Executable Language Definitions

— Case Studies and Origin Tracking Techniques —

ILLC Dissertation Series 1994-5

institute *for* logic, language *and* computation

For further information about ILLC-publications, please contact

Institute for Logic, Language and Computation
Universiteit van Amsterdam
Plantage Muidergracht 24
1018 TV Amsterdam
phone: +31-20-5256090
fax: +31-20-5255101
e-mail: illc@fwi.uva.nl

Executable Language Definitions

— Case Studies and Origin Tracking Techniques —

Academisch Proefschrift

ter verkrijging van de graad van doctor
aan de Universiteit van Amsterdam,
op gezag van de Rector Magnificus
Prof.dr P.W.M. de Meijer
ten overstaan van
een door het college van dekanen ingestelde commissie
in het openbaar te verdedigen
in de Aula der Universiteit
(Oude Lutherse Kerk, ingang Singel 411, hoek Spui)
op donderdag 29 september 1994 te 13:30 uur

door

Arie van Deursen

geboren te 's Gravenhage.

Promotor: Prof. dr. P. Klint
Faculteit: Wiskunde en Informatica

This thesis has been made possible by the financial support of:

- NWO, the Netherlands Organization for Scientific Research, project Incremental Program Generators;
- The European Communities, ESPRIT project 2177, Generation of Interactive Programming Environments GIPE II;
- ORFIS International B.V., Organization for Financial Information Systems, Huis ter Heide, The Netherlands.

CIP-GEGEVENS KONINKLIJKE BIBLIOTHEEK, DEN HAAG

Deursen, Arie van

Executable language definitions : case studies and origin
tracking techniques / Arie van Deursen. -[S.1. : s.n.].
- (ILLC dissertation series ; 1994-5)
Proefschrift Universiteit van Amsterdam. - Met lit. opg.
ISBN: 90-74795-09-9
Trefw.: programmeertalen.

CWI, P.O. Box 94079, 1090 GB Amsterdam, arie@cwi.nl

Printed and bound by CopyPrint 2000, Enschede

Contents

Preface

After many days of writing, I am very happy to have arrived at the section of my thesis in which I get an opportunity to thank every one who in one way or another has contributed to this thesis.

My sincere thanks go to my promotor Paul Klint: For his relaxed and happy attitude, for listening to all my nonsense-proposals, for giving me four years of complete freedom at work, for allowing me to drop in whenever I wanted, for showing me the right balance between theory and practice, and not in the least for being prepared to read and correct the manuscript even during his holidays.

It is an equally great pleasure to thank Jan Heering. No one can read a paper like he can, returning it fully decorated with highly relevant remarks. Missing dots, wrong indices, inconsistencies in examples, obsolete citations, vagueness, theoretical subtleties, verbosity: he does not leave anything unreported. I hope that he will keep on reading all of my future papers.

With T.B. Dinesh I could discuss any subject, be it in computer science or not. His favorite question ("Why Arie, why?") has helped me every time he asked it.

I would like to thank Emma van der Meulen for good times and encouragement. Without her work on PRSs, I could not have written Chapter 7.

It is a privilege to thank the co-authors of the various chapters: Bert Arnold (Chapter 3); Peter Mosses (5); Frank Tip and Paul Klint (6); and T.B. Dinesh (8).

I am honoured that prof. dr. K.R. Apt, prof. dr. J.A. Bergstra, prof. dr. J. van Katwijk, prof. dr. ir. R. Maes, and prof. dr. P.D. Mosses, were willing to be a member of the reading committee.

Various people supported me greatly by acting as a "local reading committee", and providing me with numerous useful comments: Dinesh, Emma van der Meulen, Femke van Raamsdonk, Frank Tip, Susan Üsküdarlı, Eelco Visser, and Machteld Vonk are all gratefully acknowledged.

Working in the ASF+SDF/GIPE group at CWI and UvA has been most enjoyable, and I thank all members: Huub Bakker, Mark van den Brand, Dinesh, Casper Dik, Job Ganzevoort, Jan Heering, Paul Hendriks, Jasper Kamperman, Paul Klint, Wilco Koorn, Emma van der Meulen, Leon Moonen, Jan Rekers, Martijn Res, Frank Tip, Susan Üsküdarlı, Eelco Visser, Paul Vriend, and Pum Walters.

It is a privilege to work at an institute like CWI. I would like to thank the head of the department of software technology, Jaco de Bakker, as well as Mieke Bruné, for being there whenever I needed some form of help. Colleagues like Doeko Bosscher, Wan Fokkink, Steven Klusener, and Henri Korver made working at CWI great fun.

My family and friends had to listen to my PhD thesis progress reports, and had to suffer my "growing neglect". I thank them for remaining friendly, and for forcing me to chat and have a drink with them. I particularly would like to thank my parents, for their confidence in the good ending of my PhD efforts.

Finally, I am indebted to Machteld Vonk. She contributed in many ways, of which proof-reading every sentence in every chapter and happily listening to important discoveries in the areas of term rewriting, origin tracking, or action-semantic tool-support are just two of the least important ones.

