a set of constant symbols* $\mathbf{C}$*. Then we define the* value $val_{\sigma,\mathcal{I}}[t]$ *of a term* t *in a state* $\sigma \in \Sigma$ *under interpretation* $\mathcal{I}$ *recursively as follows:*

$$\begin{aligned} val_{\sigma,\mathcal{I}}[c] &= \mathcal{I}(c) \qquad \textit{for} \quad c \in \mathbf{C} \\ val_{\sigma,\mathcal{I}}[\langle s\ t\rangle] &= \sigma\ val_{\sigma,\mathcal{I}}[s]\ val_{\sigma,\mathcal{I}}[t] \\ val_{\sigma,\mathcal{I}}[s = t] &= \begin{cases} \top & val_{\sigma,\mathcal{I}}[s] = val_{\sigma,\mathcal{I}}[t] \\ \perp & \textit{otherwise} \end{cases} \end{aligned}$$

B.4.2 Basic rules constructs

Now we will outline a few basic rule constructs and give their meaning by the rule they denote.

The skip construct

$$skip$$

has no effect on the state. Its denotation is accordingly the empty set for any state:

$$Den_{\mathcal{I}}[\textit{skip}](\sigma) =_{def} \emptyset$$

The most fundamental non-empty rule construct is the single atomic update, which we denote as

$$t_1\ t_2 := t$$

Given a state σ, it denotes an update set consisting of one update:

$$Den_{\mathcal{I}}[t_1\ t_2 := t](\sigma) =_{def} \{((val_{\sigma,\mathcal{I}}[t_1], val_{\sigma,\mathcal{I}}[t_2]), val_{\sigma,\mathcal{I}}[t])\}$$

The conditional rule construct decides which of two rules to fire according to the value of a term:

$$\textit{if } t \textit{ then } R_1 \textit{ else } R_2 \textit{ endif}$$

Its denotation is therefore:

$$Den_{\mathcal{I}}[\textit{if } t \textit{ then } R_1 \textit{ else } R_2 \textit{ endif}](\sigma) =_{def} \begin{cases} Den_{\mathcal{I}}[R_1](\sigma) & val_{\sigma,\mathcal{I}}[t] \neq \perp \\ Den_{\mathcal{I}}[R_2](\sigma) & \text{otherwise} \end{cases}$$

We also define the parallel composition of two rule descriptions, written as[5]

$$R_1\ R_2$$

Its denotation is simply the union of the update sets:

$$Den_{\mathcal{I}}[\ R_1\ R_2](\sigma) =_{def} Den_{\mathcal{I}}[R_1](\sigma) \cup Den_{\mathcal{I}}[R_2](\sigma)$$

B.4.3 First-order extensions

As shown by Gurevich (82), one can add first-order constructs to describe both rules and terms. We will start with rule constructs and then turn to first-order terms.

B.4.3.1 Do-forall rule

The do-forall rule construction allows to compute the update set of a rule description R with some constant symbol bound to each element of some set. Its syntax is as follows:

$$\textit{do forall } c \textit{ in } s\ :\ R \textit{ enddo}$$

c is a constant symbol, R a rule description, and s specifies the set the elements which c will be bound to in R.

Clearly, we must somehow restrict the sets that may thus be iterated upon, not only for practical reasons.[6] We choose to restrict s to constructions of the form $dom\ t$ or $ran\ t$, where t is any term. These then denote the domain and range, respectively, of the mapping associated with the value of t.[7]

Def. 38: Domain and range of mappings.. *Given an $a \in \mathcal{U}$, we define its domain and range (equivalently the domain and range of the mapping associated with it) as*

$$\begin{aligned} dom_\sigma\ a &=_{def} \{x \in \mathcal{U} \mid \sigma\ a\ x \neq \bot\} \\ ran_\sigma\ a &=_{def} \{x \in \mathcal{U} \setminus \{\bot\} \mid \exists y \in \mathcal{U} : \sigma\ a\ y = x\} \end{aligned}$$

With this, the denotation of the above set constructions becomes

$$Set_{\sigma,\mathcal{I}}[dom\ t] =_{def} dom_\sigma\ val_{\sigma,\mathcal{I}}[t]$$
$$Set_{\sigma,\mathcal{I}}[ran\ t] =_{def} ran_\sigma\ val_{\sigma,\mathcal{I}}[t]$$

Now we can define the denotation of the do-forall rule construct as the union of all updates resulting from the body for each individual element of the specified set bound to the constant symbol:

$$Den_{\mathcal{I}}[\textit{do forall } c \textit{ in } s\ :\ R \textit{ enddo}](\sigma) =_{def} \bigcup_{a \in Set_{\sigma,\mathcal{I}}[s]} Den_{\mathcal{I}[c \mapsto a]}[R](\sigma)$$

[5] Since at this point we have no notion of *blocks* as in (82), we need no *do in-parallel* syntax that except for inconsistencies, this rule notation is otherwise equivalent to.

[6] From a theoretical point of view, allowing, a rule to iterate on, say, $\mathcal{U}$ would potentially make the entire universe accessible, and thus the reserve empty – see section B.4.5 for details.

[7] Further constructions might be useful here and *harmless* in the sense discussed in the previous footnote, such as a range of integers (if these are available) etc. However, without making any assumptions about the structure of $\mathcal{U}$, the above seem to be most natural.

B.4.3.2 First-order terms

First-order terms extend the definitions of the set $\mathcal{T}_{\mathbf{C}}$ of terms for a set of constant symbols **C** (see definition 36 by the following clauses, assuming $S =_{def} \{dom\ t \mid t \in \mathcal{T}_{\mathbf{C}}\} \cup \{ran\ t \mid t \in \mathcal{T}_{\mathbf{C}}\}$ the set of set-expressions:

- $c \in \mathbf{C} \wedge s \in S \wedge t \in \mathcal{T}_{\mathbf{C}} \Longrightarrow (forall\ c\ in\ s\ :\ t) \in \mathcal{T}_{\mathbf{C}}$
- $c \in \mathbf{C} \wedge s \in S \wedge t \in \mathcal{T}_{\mathbf{C}} \Longrightarrow (exists\ c\ in\ s\ :\ t) \in \mathcal{T}_{\mathbf{C}}$

The forall-term evaluates to $\top$ iff t evaluates to something else than $\bot$ for all elements of the set denoted by s bound to the symbol c, and to $\bot$ otherwise. The exists-term is $\bot$ if t is $\bot$ for all elements of that set, and $\top$ otherwise. Binding an object to a constant symbol c is tantamount to changing the interpretation at point c to this new value, which we will write as $\mathcal{I}[c \mapsto a]$.

$$val_{\sigma,\mathcal{I}}[(\textit{forall}\ c\ \textit{in}\ s\ :\ t)] =_{def} \begin{cases} \top & \forall a \in Set_{\sigma,\mathcal{I}}[s] : val_{\sigma,\mathcal{I}[c \mapsto a]}[t] \neq \bot \\ \bot & \text{otherwise} \end{cases}$$

$$val_{\sigma,\mathcal{I}}[(\textit{exists}\ c\ \textit{in}\ s\ :\ t)] =_{def} \begin{cases} \top & \exists a \in Set_{\sigma,\mathcal{I}}[s] : val_{\sigma,\mathcal{I}[c \mapsto a]}[t] \neq \bot \\ \bot & \text{otherwise} \end{cases}$$

B.4.4 Nondeterministic rules

The basic nondeterministic construction is

$$\textit{choose}\ c\ \textit{in}\ s\ :\ R\ \textit{endchoose}$$

Intuitively, this nondeterministically selects one of the values in the set denoted by s, binds it to c and evaluates R. In order to capture this intuition we must introduce a nondeterministic denotation $NDen_{\mathcal{I}}[R](\sigma)$ of a rule description R, which is a *set* of alternative update sets. For the choose-construct above, its (nondeterministic) denotation would be as follows:

$$NDen_{\mathcal{I}}[\textit{choose}\ c\ \textit{in}\ s\ :\ R\ \textit{endchoose}](\sigma) =_{def}$$

$$\begin{cases} \{\emptyset\} & Set_{\sigma,\mathcal{I}}[s] = \emptyset \\ \bigcup\limits_{a \in Set_{\sigma,\mathcal{I}}[s]} NDen_{\mathcal{I}[c \mapsto a]}[R](\sigma) & \text{otherwise} \end{cases}$$

Of course, we now have to give nondeterministic denotations for the other rule constructs as well, which can be done as follows:

$$\begin{aligned}
NDen_{\mathcal{I}}[\textit{skip}](\sigma) &=_{def} \\
\{Den_{\mathcal{I}}[\textit{skip}](\sigma)\} & \\
NDen_{\mathcal{I}}[(t_1,t_2) := t](\sigma) &=_{def} \\
\{Den_{\mathcal{I}}[(t_1,t_2) := t](\sigma)\} & \\
NDen_{\mathcal{I}}[\textit{if}\ t\ \textit{then}\ R_1\ \textit{else}\ R_2\ \textit{endif}](\sigma) &=_{def} \\
\begin{cases} NDen_{\mathcal{I}}[R_1](\sigma) & val_{\sigma,\mathcal{I}}[t] \neq \bot \\ NDen_{\mathcal{I}}[R_2](\sigma) & \text{otherwise} \end{cases} & \\
NDen_{\mathcal{I}}[\ R_1\ R_2](\sigma) &=_{def} \\
\{d_1 \cup d_2 \mid d_1 \in NDen_{\mathcal{I}}[R_1](\sigma) \wedge d_2 \in NDen_{\mathcal{I}}[R_2](\sigma)\} & \\
NDen_{\mathcal{I}}[\textit{do forall}\ c\ \textit{in}\ s\ :\ R\ \textit{enddo}](\sigma) &=_{def} \\
\left\{ \bigcup_{a \in Set_{\sigma,\mathcal{I}}[s]} d_a \mid d_a \in NDen_{\mathcal{I}[c \mapsto a]}[R](\sigma) \right\} &
\end{aligned}$$

Except for the do-forall case (and the parallel composition case, which can be considered a special case of the former), the nondeterministic denotation is very similar to the deterministic case, except that we talk about a set of update sets. For the do-forall construct, one has to consider all combinations of nondeterministic choices at each instance of the rule and build the union over these.

The notion of a run is of course also affected by non-deterministic constructions. If a rule yields a set of update sets instead of just one, a non-deterministic run then is defined like this:

Def. 39: **Non-deterministic run..** *A non-deterministic run of an MA* $(\mathbf{C}, \mathbf{R})$ *starting from some initial state* σ *is a sequence* $(\sigma_i)_{i \in \mathbb{N}}$ *such that*

- $\sigma_0 = \sigma$
- $\sigma_i \xrightarrow{\mathbf{u}} \sigma_{i+1}$ *such that* $\mathbf{u} \in \mathbf{R}(\sigma_i)$

B.4.5 Creating new objects

Even though the universe is a static collection of objects, in specifications we often wish to refer to hitherto *unused* or fresh objects. Therefore, instead of creating new objects and extending the universe itself, we make objects that have so far been unaccessible to the MA *accessible* by picking them from a part of the universe that we could not refer to. This part, which we will make more precise below, is called our *reserve*.

B.4.5.1 Accessibility and allocation

We will define the set of all objects $\mathcal{U}_{\sigma,\mathcal{I}}$ (or just $\mathcal{U}_\sigma$ if the interpretation is understood) that a rule can refer to and depend on in a given state σ under and interpretation $\mathcal{I}$. The definition will inductively include all elements that can be reached by the constructions of the language, starting from the elements which are the interpretation of the constant symbols:

Def. 40: Accessibility.. *Given constant symbols* $\mathbf{C}$*, we define the set* $\mathcal{U}_{\sigma,\mathcal{I}}$ *of all accessible elements of* $\mathcal{U}$ *in state* σ *under interpretation* $\mathcal{I}$ *to be the smallest set such that:*

- $\forall c \in \mathbf{C} : \mathcal{I}\ c \in \mathcal{U}_{\sigma,\mathcal{I}}$
- $a, b \in \mathcal{U}_{\sigma,\mathcal{I}} \Longrightarrow \sigma\ a\ b \in \mathcal{U}_{\sigma,\mathcal{I}}$
- $a \in \mathcal{U}_{\sigma,\mathcal{I}} \Longrightarrow dom_\sigma\ a \subset \mathcal{U}_{\sigma,\mathcal{I}}$
- $a \in \mathcal{U}_{\sigma,\mathcal{I}} \Longrightarrow ran_\sigma\ a \subset \mathcal{U}_{\sigma,\mathcal{I}}$

Clearly, the result of any rule cannot depend on any object and its surrounding structure that is not in $\mathcal{U}_{\sigma,\mathcal{I}}$. In this sense, the accessibility criterion is similar to the rules that govern garbage collection in programming language implementations.[8]

So in any state σ and interpretation $\mathcal{I}$, we can only talk about the accessible objects in $\mathcal{U}_{\sigma,\mathcal{I}}$. If we allow arbitrary 'construction' of new objects (as we do in the rule language in section B.4), we have to provide a sufficiently large universe so that we can guarantee that we can recruit new objects from the hitherto 'unused' (i.e. irrelevant) portion of the universe, which we will call our *reserve*:

Def. 41: Reserve. *The set* $\mathcal{R} = \mathcal{U} \setminus \mathcal{U}_{\sigma,\mathcal{I}}$ *is called the reserve (of state* σ*).*

The requirement for a meaningful execution of an MA is therefore that its reserve be non-empty in any reachable state. Clearly, this rules out constructions that allow iteration and updates on the entire universe, such as

$$\textit{do forall } x \textit{ in } \mathcal{U}\ :\ c(x) := c \textit{ enddo}$$

If c is a constant symbol interpreted as any non-$\perp$ value, applying the denotation of this rule to any state leads to a state where the entire universe becomes accessible.