Arie van Deursen
Amsterdam, 19 August 1994

The Origins of Chapters

Several chapters are revised versions of earlier publications:

- **Chapter 2: Introducing ASF+SDF**

  Based on "Specification and Generation of a λ-calculus Environment" [Deu92], appeared in the CSN'92 (Computing Science in the Netherlands) conference proceedings and as CWI technical report CS-R9233.

- **Chapter 3: A Language for Defining Financial Products**

  Based on joint work with B.R.T. Arnold from CAP Volmac, entitled "Algebraic Specification of a Language Defining Interest Rate Products" [AD92].

  CAP Volmac and Bank MeesPierson are acknowledged for their permission to publish a summary of the confidential report of their former subsidiary Orfis International.

- **Chapter 4: The Static Semantics of Pascal**

  Based on "An Algebraic Specification for the Static Semantics of Pascal" [Deu91], appeared in the CSN'91 conference proceedings and as CWI technical report CS-R9129.

- **Chapter 5: Tools for Action Semantics**

  Joint work with Peter D. Mosses from Aarhus University, Denmark.

- **Chapter 6: Principles of Origin Tracking**

  Based on joint work with P. Klint and F. Tip, appeared as "Origin Tracking" [DKT93] in the *Journal of Symbolic Computation* and as CWI technical report CS-R9230.

  Academic Press is acknowledged for their permission to reprint parts of this paper.

- **Chapter 7: Origin Tracking in Primitive Recursive Schemes**

  An earlier version appeared under the same title in the CSN'93 conference proceedings and as CWI technical report CS-R9401 [Deu94].

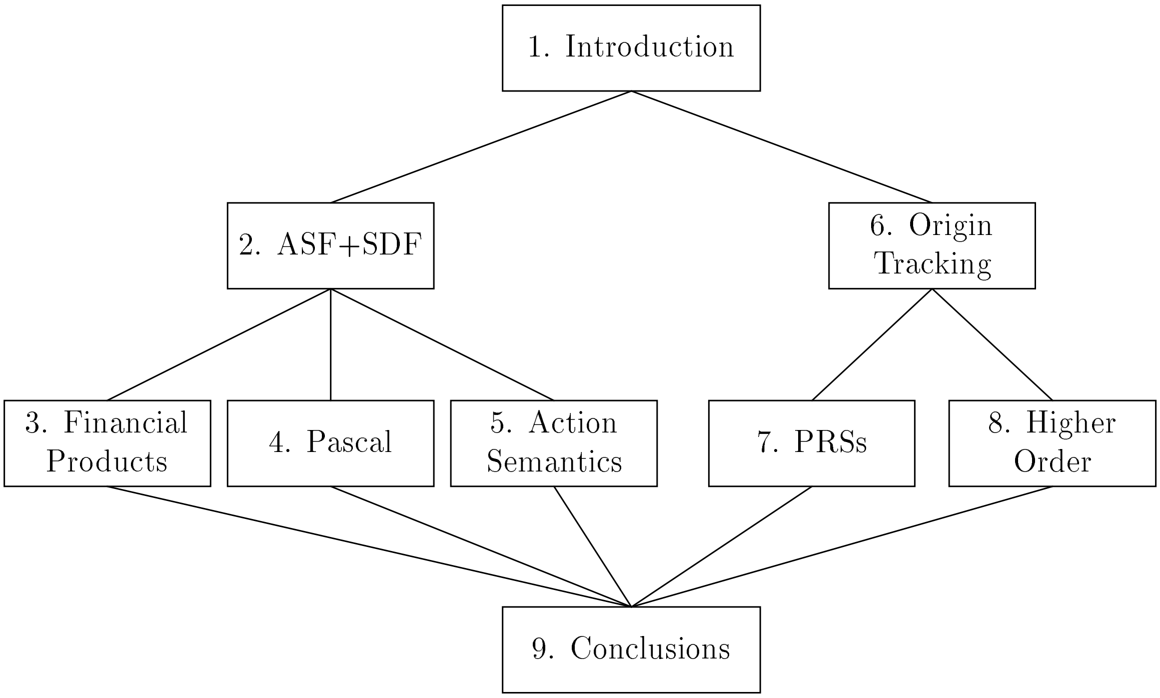
- **Chapter 8: Origin Tracking for Higher-Order Rewriting Systems**

  Joint work with T. B. Dinesh. Appeared under the same title in "Higher-Order Algebra, Logic, and Term Rewriting (HOA'93)", LNCS 816, 1994 [DD94] and as CWI technical report CS-R9425.

  Springer-Verlag is acknowledged for their permission to reprint this paper.

Chapter Dependencies

The following diagram indicates the main dependencies between the individual chapters:



Chapter 1

Introduction

The aims of this thesis are to gain experience and to localize problems with the use of algebraic specifications for writing language definitions, and the use of term rewriting for executing them. Three case studies, performed in the context of the ASF+SDF Meta-environment, are presented. For one of the problems encountered, a solution called *origin tracking* is discussed in full detail.

1.1 Language Definitions

Computer systems can be viewed as machines capable of interpreting languages [BJ82]. This is immediately clear for language-oriented programs like a C compiler or a BASIC interpreter; for calculators evaluating arithmetic expressions, databases answering queries, interactive programs expecting input in a certain format, and even for hi-fi or video equipment accepting only a limited set of all possible button click sequences; for all these systems the statement is equally true.

As languages play such a central role, it is important to gain as much understanding as possible of languages as they are used in computing. Not only will this assist us when explaining or suggesting improvements for unfriendly systems based on unnecessarily restrictive input languages. More importantly, it will help us to do a good job when designing new (programming) languages.[1]

In order to increase our insight into languages, a great deal of computer science research has been devoted to the analysis and description of programming languages. An important device for the analysis of a language is a *formal language definition*, i.e., a mathematical model of the language. Writing such a formal definition is a time consuming task, but results in clearness and unambiguity for those language features formalized, which can be particularly important if the language is still in its design phase. Once the language design and formalization are finished, the definition can play a role as a basis for language implementations.

[1] As formulated during a recent workshop [Kam94], this should prevent us from proposing "poorly designed and documented, frustratingly limited, unfriendly, and inefficient languages and language processors". Failing to do so will result in "the dissipation of thousands of person-hours of work, using ungainly languages to produce unsafe and inefficient applications."

As in the study of natural languages, programming languages are described by specifying their *syntax* and *semantics*. The syntax, formalized using a *context-free grammar*, determines what kind of sentences are allowed, and the semantics assigns a meaning to these sentences. Context-free grammars are relatively well-understood, but it is far less clear how the semantics of a complex language can be conveniently described. Several approaches to (programming) language semantics exist, which can be roughly divided into three groups:

- Structural operational semantics [Plo81] and natural semantics [Kah87] take an *interpretive* point of view. The meaning of a construct is specified by the computation it induces when executed on a (mathematical) machine [NN92].
- Axiomatic [Apt81] and algebraic semantics [GTWW77] are more of a *property-oriented* nature. In an axiomatic semantics it is defined how certain properties of programs can be proven. In an algebraic semantics equivalences between operators in the language are formalized.
- In the *translational* point of view, finally, a program is mapped to an expression over a domain that is so well-understood that it can be regarded as the program's meaning. The best known approach is denotational semantics, which maps programs to higher-order functions expressed in the λ-calculus [Sto77, Bak80, Sch86]. Other translational approaches are *action semantics* mapping programs to "Action Notation" [Mos92], or the PIM formal system which maps programs to an algebra of so-called PIM-terms [Fie92].

Sometimes, one encounters mixtures of the above approaches. For instance, one can define a translation using structural operational semantics, or define an interpreter using algebraic techniques.

These approaches to semantics are not mere theory, but have been used to define several existing and complicated languages. Some of the more recent experiments are an operational semantics of Standard ML [MTH90], a denotational semantics of Scheme [RC86], or an action semantics of Pascal [MW93]. However, we are not yet at the stage where giving a formal description of a realistic programming language is a matter of simple routine. A draft version of the denotational semantics (using the VDM formalism [BJ82]) for Modula-2, for instance, was a 600-page document, and three years after the publication of the operational semantics of ML a report entitled "Mistakes and ambiguities in the definition of Standard ML" [Kah93] appeared.

1.2 Language-specific Tools

A formal language definition may be useful for understanding or describing a language; it is equally important that the language is actually *used*. For many languages, this is only possible if language-specific *tools* are available. For others, using the language without tools is possible but undesirable, as a tool could easily help to spot, e.g., syntactic errors.

Perhaps the best-known example of a tool is a *compiler*, mapping one language to code that is executable by the machine architecture used. Another example is a *debugger*, which helps to inspect values of variables or to set breakpoints in a program in order to find bugs. A *parser* checks whether a piece of text represents a syntactically correct program. It is used in a *syntax-directed editor* which provides language-specific help while an end-user is entering his or her program. Other tools that we will encounter in this thesis include a *type checker* performing a context-sensitive analysis of a program, an *evaluator* or *interpreter* used to run programs, a program *animator* used to visualize the execution of a program, or a *pretty printer* (also called an *unparser*) which takes a program in an abstract representation, and displays it in a human-readable format.

An integrated collection of tools will be referred to as language-specific (programming) *environment*. Again, the early availability of a (prototype) environment will help the language designer to experiment with and hence improve the language designed.

1.3 Automatic Tool Generation

As the construction of a formal language definition and the early availability of prototype tools are both beneficial to the language design process, it is natural to try to integrate them into one activity. If a language definition is sufficiently detailed and in a computer-readable format, it can be used to *generate* tools for the language automatically. This has two advantages: the tools (or at least prototype versions) come for free, and the language designer is encouraged to formalize his or her language.

Several successful tool generators are based on syntax-directed translations in the form of *attribute grammars*. One of the best-known systems is the Synthesizer Generator [RT89a]. It generates an incremental, language-specific editor from a definition of the context-free syntax and the context-sensitive requirements (the *static semantics*).