Of course, the notion of accessibility is strongly connected to the constructions of the rule notation. If some constructs do not occur in a given MA, we

[8] However, this definition of global accessibility is far too loose for many practical applications to be used as a basis for storage allocation. Consider for example a situation where $\mathbf{C}$ is the set of all integer numerals, all strings, and all identifiers. A useful interpretation will supposedly map all these infinitely many symbols to infinitely many different objects, which thus become globally accessible, while any sensible implementation will only create those number objects as they are needed during the computation process. It might make sense, therefore, to restrict the globally accessible objects for a given MA to those which can be reached by terms formulated only in constant symbols actually occurring in the MA rules. We will not further elaborate this point here.

may adapt the accessibility definition accordingly. This is of particular importance when we restrict the language by imposing some kind of static structuring on the rules – then the set of visible elements in this kind of automaton may be quite different from the one we must assume for general MA. See section B.5.2 for an example and an application of this principle.

B.4.5.2 The import-rule

Constructing the reserve in the above way allows us to give meaning to the notion of *importing* new or fresh elements into our visible part of the universe. The basic rule to pick an object from the reserve looks like this:

$$\textit{import } c\ R \textit{ endimport}$$

This rule actually does three things: it first picks an element from the reserve, binds it to the symbol c and then executes the rule body R in the new context, i.e. in an interpretation that is identical to $\mathcal{I}$ except at point c, which is mapped to the new object instead. If we call the new object chosen from the reserve a, we can write the new interpretation as $\mathcal{I}[c \mapsto a]$, and the deterministic and non-deterministic denotation, respectively, then become

$$Den_{\mathcal{I}}[\textit{import } c\ R \textit{ endimport}](\sigma) =_{def} Den_{\mathcal{I}[c\mapsto a]}[R](\sigma) \qquad a \in \mathcal{R}$$

$$NDen_{\mathcal{I}}[\textit{import } c\ R \textit{ endimport}](\sigma) =_{def} NDen_{\mathcal{I}[c\mapsto a]}[R](\sigma) \qquad a \in \mathcal{R}$$

As in (82) we assume that different imports choose different reserve elements. Furthermore, we assume that for any new element a, $\sigma\ a\ x = \perp$ for all $x \in \mathcal{U}$. Note also, that the new object does not automatically become a member of $\mathcal{U}_{\sigma',\mathcal{I}}$: although it is in $\mathcal{U}_{\sigma,\mathcal{I}[c\mapsto a]}$, the rule body has to manipulate the state so that it can be accessed outside the rule in the next state.

B.5 Comparison to traditional ASMs

In this section we will first shed some light on what we perceive as one of the basic differences between MA and GASM, and then proceed to show their fundamental equivalence (as far as computational expressibility and level of abstraction are concerned). This will serve to document our claim that MA are basically a slightly different way of doing very similar things.

B.5.1 State and automata

A key difference between traditional ASMs and MA is the relation between a state (and the set of all states) and the automaton: A GASM state is always a state *of a vocabulary*, i.e. a signature containing some function names of various arities that impose a certain structure on the state. Also, an ASM operating meaningfully on this state must in a sense 'know' about this structure, i.e. share its vocabulary.

In MA, the situation is somewhat simpler. First, the a state can be meaningfully defined without any recourse to syntactical elements such as function names, or their MA-counterparts, constant symbols. A state is a simple structure imposed on the elements of some universe, indeed, there need not even be an MA, constant symbols, or any other syntactical conventions to be able to talk about a state.

However, when we want to refer to particular parts of such a structure, say, individual objects, we must have a way of identifying them so we can investigate the structure ’around’ them. It was felt that the most straightforward way of doing this was to simply give them names, i.e. to provide a set of names and a mapping between these names and their denotations.

These names and their interpretation, however, to not in any way introduce a structure into the system – unlike function names of various fixed arities.[9] They are basically a flat collection of distinguishable identifications of elements in the universe. The structure, therefore, is completely separated from the naming.

This separation of concerns, leaving structure to the state and naming to the automaton (and its interpretation) that describes the evolution of such a structure, can be leveraged in various ways. For instance, there is no problem in applying several automata (each with its own interpretation and even different sets of constant symbols) to the same state - concurrently, independently, alternatively. This can be used to promote a much higher degree of compositionality of automata.

When composing a specification of a set of automata, it might make sense to require them to share the same set of constant symbols. For GASM, sharing the same signature over a large number of automata would seem like a somewhat unnatural requirement, and possibly even involve a good deal of renaming, prefixing, etc. to actually make it work, but for MA this might be a sensible choice for the standard case: for instance, a conceivable set of constant symbols could consist of all identifiers plus all representations of some primitive data types, such as numbers and strings.

B.5.2 Equivalence of MA and traditional ASM

In this section we show how to map a GASM into an MA and vice versa. The translation from MA to GASM is already given by the fact that MA are defined as a GASM with a special kind of structure. The translation from GASM into MA allows to use the MA tool for GASM tool support, since the translation does not change the abstraction level. In fact the translation deals only with some semantical details, e.g. the adaption of the different views on boolean and relations, and the modeling of n-ary functions with tuples.

Before we start describing the translation between GASM to MA we remember the different ways booleans and partial functions are treated. In GASM booleans are modeled by two distinct elements *true* and *false* and partial func-

[9] Of course, the names themselves become structured by the way they relate to the different or identical elements of the universe.

tions are modeled by mapping to a third element *undef*. The carrier set of each GASM needs thus at least three distinct elements, *true*, *false*, and *undef*. Differently, in MA exist only two distinct elements, called bottom $\perp$ and top $\top$. $\perp$ is used for partial functions, and as interpretation of false, *true* is represented by $\top$ or any other element in the carrier set. Both GASM and MA are not strict.

Mapping a GASM state into an MA state.

In general the universe $\mathcal{U}$ of objects in a MA consist of at least two elements, one denoted by $\perp$ and the other by $\top$. Since the GASM super-universe S contains at least three elements (*true*, *false*, and *undef*) we need to start with a $\mathcal{U}$ containing a third element. The set of constant symbols **C** of an MA modeling a GASM contains at least the three constants *true*, *false*, and *undef*, and each interpretation $\mathcal{I}$ maps *undef* to the element $\perp$, *true* to the element top, and *false* to the third default element in $\mathcal{U}$. We will no more make a difference between the symbols $\{$ *undef*, *true*, *false* $\}$ and the tree objects representing them, and for our convenience.

Tuples are modeled in MA by free generated elements with a static mapping as follows:

- the associated mapping of the 0-ary tuple () is given by:

  $$\langle ()\ t\rangle \equiv (t)$$

  where (t) is the free generated one-tuple.

- the associated mapping of a one-tuple is given by:

  $$\langle (t_1)t_2\rangle \equiv (t_1, t_2)$$

  where (t_1, t_2) is a free generated two-tuple.

- for each n $\geq$ 1 the mapping of an n-tuple is given by:

  $$\langle (t_1, \ldots, t_n)t_{n+1}\rangle \equiv (t_1, \ldots, t_n, t_{n+1})$$

If mapping a concrete GASM A into a MA B, all elements of S are included into $\mathcal{U}$ and all symbols of the vocabulary of A are included into the constant symbols **C** of B, and for each of them a new element being its interpretation is included into $\mathcal{U}$. In other words, $\mathcal{U}$ consists of the disjoint union of $\{\perp, \top, \mathit{false}\}$, the super-universe S, the elements interpreting the GASM functions, and the above introduced tuples.

We need to make a case distinction between functions and relations in GASM. The interpretation of each n-ary *function* f in structure A, i.e. f^A, is reflected in $B's$ interpretation of σ, i.e. σ^B:

$$(f^A(o_1, \ldots, o_n) = o_0) \Leftrightarrow (\sigma^B\ \mathcal{I}(f)\ (o_1, \ldots, o_n) = o_0)$$

An n-ary *relation* r in a GASM is returning either *true* or *false*. To make everything fit together we reflect the interpretation of each r as follows:

$$\begin{array}{rcl} (r^A(o_1,\ldots,o_n) = \textit{false}) & \Leftrightarrow & (\sigma^B\,\mathcal{I}(r)\;(o_1,\ldots,o_n) = \bot) \\ & \wedge & \\ (r^A(o_1,\ldots,o_n) = \textit{true}) & \Leftrightarrow & (\sigma^B\,\mathcal{I}(r)\;(o_1,\ldots,o_n) = \top) \end{array}$$

Now we need two different wrappings. One is needed to get back the original *true*,*false* results of a relational term. The second is needed to map such results back into the $\bot$,$\top$ model in MA.

Lets thus assume two constants W_1 and W_2 such that:

$$\begin{array}{rcll} \langle W_1\ \bot\rangle & \equiv & \textit{false} & \\ \langle W_1\ x\rangle & \equiv & x, & \textit{where } x \neq \bot \\ \langle W_2\ \textit{false}\rangle & \equiv & \bot & \\ \langle W_2\ x\rangle & \equiv & x, & \textit{where } x \neq \textit{false} \end{array}$$

For equality, the usual MA equality can be used, the logical operations in GASM are mapped into MA like normal binary relations.

Remark on reachability

of course the mappings associated with the tuples and the wrappings W_1 and W_2 must be excluded from the definition of reachability.

Mapping a GASM rule into an MA rule

We define now a transformation $\mathcal{T}$ from GASM rules to MA rules. For notational convenience we leave away the $\langle$ and $\rangle$ whenever the situation is clear.

Terms For all function symbols f, the subterms must be transformed:

$$\mathcal{T}[f(t_1,\ldots,t_n)] =_{def} f(\mathcal{T}[t_1],\ldots,\mathcal{T}[t_n])$$

For all relation symbols r, in addition the term is wrapped with W_1:

$$\mathcal{T}[r(t_1,\ldots,t_n)] =_{def} \langle W_1\ \langle r\ (\mathcal{T}[t_1],\ldots,\mathcal{T}[t_n])\rangle\rangle$$

Updates For all function symbols f, the subterms must be transformed::

$$\mathcal{T}[f(t_1,\ldots,t_n) := t_0] =_{def} \mathcal{T}[f(t_1,\ldots,t_n)] := \mathcal{T}[t_0]$$

For all relation symbols r, in addition the righ-hand-side is wrapped with W_2:

$$\mathcal{T}[r(t_1,\ldots,t_n) := t_0] =_{def} \mathcal{T}[r(t_1,\ldots,t_n)] := \langle W_2\ \mathcal{T}[t_0]\rangle$$

Conditional

$$\mathcal{T}[\textit{if } c \textit{ then } R_1 \textit{ else } R_2 \textit{ endif}] \ =_{def} \textit{if } \langle W_2\ c\rangle \textit{ then } \mathcal{T}[R_1] \textit{ else } \mathcal{T}[R_2] \textit{ endif}$$

Do forall

$$T[\textit{do forall } i \textit{ in } I \quad \textit{Rule enddo}]$$
$$=_{def}$$
$$\textit{do forall } i \textit{ in dom } I \quad \mathcal{T}[\textit{Rule}] \textit{ enddo}$$

Choose

$$T[\textit{choose } i \textit{ in } I \quad \textit{Rule endchoose}]$$
$$=_{def}$$
$$\textit{choose } i \textit{ in dom } I \quad \mathcal{T}[\textit{Rule}] \textit{ endchoose}$$

C

Stärk’s Model of the Imperative Java Core

In this appendix we reproduce with Montages the specification of the imperative core of Java as given by Stärk (203), which is based on Schulte and Börgers Java model (33). Our reproduction shows that their style of describing languages with ASM can be directly used with Montages. Using our framework, the resulting specification is shorter and more visual than the original ASM model. In the Montages solution the textual rules are shortened from 85 lines to 29 lines and the complete control flow is specified graphically. The given reproduction can be directly executed using the Gem-Mex tool.