Besides generators based on syntax and attribute grammars, several systems exist that take a semantic description as their point of departure. To mention just a few of them,[2] the PSG system uses a denotational semantics to generate a language's interpreter [BS86]. The CENTAUR system can execute a description written in TYPOL which is used to express the natural semantics of a language [BCD+89]. Finally, the ACTRESS compiler generator takes an action-semantic description of a language and produces a compiler translating to C [BMW92].

Whether the tools obtained are interpreters, parsers, type checkers, or debuggers depends on the properties formalized and the generators used. Several tool generators, such as the PSG or the CENTAUR system provide means to specify a series of language characteristics, ranging from the language's syntax to pretty printing, semantics, transformations, or context-sensitive constraints. Moreover, they may be capable of generating both a debugger, compiler, and an evaluator from a single definition of the language's semantics. Such systems can be called *programming environment generators*.

[2] See Section 2.5.3 for a more elaborate list of such generators.

1.4 Algebraic Specifications of Languages

In this thesis, we will use *algebraic specifications* as our device to formalize languages [GTW78, Wir90, MT92]. Moreover, we will use these algebraic language definitions to obtain tools, the backbone of which will consist of *parsers* and *term rewriting systems* (see below). There are at least three good reasons to use algebraic specifications for the definition of languages:

- A first one is *uniformity*: algebraic specifications can be used to define various language properties, including context-sensitive analysis, program transformations, and compilations [BHK89]. Moreover, they enable a specifier to choose between an operational (interpretive) or denotational (translational) style of giving semantics to languages (see, e.g., [BWP87, BHK89, Mos92]).
- A second motivation for using algebraic specifications is *simplicity*. There are only two key concepts in algebraic specifications: *terms* and *equations*. The terms indicate what kind of entities we are defining, and the equations indicate the relations between these entities, in this case equality relations. Reasoning about equalities is relatively easy, as formalized by the rules of *equational logic* [MG85].
- A third advantage of algebraic specifications is their potential *executability*. Quite often, sets of equations can be interpreted as *term rewriting systems* [Klo92]. Such term rewriting systems can *reduce* complex terms to result terms. Term rewriting is a way to execute algebraic language definitions, and thus a way to obtain tools for the language specified.

In this thesis we will take a closer look at algebraic specifications of languages. In particular, we will aim at identifying problematic issues when using algebraic specifications for language definition purposes. One of the questions that will be addressed is whether term rewriting is a sufficiently sophisticated technique to base tool generation on.

1.5 ASF+SDF

In this thesis, or at least the first part of it, we will be particularly concerned with the ASF+SDF[3] formalism, an algebraic specification formalism designed for writing language definitions [BHK89, HHKR89]. It comes with the ASF+SDF Meta-environment,[4] an environment that can be used to generate language-specific environments from language definitions written in the ASF+SDF formalism [Kli93c].

[3]The name ASF+SDF tells something about the history of the formalism; it is the result of the integration of the Algebraic Specification Formalism ASF with the Syntax Definition Formalism SDF.

[4]The ASF+SDF Meta-environment is implemented in, and distributed as part of, the "language design and implementation workbench" CENTAUR, see Section 2.5.3.

Some of the main ingredients of ASF+SDF are inspired by the success of several existing languages and systems. The ASF+SDF formalism, e.g., is a conventional first-order algebraic specification formalism, inspired by languages such as Clear [BG77] or ACT-ONE [EM85]. The idea to generate syntax-directed editors comes from systems like the Synthesizer Generator [RT89a] or the PSG system [BS86]. Executing algebraic specifications using (conditional) term rewriting, finally, has been part of the OBJ systems for several years [GWM+92]. From a different perspective, however, the ASF+SDF formalism and system are distinctive in several ways:

- First of all, ASF+SDF is a formalism providing maximal syntactic freedom.[5] The notation for terms is defined using *arbitrary* context-free grammars, not limited by requirements enforced by tool generators restricting the grammar to LR, LALR, or LL formats.

- Secondly, the ASF+SDF Meta-environment is a tool generator in which one formalism suffices to describe all language properties, and hence to obtain all possible tools. Thus, it is not necessary to learn different formalisms; The ASF+SDF formalism can be used to describe abstract syntax, concrete syntax, lexical syntax, pretty printing, type checking, operational semantics, and so on.

- Finally, the ASF+SDF Meta-environment is a highly *incremental* system. As the language definition is entered, tools are derived immediately. If the definition changes, the tools are adapted rather than regenerated from scratch.

  This incremental situation supports interactive language development: A language designer defines a new language construct, watches its behavior in the generated tools, decides to make a small modification, which causes the system to update the generated tools, which subsequently makes the designer decided that another change is needed, and so on.

In this thesis, we will specify some complex languages using ASF+SDF. Our aim is to localize problems, and, if possible, to present general solutions to these problems.

1.6 Subjects of this Thesis

Most readers will agree that it is a very good idea to combine formal language definitions with automatic tool generation. However, they will also be worried about the complexity of writing a formal language definition for realistic languages, as illustrated by the problems with the formalizations of Standard ML and Modula-2. Apparently, this route is not without problems.

This thesis aims at finding out what these problems are, and if possible to eliminate them by offering a solution. The framework in which this will be done is that of algebraic specifications. The method that will be used is carrying out a number of case studies, using the ASF+SDF formalism and Meta-environment.

[5] Again, see Section 2.5.3 for a comparison with the mixfix notation of OBJ.

Part I presents the details of the ASF+SDF approach as well as three case studies. Part II takes one specific problem apart, and discusses a general solution, referred to as *origin tracking*, in full detail.

1.6.1 Part I: Specification Case Studies

The three case studies concern (i) a commercially used application language, (ii) the programming language Pascal, and (iii) a complicated specification language, the "Meta Notation" as used in Action Semantics.

In order to assess the applicability in a commercial environment, we were fortunate to be involved in a project at the Dutch Bank MeesPierson, in cooperation with software house CAP Volmac. The bank offers its clients a wide variety of products based on interest rates. Each time a new product is introduced, the software running at the bank has to be updated — a tedious job which has to be done manually. The idea of this project was to *describe* these various products using some *language*, and to *generate* the software automatically. This language was defined, analyzed, and reviewed using the ASF+SDF formalism, commissioned by CAP Volmac. The outcome of this project is discussed in our first case study, Chapter 3.

Our second case study concerns a larger language: Pascal. Chapter 4 presents an algebraic specification of the static semantics of full ISO Pascal [ISO83]. The problems involved in deriving a useful type checker (yielding informative error messages) from this specification will receive special attention.

The third and most complicated case study focuses on the use of ASF+SDF as a platform to implement tools support for other specification languages. The tools aimed at in this case study should support the development of Action Semantic descriptions [Mos92]. The language defined is the so-called "Meta Notation", the formalism to write action-semantic descriptions. The results achieved and the problems encountered when developing these tools for action semantics are described in Chapter 5.

The point of departure in all case studies is ASF+SDF. To that end, ASF+SDF is introduced, using a simple example, in Chapter 2. The results presented will be, whenever possible, generalized to arbitrary language definition formalisms and tool generators.

1.6.2 Part II: Origin Tracking

One of the case studies of Part I, namely the specification of the static semantics of Pascal in Chapter 4, revealed a deficiency in the existing technology to generate tools from algebraic specifications. In particular, the following problems showed up.

Although it was easy to execute a specification of the static semantics for some language, resulting in a type checker that indicates that a program is wrong, what is wrong, and perhaps even why it is wrong, it was not so straightforward to identify exactly *where* in the source program the error was made. For instance, a message like "identifier i not declared" could easily show up, but how could the programmer know which of the many occurrences of i in his (block-structured) program was undeclared?

Likewise, it was straightforward to execute an operational semantics thus obtaining an interpreter for the specified language. It was more difficult, however, to derive a tool that gives extra information which could help in understanding how the program was running, e.g., an *animator* showing the flow of control by highlighting the statements executed at each moment.

Finally, it was easy to execute a translation specification yielding a compiler, but it was harder to obtain a compiler with source-level debugging facilities, i.e., a debugger that runs the target program but explains it to the user in terms of the source program.

It seemed that all these problems could be solved if the tool generator had more understanding how the computations were actually performed. In particular, a closer relationship between the outcome of a computation and the initial data was needed. In Part II we will present a technique establishing this. This new technique has to satisfy the following criteria:

- It should help to understand or explain computations described by equations.
- It should at least allow the generation of error handlers, animators, and source-level debuggers from algebraic specifications of, respectively, static semantics, dynamic semantics, and translators.
- It should not require any changes to or restrictions on the formal language specifications, nor should it require that specifiers pollute their specifications with directives specific for this technique.

Part I

Specification Case Studies

Chapter 2

Introducing ASF+SDF

The main features of the algebraic specification formalism ASF+SDF are introduced, covering context-free and lexical syntax, signatures, associative lists, and conditional equations. Moreover, the supporting environment and tool generator, the so-called ASF+SDF Meta-environment, is discussed. The formalism is illustrated using a small example, a specification of the λ-calculus with valid substitutions, α, β, and η-conversion, and left-most reductions. The tools generated automatically from this specification, which together constitute a small λ-calculus environment, are presented.

2.1 Introduction

Some familiarity with ASF+SDF will help the reader to understand the case studies in the chapters still to come. Rather than just listing the features of ASF+SDF, we illustrate its use by discussing a small example. Our example is the λ-calculus, which is (i) concise enough to be presented completely, (ii) sufficiently well-known to be understood easily, (iii) nevertheless non-trivial, and (iv) useful in practice (indeed, the generated environment has been used to teach the λ-calculus).