In the following we only provide the minimal description, in order to allow for a comparison with our alternative, more compositional specification we give in Chapter 14. The descriptions are an extract from a hand-out given to the students.

C.1 Functions

The universe *Abr* contains the unary constructors *break(_)* and *continue(_)* denoting the set of reasons for abrupt completion.

```
universe Abr = {break(_), continue(id)}
```

The universe *Nrm* is the set of normal values, including booleans, integer, ..., and the constant *normal*.

In (203) a dynamic, 0-ary function *pos* and a universe *Pos* are used to keep track of the control. These functions are not needed in our reproduction, since we use FSMs. *pos* corresponds to the current state in the FSM, and *Pos* corresponds to the set of states in the FSM.

```
function loc(_)
```

The dynamic, unary function *loc* assigns values to variables. It is updated in an assignment statement. It is also updated as a side effect during evaluation of assignment expressions. We will refer to *loc* as the local environment.

```
attr val
```

The dynamic attribute *val* is used to store intermediate values of expressions and results of the execution of statements. It assigns normal or abrupt values to the nodes of the AST.

C.2 Expressions

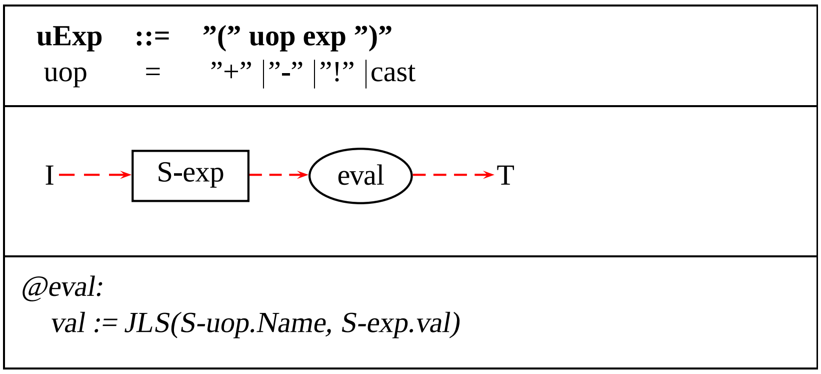
exp = lit | id | uExp | bExp | cExp | asgn

The reproduced specification contains literals, identifiers, unary-, binary, conditional-, and assignment-expressions. The dynamic semantics of these constructs is given by rules that evaluate the expression and assign the result to the attribute *val*.

lit = Boolean | Number

For simplicity only the literal numbers and booleans are considered. Their *val* attribute is statically initialized with their constant value. Their FSM consists of one state without action.

The semantics of a unary expression is given by the Montage in Figure 113 First the *exp*-component is visited resulting in its evaluation. The result is accessed as *S-exp.val* and used to calculate the value of the unary expression. According to (203) the JLS-function contains the Java Language Specification (74) definitions for operators.

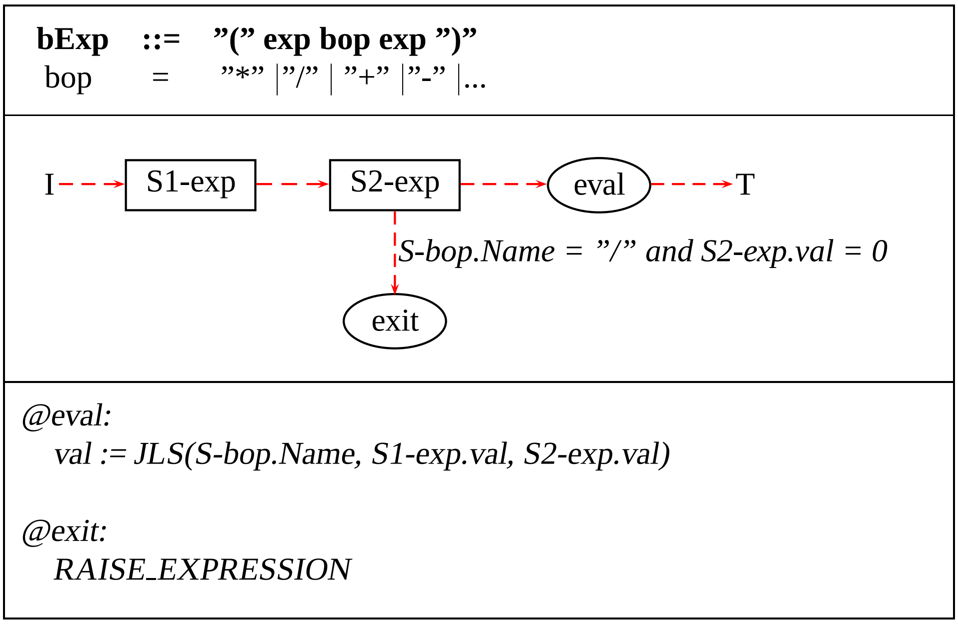


```
uExp  ::=  "(" uop exp ")"
uop    =   "+" |"-" |"!" |cast
```

```
@eval:
  val := JLS(S-uop.Name, S-exp.val)
```

Fig. 113: The uExp Montage.

In a similar way binary expressions are evaluated, see Figure 114. In the case of division by zero, the firing condition guides the FSM in the *exit* state, otherwise the *eval*-state is reached. In the *exit* state execution is stopped abruptly.



```
bExp  ::=  "(" exp bop exp ")"
bop    =   "*" |"/" | "+" |"-" |...
```

```
@eval:
  val := JLS(S-bop.Name, S1-exp.val, S2-exp.val)

@exit:
  RAISE_EXPRESSION
```

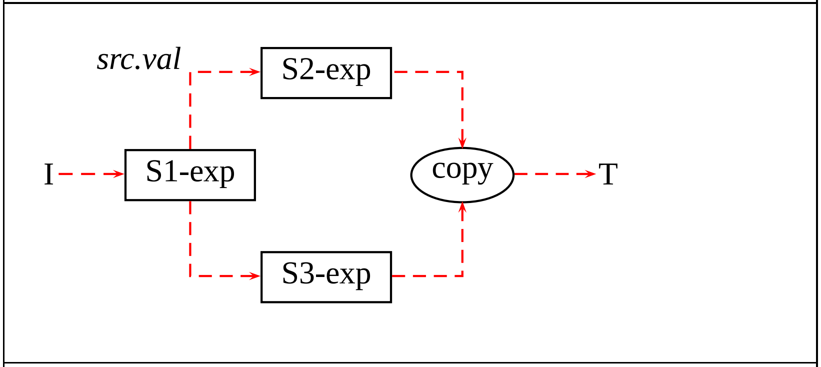
Fig. 114: The bExp Montage.

The condition expression *cExp* is given in Figure 115. After the evaluation of the first expression, depending on their value, control is passed either to the second or third expression. The three different expressions are referenced as *S1-expr*, *S2-expr*, and *S3-expr*, respectively. The condition whether to choose the second or third expression is formalized as *src.val*. The term *src* denotes the source of a control arrow. Thus in Figure A115 the firing-condition *src.val* is equivalent to *S1-val.val*. As a very convenient feature the term *src* can as well be used within transition rules. In the later case, *src* denotes the source of the control arrow that has been used to reach the current state. This fact is used in the *copy* transition rule

```
val := src.val
```

where the value of the evaluated expression is copied as value of the conditional expression.

cExp ::= ”(” exp ”?” exp ”:” exp ”)”

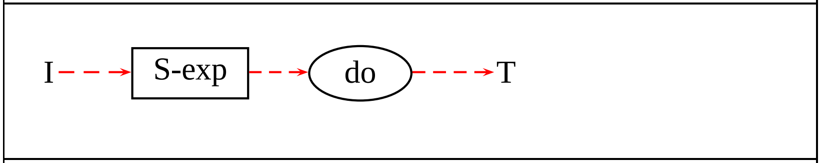


@copy:
val := src.val

Fig. 115: The cExp Montage.

The Montage of an assignment is given in Figure 116. The *do*-action updates the value of the variable S-id.Name in the local environment to the value of S-exp. Further the value of of the assignment is set to the value of S-exp.

asgn ::= ”(” id ”=” exp ”)”



@do:
loc(S-id.Name) := S-exp.val
val := S-exp.val

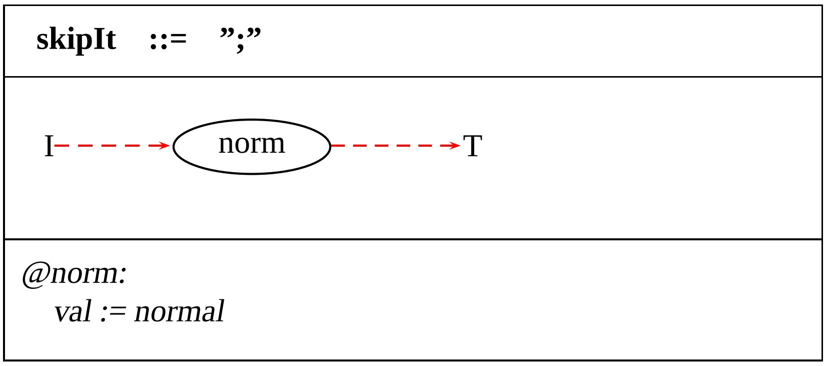
Fig. 116: The asgn Montage.

C.3 Statements

A total of 8 different statements is given:

stm = skipIt | asgnStm | ifStm | whileStm |
labeledStm | breakStm | continueStm | block

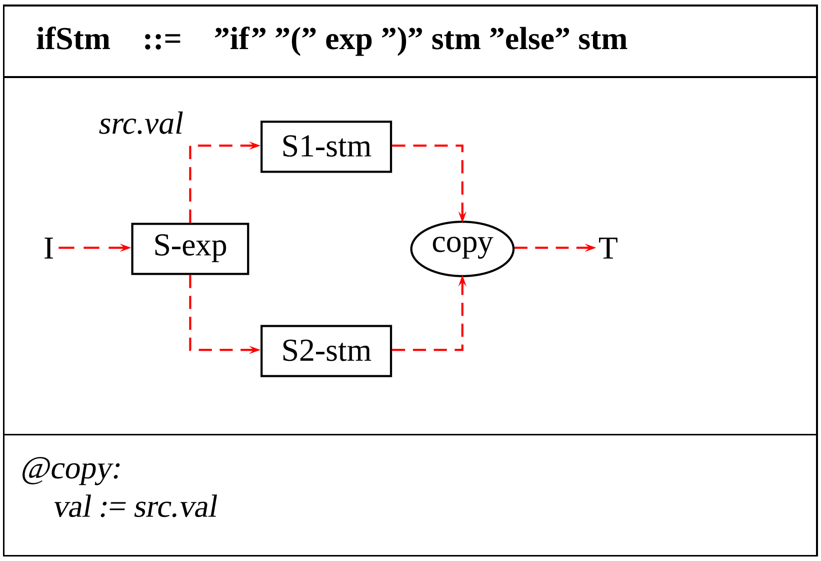
The Montages for the skip (Figure 117), the if- (Figure 118), and the assignment statement (Figure 119) are self explaining. The edges in the while statement (Figure 120) repeated the execution of the statement-component, as long as the value of the expression-component evaluates to true. Another possibility to exit the loop is, if the value of the statement-component evaluates to an abrupt-constructor. If the loop is left, the *copy*-action sets the value of the while-statement to the value of the last executed construct. In the *norm*-state, non-abrupt values are reset to *normal*.



skipIt ::= ";"

@norm:
val := normal

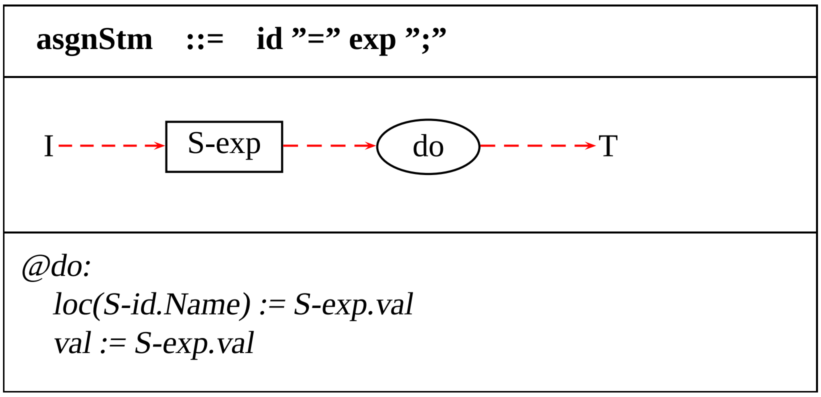
Fig. 117: The skipIt Montage.



ifStm ::= "if" "(" exp ")" stm "else" stm

@copy:
val := src.val

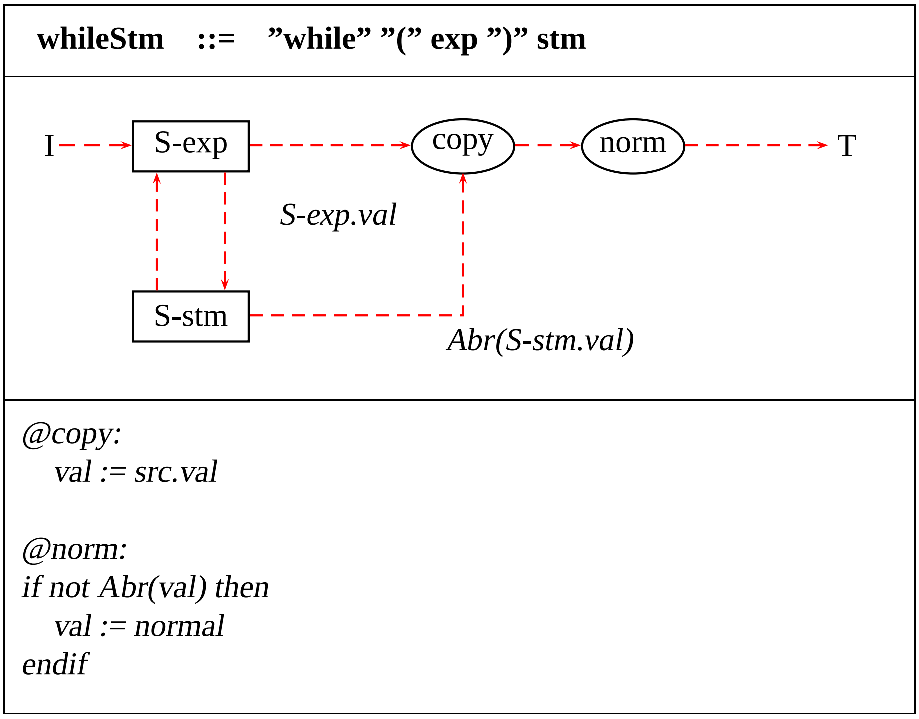
Fig. 118: The ifStm Montage.



asgnStm ::= id ”=” exp ”;”

```
@do:
   loc(S-id.Name) := S-exp.val
   val := S-exp.val
```

Fig. 119: The asgnStm Montage.



whileStm ::= ”while” ”(” exp ”)” stm

```
@copy:
   val := src.val

@norm:
if not Abr(val) then
   val := normal
endif
```

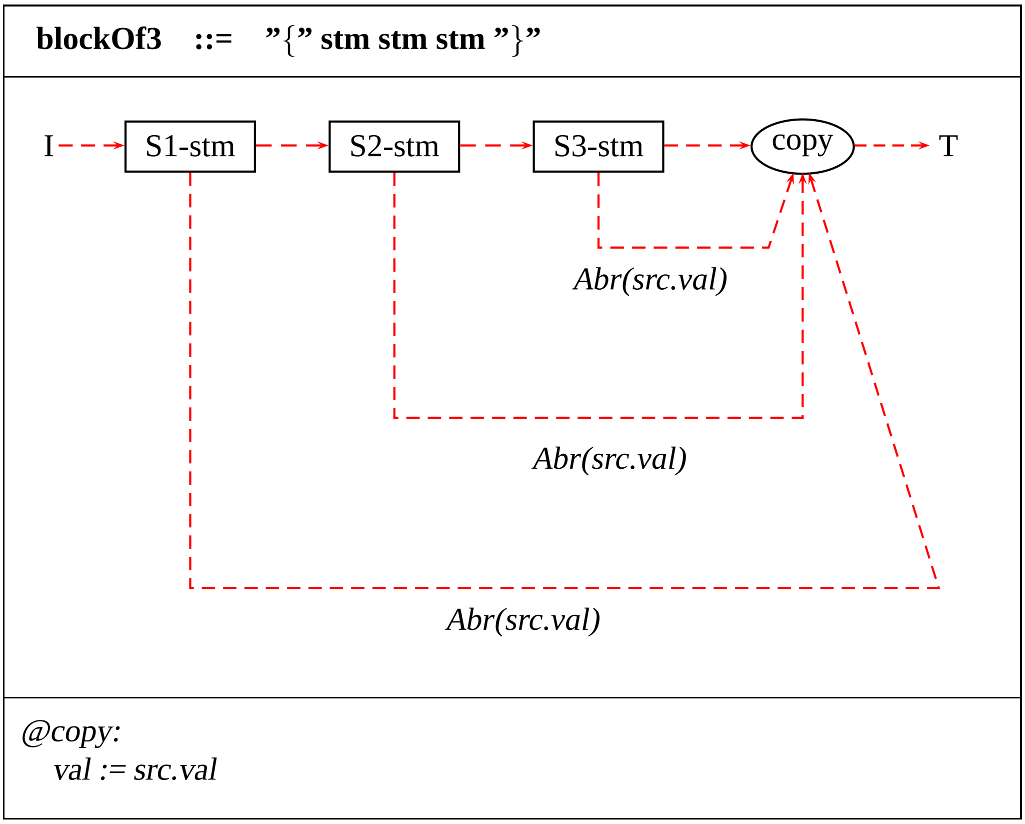
Fig. 120: The whileStm Montage.

The Montages for the break and the continue statements correspond to the literal expressions. Their value is statically initialized with the corresponding constructor terms. The EBNF rules are

breakStm	::=	“break” id “;”
continueStm	::=	“continue” id “;”

The value of the break-statement is initialized to *break(S-id.Name)* and value of the continue-statement is initialized to *continue(S-id.Name)*.

In Figure 121 a block-statement for a fixed block length of 3 is shown. In case of an abrupt completion, for instance *break(_)* or *continue(_)*, the default flow is overruled by the control arrows with the condition *Abr(src.val)*. In the copy-state the the value of the last executed statement is passed as value of the block.



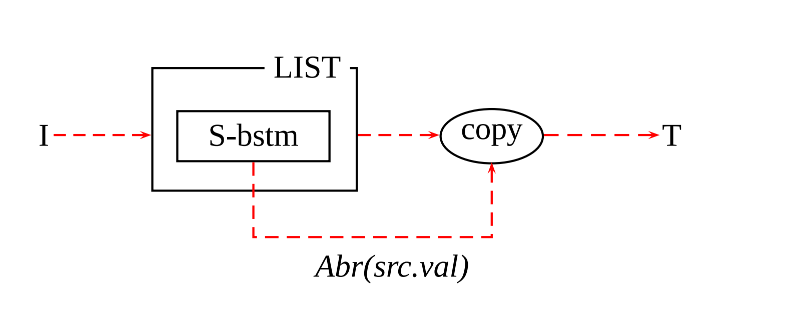
blockOf3 ::= ”{” stm stm stm ”}”

@copy:
val := src.val

Fig. 121: The blockOf3 Montage.

In Figure 122) the Montages for the block-statement with variable length is given, using the List box. The previously shown fixed-length block is an example how such a List box works: the members of the list are linked sequentially be default-arrows. An arrow leaving from the element inside a list corresponds to a family of arrows, one for each member.

block ::= ”{” { **bstm** } ”}”
bstm = stm | var

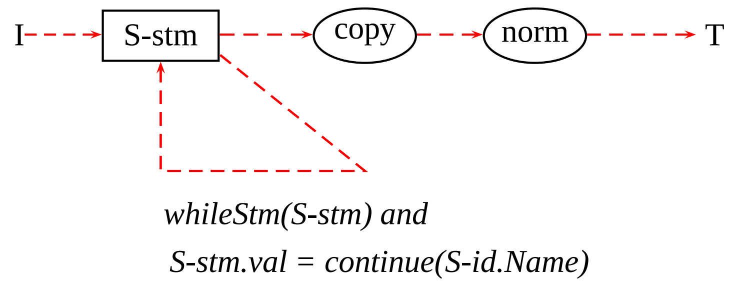


@copy:
  val := src.val

Fig. 122: The block Montage.

The labeled statement (Figure 123) is used to catch the abrupt completions of its statement component. In case of a continue-completion matching the label, and the statement component being a while loop, control is passed again to the statement component. This case is covered by the arrow leaving and entering the *S-stm* box. Otherwise the usual *copy*-state recovers the value of the statement-component. In the *norm*-state, the value is reset to *normal*, if the statement-value was a break with a matching label.

labeledStm ::= id ":" stm



```
@copy:
   val := src.val

@norm:
   if S-stm.val = break(S-id.Name) then
      val := normal
   endif
```

Fig. 123: The labeledStm Montage.

D

Type System of Java

As example for the use of static semantics technology we show the type system of the Java programming language. For examples we refer to the Java language specifications, editions 1 (74) and 2 (75). The following descriptions are minimal extracts from an executable version running on the Gem-Mex system. A detailed discussion would include a detailed discussion of Java typing, a topic which goes beyond the scope of this thesis.

D.1 Reference Types

In Java there are primitive types and reference types. Reference types are classes, interfaces, and arrays. Here we introduce classes and interfaces.

Our Java model identifies class and interface types with the syntax-tree nodes being the declarations of them. The same technique has been used in a number of ASM models of object-oriented languages (130) and will be used in Section 12. This approach has several advantages, among others the ease of animating typing annotations, and the possibility to ”reload” new versions of a class, without stopping the program; in that case one has simply two copies of the same class, one AST being the old version, used as type of all existing instances of the class, and a new version, a second AST, which will be used as type for new instances to be created. Further its the ideal bases to model advanced features like *inner classes*.

Gram. 19:

```
program               ::=  { unit } body
unit                  ::=  { classModifier } classOrInterface
classOrInterface      =    classDeclaration | interfaceDeclaration
classModifier         =    “public” | “abstract” | “final”
classDeclaration      ::=  “class” typeId [“extends” superId]
                           [“implements” interfaceId { “,” interfaceId }]
                           “{”
                           {memberDeclaration}
                           “}”
superId               =    typeRef
interfaceId           =    typeRef
typeRef               =    Ident
interfaceDeclaration  ::=  “interface” typeId
                           [“extends” interfaceId “,” interfaceId]
                           “{”
                           {interfaceMemberDeclaration}
                           “}”
```

The start symbol *program* produces a list of *units* and a body. A unit is a class or interface declaration together with a list of modifiers. The attribute *signature* is used to unify access to the names of units.

Attr. 3:

```
unit:
  attr signature == S-classOrInterface.signature

classDeclaration:
  attr signature == S-typeId.Name

interfaceDeclaration:
  attr signature == S-typeId.Name
```

Within a class or interface declaration, the function *enclosing(_,_)* (ASM 20, Section 5.3.2) together with the derived set *TypeDecl* can be used to refer to the enclosing type.

Decl. 17:

```
derived function TypeDecl ==
  {"classDeclaration","interfaceDeclaration"}
```

The term *n.enclosing(TypeDecl)* denotes the least enclosing reference type.

Static Typing

The attribute *staticType* is defined for types, where its definition is the identity, type references being used in different declarations, statements, and expressions of Java. Further each Java expression has a static type, which is used as basis for type checking and for evaluating dynamic typing.

Attr. 4:

```
classDeclaration:
  attr staticType == self

interfaceDeclaration:
  attr staticType == self
```

Instances of *program* have the attribute *declTable(_)* for looking up the class and interface declarations, given their name.

Attr. 5:

```
program:
  attr declTable(uRef) ==
  (choose u in sequence S-unit:
    u.signature = uRef).S-classOrInterface
```

Type references can determine their static type looking up the declTable of the least enclosing program or package instance. Here we abstract from packages.