The idea to use a specification of the λ-calculus came to mind while reading Gordon's book *Programming Language Theory and Implementation* [Gor88]. Gordon devotes one chapter to an informal description of the λ-calculus. He needs a second chapter to cover an implementation (in Lisp) of a λ-calculus environment, consisting of tools to experiment with conversions, left-most reductions, let-constructs, and so on. In our chapter, we will reduce these two activities to specification only; the required tools are generated automatically. Combining these tools yields an integrated λ-calculus environment consisting of syntax-directed editors for λ-expressions and let-constructs, extended with a graphical user-interface to initiate α, β, and η-conversions, let-expansions, and left-most reductions.

This chapter is organized as follows. After briefly mentioning some important concepts concerning algebraic specifications and the ASF+SDF formalism, we will present all eleven ASF+SDF modules of the λ-calculus specification. Next, we will discuss the generated λ-calculus environment. Finally, we will illustrate how the ASF+SDF Meta-environment has helped during the development of the specification.

The single motivation for this chapter is to illustrate the use of ASF+SDF. The reader is referred to [BHK89, HHKR89, Kli93a, Kli93c] for all details of ASF+SDF, to

[Bar84] for a treatise on the λ-calculus, and to [ACCL90, Har93] for more specifications of the λ-calculus.

2.2 Algebraic Specifications in ASF+SDF

In addition to discussing various important concepts related to algebraic specifications, such as constructors, sufficient completeness, and conditional equations, the main features of ASF+SDF are illustrated by examples.

2.2.1 Algebras, Terms, and Equalities

The ASF+SDF formalism is an *algebraic specification formalism*. Algebraic specifications are advocated because of their universality, simplicity, and abstractness [GTW78, Wir90, MT92]. Essential in algebraic specifications is that they are used to define *algebras*. An algebra consists of a set of *values* and a number of *functions* operating on these values. The relationships between the values and the operations can be expressed as *equalities*. A well-known example of an algebra is the ring of integers with values $\ldots, -2, -1, 0, 1, 2, \ldots$, and operations such as $+$ for addition or $*$ for multiplication. Equalities that we expect to hold are, for instance, $3 = 3 * 1$ or $3 + 0 = 4 + -1$.

An algebraic specification aims at characterizing a particular algebra.[1] To that end, algebraic specifications identify a collection of *terms*, as well as a series of *equations* over these terms. Terms are built from *function symbols*, also called *constants* if they have no arguments.

An example of a simple algebraic specification is ASF+SDF module Booleans (Module 2.1). The first five lines after the keyword **context-free syntax** introduce two constants *true* and *false*, and three function symbols, $\_ \wedge \_$, $\_ \vee \_$, and $\neg\_$. This set of five symbols is called the *signature* of the specification. From it an infinite number of terms can be built: *true*, *false*, *true* $\vee$ *false*, *true* $\wedge$ *false*, $\neg$ (*true* $\wedge$ *false*), ...

The equations listed after the **equations** keyword in the specification can be used to deduce equalities over these terms. For instance, the equations given are sufficient to deduce $\mathit{true} \wedge (\neg\, \mathit{false}) \overset{[6]}{=} \mathit{true} \wedge \mathit{true} \overset{[3]}{=} \mathit{true}$. In fact, the six equations suffice to deduce that any term over the Boolean signature is either equal to *true* or equal to *false*. For this reason *true* and *false* are called the *constructors* of the Booleans, as opposed to $\_ \wedge \_$, $\_ \vee \_$, and $\neg\_$, which are *defined functions*. A specifier can indicate which symbols are intended as constructors by attributing them with the keyword **constructor**.

A more operational point of view may provide some further intuition. The equations can be regarded as asserting, in addition to the symmetrical equality relation, a one-way *rewrite* possibility. When oriented from left to right, the equations of Booleans (Module 2.1) have a "simplifying" effect, i.e., they can be used to rewrite complex terms to simpler ones. For instance, in the previous paragraph the term *true* $\wedge$ ($\neg$ *false*) was reduced to just *true* using equations [6] and [3] as *rewrite rules*.

[1] More precisely, an isomorphism class of algebras.

module Booleans
imports Layout$^{(2.3)}$
exports
sorts BOOL
context-free syntax
true → BOOL {**constructor**}
false → BOOL {**constructor**}
BOOL "∧" BOOL → BOOL {**left**}
BOOL "∨" BOOL → BOOL {**left**}
"¬" BOOL → BOOL
"(" BOOL ")" → BOOL {**bracket**}
priorities
"∨" < "∧" < "¬"

variables
$P \rightarrow$ BOOL

equations

[1] $\mathit{true} \vee P = \mathit{true}$ [3] $\mathit{true} \wedge P = P$ [5] $\neg\, \mathit{true} = \mathit{false}$
[2] $\mathit{false} \vee P = P$ [4] $\mathit{false} \wedge P = \mathit{false}$ [6] $\neg\, \mathit{false} = \mathit{true}$

Module 2.1: Booleans. The Boolean data type in the ASF+SDF formalism.

The result of repeatedly applying rewrite rules can be a term to which no rule can be applied anymore. Such a term is said to be in *normal form*. We typically expect our normal forms to consist of constructors only. In other words, the defined functions should be eliminated by the rewrite rules, which has the effect of "executing" the defined functions (computing their values). As an example, equations [1] and [2] can be used to compute result values for ∨ functions; equations [3] and [4] for ∧, and [5] and [6] to eliminate ¬. A specification where all terms can be reduced to terms built from constructors only is said to be *sufficiently complete* [GH78].