Attr. 6:

```
typeRef:
  attr staticType ==
    enclosing({''program''}).declTable(signature)
  attr signature == Name
```

Modifiers

Instances of *unit, classDeclaration, memberDeclaration, fieldRest, methodRest, interfaceDeclaration* and *interfaceMemberDeclaration* can have modifiers. Possible modifiers for classes and interfaces are *public, final,* and *abstract*. Methods and fields may as well be *protected* or *private*, and finally fields may have the modifier *static*. The attribute *hasModifier(_)* is used to test for modifier. Its definition contains some parts related to the implicit *abstract* modifier.

Attr. 7: hasModifier(_)

```
unit:
  attr hasModifier(mStr) ==
         (exists M in sequence S-classModifier: M.Name = mStr)

classDeclaration:
  attr hasModifier(mStr) ==
      Parent.hasModifier(mStr)
   OR (   (mStr = "abstract") AND isAbstract)

interfaceDeclaration:
  attr hasModifier(mStr) ==
           mStr = "abstract"
        OR Parent.hasModifier(mStr)
```

A special case is the modifier *abstract* Class declaration are implicitly abstract, if they have at least one abstract member, or if there is a visible abstract method, which is not implemented by another visible method overriding the first one.

Attr. 8: isAbstract

```
attr isAbstract ==
        (   (exists mDec in sequence S-memberDeclaration:
                    (mDec.methodDeclaration)
                AND (mDec.hasModifier("abstract")))
        OR  (exists mDec in NODE:
                  mDec.methodDeclaration
              AND mDec.hasModifier("abstract")
              AND visible(mDec)
              AND (not (exists m2Dec in NODE:
                             m2Dec.methodDeclaration
                      AND m2Dec.signature = mDec.signature
                      AND (not (m2Dec.hasModifier("abstract")))
                      AND
    m2Dec.enclosing(Scope).subtypeOf(mDec.enclosing(Scope))
                                    AND visible(m2Dec)))))
```

Accessibility
A type A is accessible from another type B, if either A has modifier “public”, or both types are defined in the same *program*. The attribute *accessibleFrom(_)* is defined as follows.

Attr. 9: accessibleFrom(_)

```
unit:
attr accessibleFrom(tDec) ==
      (enclosing({"program"})) = (tDec.enclosing({"program"}))
   OR hasModifier("public")

classDeclaration:
  attr accessibleFrom(tDec) == Parent.accessibleFrom(tDec)

interfaceDeclaration:
  attr accessibleFrom(tDec) == Parent.accessibleFrom(tDec)
```

D.2 Subtyping

The subtyping relation is based on the direct super classes and direct interfaces. The direct super class is denoted in the “extends”-clause and the direct interfaces are denoted by the “implements”-clause. A class without extends clause has the direct super class *Object*.

Decl. 18: `constructor Object`

The definitions for direct super class and direct interfaces are given as follows.

Attr. 10:

```
classDeclaration:
  attr directSuperClass == --JLSv1, 8.1.3;line1-2
    (if S-superId.NoNode
       then Object
       else S-superId.staticType)

  attr directInterface(iDec) ==
    (exists iRef in sequence S-interfaceId:
       iDec = iRef.staticType)

interfaceDeclaration:
  attr directInterface(iDec) ==
    (exists iRef in sequence S-interfaceId.Children:
       iDec = (iRef.staticType))
```

Subtyping is basically the transitive closure over the relations *directSuperClass* and *directInterface*.

Attr. 11: subtypeOf(_)

```
classDeclaration:
  attr subtypeOf(tDec) ==
                 (self = tDec)
                 OR
                  ((directSuperClass != Object) AND
                   directSuperClass.subtypeOf(tDec))
                 OR
                  (exists iDec in interfaceDeclaration:
                      (directInterface(iDec)
                       AND iDec.subtypeOf(tDec)))

interfaceDeclaration:
  attr subtypeOf(tDec) ==
  --SPECIALIZATION FROM classDeclaration
                 (self = tDec)
               OR
                 (exists iDec in interfaceDeclaration:
                   directInterface(iDec)
                   AND iDec.subtypeOf(tDec))
```

D.3 Members

Classes and interfaces are characterized by a number of members. Members can be fields or methods. Here we use a dummy definition for methods to shorten the definitions.

Gram. 20: *memberDeclaration* ::== {*modifier*} *returnType idOrMethId*
fieldOrMethodRest

```
interfaceMemberDeclaration
                        ::=  returnType id fieldOrMethodRest
modifier                 =   “public” | “protected” | “private”
                             | “final” | “static” | “abstract”
returnType               =   voidType | type
idOrMethId               =   Ident | methId
fieldOrMethodRest        =   fieldRest | methodRest
fieldRest               ::=  [“=” exp] {“,” additionalFieldDeclaration}“;”
additionalFieldDeclaration
                        ::=  Ident [“=” exp]
methodRest              ::=  “(”“)” body
```

The attributes *fieldDeclaration* and *methodDeclaration* are used to check whether a member is a field or a method.

Attr. 12:

```
memberDeclaration, interfaceMemberDeclaration:
  attr fieldDeclaration   == S-fieldOrMethodRest.fieldRest
  attr methodDeclaration  == S-fieldOrMethodRest.methodRest
```

Static Typing

staticType denotes the type of the member, *envType* the enclosing class or interface declaration.

Attr. 13:

```
memberDeclaration:
  attr staticType == S-returnType.staticType
  attr envType    == enclosing(TypeDecl)

interfaceMemberDeclaration:
  attr staticType == S-returnType.staticType
  attr envType    == enclosing(TypeDecl)
```

Modifiers

As in the case of types, modifiers of members denote special properties of them. Some of them are given explicitly, by the modifier-sequence, and others, like “abstract”, may be derived.

Attr. 14: hasModifier(_)

```
memberDeclaration:
  attr hasModifier(mStr)==
       (exists m2Str in sequence S-modifier:
          m2Str.Name = mStr)
    OR S-fieldOrMethodRest.hasModifier(mStr)

interfaceMemberDeclaration:
  attr hasModifier(mStr) ==
           (mStr isin {"public","final"})
        OR (mStr = "abstract"
            AND S-fieldOrMethodRest.methodRest)
        OR (mStr = "static"
            AND S-fieldOrMethodRest.fieldRest)

fieldRest:
  attr hasModifier(mStr) == false

methodRest:
  attr hasModifier(mStr) ==
    (mStr = "abstract") AND (S-body.empty)
```

Accessibility

Accessibility determines whether a member m is accessible from a type t. Formally this fact is written as

$$m.\textit{accessibleFrom}(t)$$

Accessibility of members is a precondition for *visibility*, which is in turn a condition for a member being present in the declaration table *declTable* of a Java type.

A member is accessible, if it is public, or if it is private and the type from which it is accessed is the same as the type in which it is declared, or if it is not private, and the types it is accessed from and where it is declared in are in the same package, or it is protected, and the type it is accessed from is a subtype of the type it is declared in.

Attr. 15: accessibleFrom(_)

```
memberDeclaration, interfaceMemberDeclaration:
  attr accessibleFrom(tDec) ==
      hasModifier("public")
   OR (hasModifier("private") and
       (envType  = tDec))
   OR ((not hasModifier("private")) and
       (tDec.enclosing({"package"})
        = enclosing({"package"})))
   OR (    hasModifier("protected")
       AND (tDec != Object)
       AND (tDec.subtypeOf(envType)))
```

D.4 Visibility and Reference of Members

A member m is visible in type t, formally $t.\textit{visible}(m)$, if it is a direct member or the following three conditions hold. First, m is accessible from type t, second[1] there exists no other member with the same name, being a direct member of t, and third, either m is visible in the direct super-class of t, or there exists a direct interface of t where m is visible.

[1] The third condition in the formula Attr. 16.

Attr. 16: visible(_)

```
classDeclaration:
  attr visible(mDec) ==
      directMember(mDec)
   OR (   mDec.accessibleFrom(self)
      AND (   (directSuperClass != Object
              AND directSuperClass.visible(mDec))
          OR  (exists iDec in interfaceDeclaration:
                   directInterface(iDec)
                AND iDec.visible(mDec)))
      AND ( not (exists m2Dec in NODE:
                     directMember(m2Dec)
                  AND m2Dec.signature = mDec.signature)))

interfaceDeclaration:
  attr visible(mDec) ==
      directMember(mDec)
   OR (   mDec.accessibleFrom(self)
      AND (exists iDec in interfaceDeclaration:
                directInterface(iDec)
            AND iDec.visible(mDec))
      AND ( not (exists m2Dec in NODE:
                     directMember(m2Dec)
                  AND m2Dec.signature = mDec.signature)))
```

D.5 Reference of Static Fields

For the reference to static fields the above function *visible* is now used. A static field f is in the *declTable* of a type t if there exists a unique member among all members of all types with the name f being visible in t. For the reference to methods, the definition of *visible* is enough.

Attr. 17: declTable(_)

```
classDeclaration, interfaceDeclaration:
  attr declTable(mRef) ==
-- only needed for fields, for methods, visible is enough
      (choose unique mDec in NODE:
            (mDec.memberDeclSet)
        AND (mDec.signature = mRef)
        AND visible(mDec))
```

Bibliography

[1] *Proc. First USENIX Conference on Domain Specific Languages*, Santa Barbara, California, October 1997.

[2] Alcatel, I-Logix, Kennedy-Carter, Kabira, Project Technology, Rational, and Telelogic AB. Action semantics for the uml, omg ad/2001-08-04, response to omg rfp ad/98-11-01, August 2001.

[3] V. Ambriola, G. E. Kaiser, and R. J. Ellison. An action routine model for ALOE. Technical Report CMU-CS-84-156, Department of Computer Science, Carnegie Mellon University, August 1984.

[4] M. Anlauff. Aslan - programming in abstract state machines. A small stand–alone ASM interpreter written in C, ftp://ftp.first.gmd.de/pub/gemmex/Aslan.

[5] M. Anlauff. XASM - An Extensible, Component-Based Abstract State Machines Language. In Gurevich et al. (86), pages 69–90.

[6] M. Anlauff, A. Bemporad, S. Chakraborty, P. W. Kutter, D. Mignone, M. Morari, A. Pierantonio, and L. Thiele. From ease in programming to easy maintenance: Extending dsl usability with montages. Technical Report 83, ETH Zurich, Institute TIK, 1999.

[7] M. Anlauff, S. Chakraborty, P. W. Kutter, A. Pierantonio, and L. Thiele. Generating an Action Notation environment from montages descriptions. *Software Tools and Technology Transfer, Springer*, (3):431–455, 2001.

[8] M. Anlauff and P. W. Kutter. The xasm open source project. http://www.xasm.org, 2002.

[9] M. Anlauff, P. W. Kutter, and A. Pierantonio. The Gem-Mex tool homepage. URL: `http://www.gem-mex.com`.

[10] M. Anlauff, P. W. Kutter, and A. Pierantonio. Formal aspects of and development environments for Montages. In M. Sellink, editor, *Proc. 2nd International Workshop on the Theory and Practice of Algebraic Specifications*, Workshops in Computing, Amsterdam, 1997. Springer Verlag.

[11] M. Anlauff, P. W. Kutter, and A. Pierantonio. Aslan: Programming with asms. Presentation at the Second Cannes ASM Workshop 1998, June 1998.

[12] M. Anlauff, P. W. Kutter, and A. Pierantonio. Enhanced control flow graphs in Montages. In D. Bjorner, M. Broy, and A.V. Zamulin, editors, *Perspective of System Informatics*, volume 1755 of *Lecture Notes in Computer Science*, pages 40 – 53. Springer Verlag, 1999.

[13] M. Anlauff, P. W. Kutter, A. Pierantonio, and A. Sünbül. Using domain-specific languages for the realization of component composition. In T. Maibaum, editor, *Fundamental Approaches to Software Engineering (FASE 2000)*, volume 1783 of *Lecture Notes in Computer Science*, pages 112 – 126, 2000.

[14] G. Arango. Domain analysis: From art form to engineering discipline. *ACM SIGSOFT Engineering Notes*, 14(3):152–159, May 1989. 5th Int. Workshop on Software Specification and Design.

[15] M. A. Ardis, N. Daley, D. Hoffman, H. Siy, and D. M. Weiss. Software product lines: a case study. *Software Practice and Experience*, 30(7):825–847, June 2000.

[16] M. A. Ardis and J. A. Green. Successful introduction of domain engineering into software development. *Bell Labs Technical Journal*, pages 10 – 20, July-September 1998.

[17] E. Astesiano and E. Zucca. D-oids: a model for dynamic data-types. *Mathematical Structures in Computer Science*, 5(2):257–282, June 1995.

[18] R. Bahlke and G. Snelting. Design and structure of a semantics-based programming environment. *International Journal of Man-Machine Studies*, 37(4):467 – 479, October 1992.

[19] R. A. Ballance, S. L. Graham, and M. L. Van De Vanter. The Pan language-based editing system. *ACM Transactions on Software Engineering and Methodology*, 1(1):95 – 127, January 1992.

[20] A. Basu, M. Hayden, G. Morrisett, and T. von Eicken. A language-based approach to protocol construction. In *Pcoc. DSL'97, ACM SIGPLAN Workshop on Domain-Specific Languages*, Univ. of Ill. Comp. Sci. Report, pages 1–15, 1997.

[21] D. Batory, B. Lofaso, and Y. Samaragdakis. Jts: tools for implementing domain-specific languages. In *Proc. of 5th Int. Conf. on Software Reuse*, pages 143–153. IEEE Computer Society Press, June 1998.

[22] A. Beetem and J. Beetem. Introduction to the galaxy language. *IEEE Software*, May 1989.

[23] D. Bell and M. Parr. Spreadsheets: a research agenda. *SIGPLAN notices*, 28(9):26–28, September 1993.

[24] J. L. Bentley. Programming pearls: Little languages. *Communications of the ACM*, 29(8):711–721, 1986.

[25] OMB Architecture Board. Model-driven architecture: A technical perspective. ftp://ftp.omg.org/pub/docs/ab/01-02-01.pdf, 2001.

[26] B. Boehm. Making RAD work for your project. *IEEE Computer*, pages 113–119, March 1999.

[27] E. Börger and I. Durdanović. Correctness of Compiling Occam to Transputer Code. *Computer Journal*, 39(1):52 – 92, 1996.

[28] E. Börger, I. Durdanović, and D. Rosenzweig. Occam: Specification and Compiler Correctness. Part I: The Primary Model. In *IFIP 13th World Computer Congress, Volume I: Technology/Foundations*, pages 489 – 508. Elsevier, Amsterdam, 1994.

[29] E. Börger and J. Huggins. Abstract state machines 1988 – 1998: Commented ASM bibliography. In H. Ehrig, editor, *EATCS Bulletin, Formal Specification Column*, number 64, pages 105 – 127. EATCS, February 1998.

[30] E. Börger and D. Rosenzweig. A mathematical definition of full prolog. In *Science of Computer Programming*, volume 24, pages 249–286. North-Holland, 1994.

[31] E. Börger and D. Rosenzweig. *The WAM - Definition and Compiler Correctness*, chapter 2, pages 20 – 90. Series in Computer Science and Artificial Intelligence. Elsevier Science B.V.North Holland, 1995.

[32] E. Börger and J. Schmid. Composition and submachine concepts for sequential ASMs. In P. Clote and H. Schwichtenberg, editor, *Gurevich Festschrift CSL 2000*, LNCS. Springer-Verlag, 2000. to Appear.

[33] E. Börger and W. Schulte. A programmer friendly modular definition of the semantics of Java. In J. Alves-Foss, editor, *Formal Syntax and Semantics of Java*, volume 1523 of *Lecture Notes in Computer Science*. Springer Verlag, 1998.

[34] P. Borra, D. Clement, T. Despeyroux, J. Incerpi, G. Kahn, B. Lang, and V. Pascual. CENTAUR: The System. Technical Report 777, INRIA, Sophia Antipolis, 1987.

[35] P. Borras, D. Clément, T. Despeyroux, J. Incerpi, G. Kahn, B. Lang, and V. Pascual. Centaur: The system. In *Proc. SIGSOFT 88: 3rd. Annual Symposium on Software Development Environments*, Boston, November 1988. ACM, New York.

[36] G. H. Campbell. Domain-specific engineering. In *Proceedings of the Embedded Systems Conference*, San Jose, September 1997. Miller Freeman, Inc., San Francisco, www.mfi.com.

[37] G. H. Campbell, S. Faulk, and D. M. Weiss. Introduction to synthesis. Technical Report INTRO-SYTNTHESIS-PROCESS-90019-N, Software Productivity Consortium Services Corporation, 2214 Rock Hill Road, Herndon, Virginia 22070, 1990.

[38] M. Caplinger. *A Single Intermediate Language for Programming Environments*. PhD thesis, Department of Computer Science, Rice University, Houston, Texas, 1985. Available as COMP TR85-28.

[39] R. J. Casimir. Real programmers don't use spreadsheets. *SIGPLAN notices*, 27(6):10–16, June 1992.

[40] S. C. Cater and J. K. Huggins. An ASM dynamic semantics for standard ML. In Gurevich et al. (86), pages 203–223.

[41] M. Chandy and J. Misra. *Parallel Program Design: A Foundation*. Addison-Wesley, Reading, MA, 1988.

[42] N. Chomsky. Three Models for the Description of Language. *IRE Trans.on Information Theory*, IT–2:113 – 124, 3 1956.

[43] T. Clark, A. Evans, and S. Kent. Engineering modelling languages: A precise meta-modelling approach. In R.-D. Kutsche and H. Weber, editors, *Fundamental Approaches to Software Engineering, 5th International Conference, FASE 2002, held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2002, Grenoble, France, April 8-12, 2002, Proceedings*, volume 2306 of *Lecture Notes in Computer Science*, pages 159–173. Springer, 2002.

[44] J. G. Cleaveland. Building application generators. *IEEE Software*, pages 25–33, July 1988. also reprinted in Domain Analysis and Software System Modeling by Prieto-Diaz and Arango 1991.

[45] J. G. Cleaveland. *Program Generators with XML and Java*. The Charles F. Goldfarb Series on Open Information Managment. Prentice Hall PTR, NJ, 2001.

[46] C. Consel and O. Danvy. Tutorial notes on partial evaluation. In ACM Press, editor, *20th ACM Symposium on Principles of Programming Languages*, pages 493–501, Chaleston, South Caroline, 1993.

[47] C. Consel and R. Marlet. Architecturing software using a methodology for language development. In C. Palamidessi, H. Glaser, and K. Meinke, editors, *PLILP/ALP'98 Proc. of the 10th Int. Symposium on Programming Languages, Implementations, Logics, and Programs*, volume 1490, pages 170–194. Springer, Heidelberg, September 1998.

[48] J. O. Coplien. *Multi-Paradigm Design for C++*. Addison-Wesley, Reading, MA, 1999.

[49] J. R. Cordy and C. D. Halpern. Txl: a rapid prototyping system for programming language dialects. In *Proc. IEEE 1988 Int. Conf. on Computer Languages*, pages 280–285, 1988.

[50] D. Cuka and D. M. Weiss. Specifying executable commands: An example of fast domain engineering. *Comm. of the ACM*, 2001.

[51] K. Czarnecki and U. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, Reading, MA, 2000.

[52] P. Dauchy and M. C. Gaudel. Algebraic Specification with Implicit States. Technical report, Univ. Paris–Sud, 1994.

[53] M. DeAddio and A. Kramer. *The Handbook of Fixed Income Technology*, chapter An Object Oriented Model for Financial Instruments, pages 269–301. The Summit Group Press, 1999.

[54] G. Del Castillo. Towards comprehensive tool support for abstract state machines: The asm workbench tool environment and architecture. In D. Hutter, W. Stephan, P. Treaverso, and M. Ullman, editors, *Applied Formal Methods – FM-Trends 98*, number 1641 in LNCS, pages 311–325. Springer, 1999.

[55] Ch. Denzler. *Modular Language Specification and Composition*. PhD thesis, Swiss Federal Institute of Technology (ETH), Zürich, 2000.

[56] V. O. Di Iorio. *Avaliação Parcial em Máquinas de Estado Abstratas*. PhD thesis, Departamento de Ciência da Computação da Universidade Federal de Minas Gerais, março 2001. in Portuguese.

[57] V. O. Di Iorio, R. S. Bigonha, and M. A. Maia. A Self-Applicable Partial Evaluator for ASM. In *Proceedings of the ASM 2000 Workshop*, pages 115–130, Monte Veritá, Switzerland, March 2000.

[58] B. DiFranco. Specification of ISO SQL using Montages. Master's thesis, Università di l'Aquila, 1997. in Italian.

[59] E. W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, NJ, 1976.

[60] K.-G. Doh and P. D. Mosses. Composing programming languages by combining action-semantics modules. In Mark van den Brand and Didier Parigot, editors, *Electronic Notes in Theoretical Computer Science*, volume 44. Elsevier Science Publishers, 2001.

[61] V. Donzeau-Gouge, G. Huet, G. Kahn, and B. Lang. Programming environments based on structured editors: The MENTOR experience. In D. R. Barstow, H. E. Shrobe, and E. Sandewell, editors, *Interactive Programming Environments*, chapter 7, pages 128 – 140. McGraw-Hill, New York, 1984.

[62] J.-M. Eber and Risk Awards Editorial Bord. Software product of the year. *Risk Magazine*, 2001.

[63] W. Edwardes. *Key Financial Instruments, understanding and innovating in the world of derivatives*. Financial Times, Prentice Hall, Pearson Education, 2000.

[64] P. D. Edwards and R. S. Rivett. Towards an automative 'safer subset' of c. In P. Daniel, editor, *SAFECOMP'97 16th Int. Conf. on Comp. Safety, Reliability, and Security*. Springer, 1997.

[65] H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification 1: Equations and Initial Semantics*, volume 6 of *EATCS Monographs on Theoretical Computer Science*. Springer-Verlag, Berlin, 1985.

[66] G. Engels, C. Lewerentz, M. Nagl, W. Schafer, and A. Schurr. Building integrated software development environments Part I: Tool specification. *ACM Transactions on Software Engineering and Methodology*, 1(2):135 – 167, April 1992.

[67] D. K. Every. What is the history of vb? www.mackido.com/History/History_VB.html, 1999.

[68] R. E. Faith, L. S. Nyland, and J. F. Prins. KHEPERA: A system for rapid implementation of domain specific languages. In *Proc. First USENIX Conference on Domain Specific Languages* (1).

[69] Russian Institute for System Programming. The mpC website. www.ispras.ru/∼ mpc, 2003.

[70] H. Ganzinger. Modular first-order specifications of operational semantics. In H. Ganzinger and N.D. Jones, editors, *Programs as Data Objects*, volume 217 of *Lecture Notes in Computer Science*. Springer Verlag, 1985.

[71] K. Godel. *The REXX Language*. Prentice-Hall, Englewood Cliffs, NJ, 1985.

[72] J. W. Goguen, J. W. Thatcher, E. G. Wagner, and J. B. Wright. Initial algebra semantics and continuous algebras. *Journal of the ACM*, 24:68–95, 1977.

[73] G. Goos and W. Zimmermann. ASMs and Verifying Compilers. In Gurevich et al. (86), pages 177–202.

[74] Gosling. *The Java Language Specification*. Sun Java Press, 1 edition.

[75] Gosling. *The Java Language Specification*. Sun Java Press, 2 edition.

[76] R. W. Gray, V. P. Heuring, S. P. Levi, A. M. Sloane, and W. M. Waite. Eli: A complete, flexible compiler construction system. *Communications of the ACM*, 35(2):121–131, February 1992.

[77] J. Grosch and H. Emmelmann. A tool box for compiler construction. In D Hammer, editor, *Proceedings of CC'90*, number 477 in LNCS, pages 106–116. Springer Verlag, 1990.

[78] C. A. Gunter. *Semantics of Programming Languages*. Foundations of Computing. The MIT Press, 1992.

[79] Y. Gurevich. A new thesis. *Abstracts, American Mathematical Society*, page 317, August 1985.

[80] Y. Gurevich. Logic and the Challenge of Computer Science. In E. Börger, editor, *Theory and Practice of Software Engineering*, pages 1–57. CS Press, 1988.

[81] Y. Gurevich. Evolving Algebras 1993: Lipari Guide. In E. Börger, editor, *Specification and Validation Methods*. Oxford University Press, 1995.

[82] Y. Gurevich. May 1997 draft of the ASM guide. Technical Report CSE-TR-336-97, University of Michigan EECS Department Technical Report, 1997.

[83] Y. Gurevich. Sequential ASM Thesis. *Bulletin of European Association for Theoretical Computer Science*, (67):93–124, February 1999. Also Microsoft Research Technical Report No. MSR-TR-99-09.

[84] Y. Gurevich. Sequential abstract state machines capture sequential algorithms. *ACM Transaction on Computational Logic*, 1(1):77–111, July 2000.