Recall that we started by proclaiming that we want to define *algebras*. One algebraic specification can characterize various algebras, the so-called *models* of the specification. A model algebra is obtained by assigning values to terms over the signature, and functions to the function symbols defined in the signature. The valid equalities between the values are those that can be deduced from the equations in the specification using the rules formalized in *equational logic*. If we want to restrict ourselves to just one model the most natural candidate is the one based on rewriting, with the constructor terms representing the values of the algebra, and the defined functions the operations. The model-theoretic issues related to the corresponding (more general) *initial model* are discussed in full detail in [MG85]. Notice that, although constructors and defined functions are important, we neither insist on the use of **constructor** declarations nor on sufficient completeness.

2.2.2 Signatures and Grammars

In our specifications we will encounter several *sorts* of values, such as BOOL, NAT, LIST, etc. A specifier can list his sorts after the keyword **sorts**. The sorts of the arguments and result of each function are declared in the signature, as in "$\neg$" BOOL $\rightarrow$ BOOL.

A closer look at sort and function declarations in algebraic specifications reveals that together they form a *context-free grammar*. Regarding sorts as nonterminals, and function declarations as grammar productions, we can read the $\neg$ declaration also from left to right: $\langle$BOOL$\rangle$ ::= "$\neg$" $\langle$BOOL$\rangle$. This provides a way to check whether a sentence denotes a term over a given signature: It must be possible to parse the sentence according to the derived grammar.

It would be ideal if such a parse attempt would yield one term corresponding to that sentence. Unfortunately context-free grammars are known to be possibly ambiguous: One sentence can sometimes be built in two (or more) different ways from the same grammar. To help overcome this problem, ASF+SDF supports disambiguation using priority declarations and function attributes. In module Booleans (Module 2.1), e.g., we have given $\wedge$ a higher **priority** than $\vee$, thus forcing *true* $\wedge$ *false* $\vee$ *true* to be parsed as (*true* $\wedge$ *false*) $\vee$ *true* rather than *true* $\wedge$ (*false* $\vee$ *true*). Likewise, we indicated that *true* $\wedge$ *true* $\wedge$ *true* is to be read as (*true* $\wedge$ *true*) $\wedge$ *true* rather than *true* $\wedge$ (*true* $\wedge$ *true*) by declaring $\wedge$ to be **left** associative. The symbols (and) declared as **bracket** can be used to override these priorities. The details of priority and bracket declarations are discussed in [HHKR89, Chapter 6].

Now we are finally in a position to explain the name "ASF+SDF" a bit further. Originally, an Algebraic Specification Formalism called ASF [BHK89] was developed, featuring modularization and parameterization, conditional equations, and hidden functions. It only allowed unambiguous prefix notation for writing down terms. A Syntax Definition Formalism named SDF [HHKR89] was designed later, aiming at providing one formalism to define both lexical, concrete, as well as abstract syntax. The combination of these two formalisms was based on the fixed mapping between signatures and context-free grammars. The result of the merge was baptized "ASF+SDF". It is an algebraic specification formalism which can be used to define context-free syntax for arbitrary languages, and which supports complete notational freedom for function names introduced in signatures.

2.2.3 Lexical Syntax

A **lexical syntax** section can be used to define basic lexical words like numbers (consisting of, e.g., a non-zero digit followed by zero or more digits) or identifiers (which in Identifiers (Module 2.2) are defined to be built from one or more alpha-numerical characters, starting with a letter, possibly including hyphens, and ending with zero or more prime ' characters).

module Identifiers
imports Layout$^{(2.3)}$
exports
  sorts ID
  lexical syntax
    $[a\text{-}z][\backslash\text{-}a\text{-}z0\text{-}9]{*}[\,']{*} \rightarrow$ ID
  context-free syntax
    $prime(\mathrm{ID}) \rightarrow \mathrm{ID}$
  variables
    "ξ" $\rightarrow$ CHAR+

equations

$$prime(id(\xi)) = id(\xi \text{ “}'\text{”}) \qquad [1]$$

Module 2.2: Identifiers. Function prime adds a quote to an identifier.

module Layout
exports
  lexical syntax
    $[\ \backslash t \backslash n] \rightarrow$ LAYOUT
    “$\%\%$”$\sim[\backslash n]{*} \rightarrow$ LAYOUT

Module 2.3: Layout. Definition of white space and comment

Besides defining the identifiers, the signature of module Identifiers introduces a function *prime*. This function will be used in the λ-calculus specification to produce new variables $prime(v)$, $prime(prime(v))$, ..., given a variable v. The equation of module Identifiers simply states that the result of applying a *prime* function to an identifier built from character list ξ is the same as concatenating a prime character to ξ. All details concerning the built-in sort CHAR can be found in [HHKR89].