[85] Y. Gurevich and J. K. Huggins. The semantics of the C programming language. In E. Börger, G. Jäger, H. Kleine Bünig, S. Martini, and M.M. Richter, editors, *Computer Science Logic*, volume 702 of *Lecture Notes in Computer Science*, pages 274–308. Springer Verlag, 1993.

[86] Y. Gurevich, P. W. Kutter, M. Odersky, and L. Thiele, editors. *Abstract State Machines: Theory and Applications*, volume 1912 of *Lecture Notes in Computer Science*. Springer Verlag, 2000.

[87] A. Heberle. *Korrekte Transformationsphase - der Kern korrekter Übersetzer*. PhD thesis, Universität Karlsruhe, 2000.

[88] A. Heberle, W. Löwe, and M. Trapp. Safe reuse of source to intermediate language compilations. Fast Abstract, 9th International Symposium on Software Reliability Engineering, September 1998. http://chillarege.com/issre/fastabstracts/98417.html.

[89] G. Hedin. Reference Attribute Grammarsa. *Informatica*, 24(3):301–318, sep 2000.

[90] J. Heering. Application software, domain–specific languages, and language design assistants. In *Proc. SSGRR'00 Inter. Conf. on Adv. in Infrastructure for Electronic Business, Science and Education on the Internet*, 2000.

[91] J. Heering, G. Kahn, P. Klint, and B. Lang. Generation of interactive programming environments. In *ESPRIT'85: Status Report of Continuing Work, Part I*, pages 467 – 477. North-Holland, 1986.

[92] J. Heering and P. Klint. Semantics of programming languages: A tool-oriented approach. *ACM SIGPLAN Notices*, 35(3), March 2000.

[93] P. Henriques, M. V. Pereira, M. Mernik, M. Lenič, E. Avdičaušević, and V. Žumer. Automatic generation of language-based tools. In Mark van den Brand and Ralf Laemmel, editors, *Electronic Notes in Theoretical Computer Science*, volume 65. Elsevier Science Publishers, 2002.

[94] R. M. Herndon and V. A. Berzins. The realizable benefits of a language prototyping language. *IEEE Transactions on Software Engineering*, 14:803–809, 1988.

[95] C. A. R. Hoare. Proof of a program: Find. *Comm. of the ACM*, 14(1):39–45, 1971.

[96] C. A. R. Hoare. *Hints on programming language design*, chapter 0, pages 31–40. Computer Science Press, 1983. Reprinted from Sigact/Sigplan Symposium on Principles of Programming Languages, Oct. 1973.

[97] J. Huggins. The Abstract State Machine Homepage at Michigan, URL: `http://www.eecs.umich.edu/gasm/`.

[98] J. K. Huggins and W. Shen. The static and dynamic semantics of c: Preliminary version. Technical Report CPSC-1999-1, Computer Science Program, Kettering University, February 1999.

[99] J. W. Janneck. Object-based mapping automata - reference manual. Technical report, Institute TIK, ETH Zürich.

[100] J. W. Janneck. Object-based mapping automata home page. http://www.tik.ee.ethz.ch/ janneck/OMA.

[101] J. W. Janneck and P. W. Kutter. Mapping automata. Technical Report TIK Report 89, Institute TIK, ETH Zürich, Institute TIK, ETH Zurich, June 1998.

[102] J. W. Janneck and P. W. Kutter. Object-based abstract state machines. Technical Report TIK Report 47, Institute TIK, ETH Zürich, Institute TIK, ETH Zurich, 1998.

[103] M. Jazayeri. A simpler construction showing the intrinsically exponential complexity of the circularity problem of attribute grammars. *Journal of the ACM*, 28(4):715–720, 1981.

[104] S. C. Johnson and R. Sethi. yacc: A parser generator. In *Unix Research System Papers. Tenth Edition*. Murray Hill, NJ: AT&T Bell Laboratories, 1990.

[105] C. Jones. End-user programming. *IEEE Computer*, pages 68–70, September 1995.

[106] C. Jones. *Estimating Software Costs*. McGraw-Hill, 1998.

[107] M. A. Jones and L. H. Nakatani. Method to produce application oriented languages. Patent WO9815894, April 1999.

[108] N. Jones, C. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall, 1993.

[109] S. P. Jones, J.-M. Eber, and J. Seward. Composing contracts: an adventure in financial engineering. In *International Conference on Functional Programming*.

[110] G. Kahn. Natural Semantics. In *Proceedings of the Symp. on Theoretical Aspects of Computer Science, Passau, Germany*, 1987.

[111] G. E. Kaiser. *Semantics for Structure Editing Environments*. PhD thesis, Department of Computer Science, Carnegie Mellon University, Pittsburg, Pennsylvania, May 1985.

[112] G. E. Kaiser. Incremental dynamic semantics for language-based programming environments. *ACM Transactions on Programming Languages and Systems*, 11(2):169 – 193, April 1989.

[113] A. Kalinov, A. Kossatchev, A. Petrenko, M. Posypkin, and V. Shishkov. Coverage-driven automated compiler test suite generation. accepted at LDTA 2003, 2002.

[114] A. Kalinov, A. Kossatchev, A. Petrenko, M. Posypkin, and V. Shishkov. Using ASM Specifications for Compiler Testing. In *Abstract State Machines - Advances in Theory and Applications 10th International Workshop, ASM 2003*, volume 2589 of *LNCS*, 2003.

[115] A. Kalinov, A. Kossatchev, M. Posypkin, and V. Shishkov. Using ASM specification for automatic test suite generation for mpC parallel programming language compiler. In *Proceedings of Fourth International Workshop on Action Semantics and Related Frameworks, AS'2002 NS-00-8 Department of Computer Science, University of Aarhus, Technical Report*, pages 96–106, 2002.

[116] S. Kamin, editor. *Proc. First ACM SIGPLAN Workshop on Domain Specific Languages*, Paris, January 1997. Published as University of Illinois at Urbana Champaign Computer Science Report URL: `www-sal.cs.uiuc.edu/~kamin/dsl`.

[117] U. Kastens. Ordered attribute grammars. *Acta Informatica*, 13(3):229–256, 1980.

[118] H. M. Kat. *Structured Equity Derivatives, the definitive guide to exotic options and structured notes*. Wiley Finance, 2001.

[119] J. Kiczales, G. des Riviéres and D. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, Cambridge, MA, 1991.

[120] P. Klint. A meta-environment for generating programming environments. *ACM Transactions on Software Engineering and Methodology*, 2(2):176–201, 1993.

[121] D. Knuth. An empirical study of FORTRAN programs. *Software – Practice and Experience*, 1:105–133, 1971.

[122] D. E. Knuth. Semantics of Context–Free Languages. *Math. Systems Theory*, 2(2):127 – 146, 1968.

[123] B. Kramer and H-W. Schmidt. Developing integrated environments with ASDL. *IEEE Software*, pages 98 – 107, January 1989.

[124] P. W. Kutter. Executable Specification of Oberon Using Natural Semantics. Term Work, ETH Zürich, implementation on the Centaur System (35), 1996.

[125] P. W. Kutter. Integration of the Statecharts in Specware and Aspects of Correct Oberon Code Generation. Master's thesis, ETH Zurich, 1996.

[126] P. W. Kutter. Methods and Systems for Direct Execution of XML Documents. Patent Applications PCT/IB 00/01087, US 09921298, August 2000.

[127] P. W. Kutter. The formal definition of anlauff's extensible abstract state machines. Technical Report 136, ETH Zurich, Switzerland, Institute TIK, June 2002. ftp://ftp.tik.ee.ethz.ch/pub/publications/TIK-Report136.pdf.

[128] P. W. Kutter. Oo xasms executable semantics, xasmmontages-v0.2.tar. http://www.xasm.org, 2002.

[129] P. W. Kutter. Replacing Generation of Interpreters with a Combination of Partial Evaluation and Parameterized Signatures, leading to a Concept for Meta-Bootstrapping. submitted for publication, April 2002.

[130] P. W. Kutter and F. Haussmann. Dynamic Semantics of the Programming Language Oberon. Term work, ETH Zürich, July 1995. A revised version appeared as technical report of Institut TIK, ETH, number 27, 1997.

[131] P. W. Kutter and A. Pierantonio. Montages: Unified static and dynamic semantics of programming languages. Technical Report 118, Universita de L'Aquila, July 1996. As well appeared as technical report Kestrel Institute.

[132] P. W. Kutter and A. Pierantonio. The formal specification of Oberon. *Journal of Universal Computer Science*, 3(5):443–503, 1997.

[133] P. W. Kutter and A. Pierantonio. Montages: Specification of realistic programming languages. *Journal of Universal Computer Science*, 3(5):416 – 442, 1997.

[134] P. W. Kutter, D. Schweizer, and L. Thiele. Integrating formal Domain-Specific Language design in the software life cycle. In D. Hutter, W. Stephan, P. Traverso, and M. Ullmann, editors, *Current Trends in Applied Formal Methods*, volume 1641 of *Lecture Notes in Computer Science*, pages 196 – 212. Springer Verlag, October 1998.

[135] P.W. Kutter. The A4M homepage, URL: `http://www.a4m.biz`.

[136] P.W. Kutter. State transitions modeled as refinements. Technical report, Kestrel Institute, 1996.

[137] D. A. Ladd and J. C. Ramming. Two application languages in software production. In *USENIX Symposium on Very High Level Languages*, pages 169–177, New Mexico, October 1994.

[138] R. Lämmel and C. Verhoef. Cracking the 500-languages problem. *IEEE Software,* 18(6):78–88, November/December 2001.

[139] R. Lämmel and C. Verhoef. Semi-automatic grammar recovery. *Software: Practice and Experience,* 31(15):1395–1438, December 2001.

[140] L. Lamport. The temporal logic of actions. *ACM TOPLAS,* 16(3):872–923, 1994.

[141] P. J. Landin. The next 700 programming languages. *Comm. of the ACM,* 9(3):157–166, May 1966.

[142] C. Larman. Protected variation: The importance of being closed. *IEEE Software,* 18(3):89–91, 2001.

[143] M. E. Lesk and E. Schmidt. Lex - a lexical analyzer generator. In *Unix Research System Papers. Tenth Edition*. Murray Hill, NJ: AT&T Bell Laboratories, 1990.

[144] R. Lipsett, E. Marschner, and M. Shahdad. VHDL- The Language. *IEEE Design & Test of Computers,* 3(2):28–41, 1986.

[145] J.A. Lowell. *Unix Shell Programming*. John Wiley & Sons, 2nd edition, September 1990.

[146] M. Lutz. *Programming Python*. Number ISBN 1-56592-197-6. O'Reilley, 1996.

[147] B. Magnusson, M. Bengtsson, L-O. Dahlin, G. Fries, A. Gustavsson, G. Hedin, S. Minor, D. Oscarsson, and M. Taube. An overview of the Mjolner/ORM environment: Incremental language and software development. In *Proc. Second International Conference TOOLS (Technology of Object-Oriented Languages and Systems)*, pages 635 – 646, Paris, June 1990.

[148] J. Malenfant. Modélisation de la sémantique formelle des langages de programmation en UML et OCL. Rapport de recherche 4499, INRIA, Rennes, Juillet 2002. in French.

[149] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems, Volume 1: Specification*. Springer-Verlag, New York, NY, 1992.

[150] W. May. Specifying complex and structured systems with evolving algebras. In M. Bidoit and M. Dauchet, editors, *Proc. of TAPSOFT'97: Theory and Practice of Software Development, 7th International Joint Conference CAAP/FASE*, number 1214 in LNCS, pages 535–549. Springer, 1997.

[151] R. Medina-Mora. *Syntax-directed Editing: Towards Integrated Programming Environments*. PhD thesis, Carnegie Mellon University, March 1982. Tech. Rep. CMU-CS-82-113.

[152] S.J. Mellor and M.J. Balcer, editors. *Executable UML: A Foundation for Model Driven Architecture*. Addison Wesley Professional, May 2002.

[153] Marjan Mernik, Mitja Lenič, Enis Avdičaušević, and Viljem Žumer. A reusable object-oriented approach to formal specifications of programming languages. *L'Objet*, 4(3):273–306, 1998.

[154] Marjan Mernik, Mitja Lenič, Enis Avdičaušević, and Viljem Žumer. Multiple Attribute Grammar Inheritance. *Informatica*, 24(3):319–328, September 2000.

[155] R. Milner, M. Tofte, and R. Harper. *The Definition of StandardML*. MIT Press, Cambridge, Massachusetts, 1990.

[156] M. Mlotkowski. *Specification and Optimization of Smalltalk Programs*. PhD thesis, Institute of Computer Science, University of Wroclaw, 2001.

[157] J. Morris. *Algebraic Operational Semantics and Modula-2*. PhD thesis, University of Michigan, 1988.

[158] P. D. Mosses. *Action Semantics*. Number 26 in Cambridge Tracts in theoretical Computer Science. Cambridge University Press, 1992.

[159] P. D. Mosses. Modularity in natural semantics (extended abstract). Available at http://www.brics.dk/∼pdm, 1998.

[160] P. D. Mosses. A modular SOS for Action Notation. Research Series BRICS-RS-99-56, BRICS, Department of Computer Science, University of Aarhus, 1999.

[161] L. H. Nakatani, M. A. Ardis, R. O. Olsen, and P. M. Pontrelli. Jargons for domain engineering. In *DSL 99, Domain-Specific Languages*, pages 15–24, 1999.

[162] L. H. Nakatani and L. W. Ruedisueli. Fit programming language primer. Technical Report Memorandum 1264-920301-03TMS, AT&T Bell Laboratories, March 1992.

[163] L.H. Nakatani and M.A. Jones. Jargons and infocentrism. In Kamin (116), pages 59–74. Published as University of Illinois at Urbana Champaign Computer Science Report URL: `www-sal.cs.uiuc.edu/~kamin/dsl`.

[164] P. Naur. Revised report on the algorithmic language algol 60. *Numerical Mathematics*, (4):420–453, 1963.

[165] J. Neighbors. *Software Construction Using Components*. PhD thesis, University of California, Irvine, 1980. Also tech. report UCI-ICS-TR160.

[166] J. Neighbors. The evolution from software components to domain analysis. *Int. Journal of Knowledge Engineering and Software Engineering*, 1992.

[167] M. Odersky. *A New Approach to Formal Language Definition and its Application to Oberon*. PhD thesis, ETH Zürich, 1989.

[168] M. Odersky. Programming with variable functions. In *International Conference on Functional Programming*, Baltimore, 1998. ACM.

[169] R. O'Hara and D. Gomberg. *Modern Programming Using REXXX*. Number ISBN 0-13-597329-5. Prentice Hall, 1988.

[170] OMB. Model-driven architecture home page. http://www.omg.org/omg/index.htm.

[171] J. Ousterhout. *Tcl and the Tk Toolkit*. Number ISBN 0-201-63337-X. Addison-Wesley, 1994.

[172] J. K. Ousterhout. Tcl: An embeddable command language. In *Winter USENIX Conference Proceedings*, 1990.

[173] J. K. Ousterhout. Scripting: Higher level programming for the 21st century. *IEEE Computer*, 31(3):23–30, March 1998.

[174] D. Parigot. Attribute Grammars Home Page, URL: `http://www-sop.inria.fr/oasis/Didier.Parigot/www/fnc2/attr`

[175] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Comm. ACM*, 12(2), 1972.

[176] D. L. Parnas. On the design and development of program families. *IEEE Transactions on Software Engineering*, pages 1–9, March 1976.

[177] D. Pavlovic and R. Smith. Composition and refinement of behavioral specifications. In *Proceedings of 16th Automated Software Engineering Conference*, pages 157–165. IEEE press, November 2001.

[178] D. Pavlovic and R. Smith. Guarded transitions in evolving specifications. In *Proceedings of AMAST'02*, 2002.

[179] R. Pawson. *Expressive Systems, a manifesto for radical business software*, chapter An expressive system to improve risk management in options trading, pages 36–43. CSC Research Services, 2000.

[180] P. Pfahler and U. Kastens. Language design and implementation by selection. In Kamin (116). Published as University of Illinois at Urbana Champaign Computer Science Report URL: `www-sal.cs.uiuc.edu/~kamin/dsl`.

[181] A. Pierantonio. Making statics dynamic: Towards an axiomatization for dynamic adt's. In G. Hommel, editor, *Proc. Int. Workshop on Communication Based Systems*, pages 19–34. Kluwer Accademic Publisher, 1995.

[182] G. D. Plotkin. A structural approach to operational semantics. Lecture Notes DAIMI FN-19, Department of Computer Science, University of Aarhus, 1981.

[183] A. Poetzsch-Heffter. *Formale Spezifikation der kontextabhängigen Syntax von Programmiersprachen*. PhD thesis, Technische Uni. München, 1991. in german.

[184] A. Poetzsch-Heffter. Programming Language Specification and Prototyping using the MAX System. In M. Bruynooghe and J. Penjam, editors, *Programming Language Implementation and Logic Programming*, volume 714 of *Lecture Notes in Computer Science*, pages 137 – 150. Springer–Verlag, 1993.

[185] A. Poetzsch-Heffter. Developing Efficient Interpreters Based on Formal Language Specifications. In P. Fritzson, editor, *Compiler Construction*, volume 786 of *Lecture Notes in Computer Science*, pages 233 – 247. Springer–Verlag, 1994.

[186] A. Poetzsch-Heffter. Prototyping realistic programming languages based on formal specifications. *Acta Informatica*, 34:737–772, 1997. 1997.

[187] M. Posypkin. Personal communications. email, January 2003.

[188] S. P. Reiss. PECAN: Program development systems that support multiple views. *IEEE Transactions on Software Engineering*, SE-11(3):276 – 285, March 1985.

[189] T. Reps and T. Teitelbaum. *The Synthesizer Generator: A System for Constructing Language-Based Editors*. Texts and Monographs in Computer Science. Springer Verlag, New York, 1989.

[190] J. C. Reynolds. Reasoning about arrays. *Communications of the ACM*, 22:290–299, 1979.

[191] D. A. Schmidt. On the need for a popular formal semantics. *ACM SIGPLAN Notices*, 32(1):115–116, 1997.

[192] D. S. Scott and C. Strachey. Toward a mathematical semantics for computer languages. *Computers and Automata*, (21):14 – 46, 1971. Microwave Research Institute Symposia.

[193] M. G. Sèmi. Gènèration de spècifications centaur á partir de spècifications montages. Master's thesis, Universitè de Nice–Sophia Antipolis, June 1997. in French.

[194] N. Shankar. Symbolic Analysis of Transition Systems. In Gurevich et al. (86).

[195] E. Sheedy and S. McCracken, editors. *Derivatives, the risks that remain*. Macquarie Series in Applied Finance. Allen & Unwin, 1997.

[196] S. Siewert. A common core language design for layered language extension. Master's thesis, Univ. of Colorado, 1993. http://www-sgc.colorado.edu/people/siewerts/msthesis/thesisw6.htm.

[197] C. Simonyi. The death of computer languages, the birth of intentional programming. Technical Report MSR-TR-95-52, Microsoft Research, 1995.

[198] C. Simonyi. The future is intentional. *IEEE Computer*, pages 56–57, May 1999.

[199] D. Spinellis. Reliable software implementation using domain-specific languages. In G.I. Schuëller and P. Kafka, editors, *Proc. ESREL'99 – 10th European Conf. on Safety and Reliability*, pages 627 – 631, September 1999.

[200] D. Spinellis. Notable design patterns for domain specific languages. *Journal of Systems and Software*, 56(1):91–99, February 2001.

[201] D. Spinellis and V. Guruprasad. Lightweight languages as software engineering tools. In *USENIX Conference on Domain-Specific Languages* (1), pages 67–76.

[202] T. Standish. Extensibility in programming languages design. *SIGPLAN Notices*, July 1975.

[203] R. Stärk. Abstract state machines for java. Lecture Notes for Computer Science Students, Theoretische Informatik 37-402, Departement Informatik, ETH Zürich, 1999. available at http://www.inf.ethz.ch/∼staerk/teaching.html.

[204] R. Stärk, J. Schmid, and E. Börger. *Java and the Java Virtual Machine - Definition, Verification, Validation*. Springer Verlag, 2001.

[205] L. Starr. *Executable UML: A Case Study*. Model Integration, LLC, February 2001.

[206] J. Szyperski, C.and Gough. The role of programming languages in the life-cycle of safe systems. In *STQ'95, 2nd Int. Conf. on Safety Through Quality, Kennedy Space Center, Cape Canaveral, Florida, USA*, October 1995.

[207] A. Tarski. Der wahrheitsbegriff in den formalisierten sprachen. *Studia Philosophica*, (1):261–405, 1936. English translation in A. Tarski. Logic, Semantics, Methamathematics. Oxford University Press.

[208] J. Teich, P. W. Kutter, and R. Weper. Description and Simulation of Microprocessor Instruction Sets Using ASMs. In Gurevich et al. (86), pages 266–286.

[209] S. Thibault. *Domain-Specific Languages: Conception, Implementation and Application*. PhD thesis, l'Université Rennes 1, Institut de Formation Supérieure en Informatique et Communication, October 1998.

[210] S. Thibault and C. Consel. A framework of application generator design. In M. Harandi, editor, *Proceedings of the ACM SIGSOFT Symposium on Software Reliability (SSR '97)*, volume 22 of *Software Engineering Notes*, pages 131 – 135, Boston, USA, May 1997.

[211] A. M. Turing. On computable numbers, with an application to the Entscheidungs problem. *Proc. London Math. Soc.*, 2(42):230–265, 1936. (Corrections on volume 2(43):544–546).

[212] J. Uhl. Spezifikation von programmiersprachen und uebersetzern. Berichte 161, Gesellschaft fuer Mathematik und Datenverarbeitung, 1986. in German.

[213] M. G. J. van den Brand, J. Heering, P. Klint, and P. A. Olivier. Compiling language definitions: The asf+sdf compiler. *ACM Transactions on Programming Languages and Systems*, 24(4):334–368, 2002.

[214] M. G. J. van den Brand, A. van Deursen, P. Klint, S. Klusener, and E. A. van der Meulen. Industrial applications of ASF+SDF. In *Proc. AMAST'96, 5th International Conference on Algebraic Methodology and Software Technology*, Munich, Germany, July 1996. Springer-Verlag. Lecture Notes in Computer Science 1101.

[215] A. van Deursen. Using a domain-specific language for financial engineering. *ERCIM News*, (38), July 1999.

[216] A. van Deursen and P. Klint. Little languages: Little maintenance? *Journal of Software Maintenance*, 10:75 – 92, 1998.

[217] A. van Deursen, P. Klint, and J. Visser. Domain–specific languages – an annotated bibliography. *ACM SIGPLAN Notices*, 35(6), June 2000.

[218] T. van Rijn. Financial product solution. Cap Gemini Ernst & Young, internal documentation.

[219] G. van Rossum and J. de Boer. Interactively testing remote servers using the Python programming language. *CWI Quarterly, Amsterdam*, 4(4):283–303, 1991.

[220] J. Visser. Evolving algebras. Master's thesis, Delft University of Technology, 1996.

[221] W. M. Waite and G. Goos. *Compiler Construction*. Springer Verlag, 1984.

[222] L. Wall, T. Christiansen, and R. Schwartz. *Programming Perl*. Number ISBN 1-56592-149-6. O'Reilly and Associates, second edition, 1996.

[223] L. Wall and R. L. Schwartz. *Programming Perl*. O'Reilly & Associates, Inc., 1991.

[224] C. Wallace. The Semantics of the C++ Programming Language. In E. Börger, editor, *Specification and Validation Methods*, pages 131 – 164. Oxford University Press, 1994.

[225] C. Wallace. The Semantics of the Java Programming Language: Preliminary Version. Technical Report CSE-TR-355-97, University of Michigan EECS Department Technical Report, 1997.

[226] R. Weicker. Dhrystone: A synthetic systems programming benchmark. *Comm. of the ACM*, 27(10):1013–1030, October 1984.

[227] D. M. Weiss and C. T. R. Lai. *Software Product Line Engineering: A Family-Based Software Development Process*. Addison Wesley, Reading, MA, 1999.

[228] R. L. Wexelblat. Maxims for malfeasant designers, or how to design languages to make programming as difficult as possible. In *Proc. of the 2nd Int. Conf. on Software Engineering*, pages 331–336. IEEE Computer Society Press, 1976.

[229] I. Wilkie, A. King, M. Clarke, C. Weaver, and C. Raistrick. Uml action specification language (asl), reference guide. Kennedy Carter Limited, KC/CTN/06, www.kc.com, February 2001.

[230] N. Wirth. On the design of programming languages. In J.L. Rosenfeld, editor, *Information Processing 74, Proc. of IFIP Congress 74*, pages 386–393. North-Holland Publishing Company, 1074.

[231] N. Wirth. The programming language PASCAL. *Acta Informatica*, 1(1):35 – 63, 1971.