As the reader may have noticed, module Booleans (Module 2.1) imports module Layout (Module 2.3). This is needed to define the exact format of "white space" used, e.g., in the equations. This is done using the predefined sort LAYOUT [HHKR89, Chapter 4]. Module Layout defines all spaces, tabs, newlines, or lines starting with two percent signs as being white space that can safely be ignored.

Finally, a rather surprising use of lexical syntax is made in SDF when declaring variables. Arbitrary regular expressions can be given to define the names of variables. Thus, a declaration like **variables** P *[0-9]*$*$ $\rightarrow$ BOOL defines infinitely many variables P, P_1, P_2, P_{01}, ..., all of sort BOOL.

2.2.4 Built-in Associative Lists

ASF+SDF supports associative list functions and variables [HHKR89, Hen91]. List functions have a varying number of arguments, and list variables may range over any number of arguments of a list function. As an example, module Id-Lists (Module 2.4) introduces lists of identifiers, with operations for membership test, concatenation, and element removal. The line "[" {ID ","}* "]" → ID-LIST defines terms like [ ], [*v1*], [*v1, v2*], ..., to be lists of 0, 1, 2, ..., elements. The asterisk * indicates zero or more elements, while the comma is the concrete representation for the separators (note that by definition they are separators rather than terminators, see [HHKR89, Chapter 5]). The [and] symbols serve as list delimiters. The list notation is an abbreviation for the declaration of infinitely many functions [...] each with a different number of arguments. When necessary, instead of an asterisk indicating "zero or more", the plus character can be used to indicate "one or more". Lists without separators can be defined by omitting the curly braces and the separator (e.g., ID+).

List variables are needed to define equations over list functions. Module Id-Lists defines the variables Ids_1 and Ids_2 as ranging over zero or more elements separated by commas. Equation [1] joins two lists; equations [2] and [3] remove all occurrences of one particular element from a list. Equations [4] to [6] define the membership function on lists.

2.2.5 Conditional Equations

To obtain more flexibility in algebraic specifications, *conditional equations* can be used. In module Id-Lists (Module 2.4) we have seen examples of the use of a *positive* condition [3], and a *negative* condition [6]. The idea of conditions is that the consequence (left of $\Leftarrow$) only holds if the sides of the conditions (right of $\Leftarrow$) can be proved equal or unequal. Alternatively, we may use a horizontal line to write conditional equations, as done, e.g., module Substitute (Module 2.7).

module Id-Lists
imports Identifiers$^{(2.2)}$ Booleans$^{(2.1)}$
exports
sorts ID-LIST
context-free syntax

$$
\begin{array}{lll}
\text{“[” \{ID “,”\}* “]”} & \rightarrow \text{ID-LIST} & \{\textbf{constructor}\} \\
\text{ID-LIST “−” ID} & \rightarrow \text{ID-LIST} & \\
\text{ID-LIST “}\cup\text{” ID-LIST} & \rightarrow \text{ID-LIST} & \{\textbf{left}\} \\
\text{ID “}\in\text{” ID-LIST} & \rightarrow \text{BOOL} &
\end{array}
$$

variables

$$
\begin{array}{ll}
\mathit{Ids}\ [12] & \rightarrow \{\text{ID “,”}\}* \\
[XY] & \rightarrow \text{ID} \\
\mathit{I\text{-}List} & \rightarrow \text{ID-LIST}
\end{array}
$$

equations

$$[\mathit{Ids}_1] \cup [\mathit{Ids}_2] = [\mathit{Ids}_1, \mathit{Ids}_2] \qquad [1]$$

$$\mathit{I\text{-}List} - X = \mathit{I\text{-}List} \quad \Leftarrow X \in \mathit{I\text{-}List} = \mathit{false} \qquad [2]$$

$$[\mathit{Ids}_1, X, \mathit{Ids}_2] - X = [\mathit{Ids}_1, \mathit{Ids}_2] - X \qquad [3]$$

$$X \in [\,] = \mathit{false} \qquad [4]$$

$$X \in [X, \mathit{Ids}_1] = \mathit{true} \qquad [5]$$

$$X \in [Y, \mathit{Ids}_1] = X \in [\mathit{Ids}_1] \Leftarrow X \neq Y \qquad [6]$$

Module 2.4: Id-Lists. Lists of Identifiers

Readers familiar with the theory of algebraic specifications will be aware of the fact that the use of negative conditions causes certain model-theoretic problems (see, e.g., [Kap88] or [BHK89, Chapter 9]). The use of negative conditions, however, typically reduces the size of the specifications significantly, which makes them indispensable when writing very large specifications. Moreover, a disciplined use is harmless. For instance, an inequality test over a sufficiently complete sort with respect to a set of free constructors is equivalent to a positive equality test between the Boolean value *false* and a Boolean equality predicate over that sort. See, e.g., [MS88, MS89] for a further analysis of these issues.

To facilitate the description of equations having an if-then-else like character, ASF+SDF supports the **otherwise**[2] construct. It can be used to cover "the remaining cases" in one equation. Using **otherwise**, the $\in$ function, which was defined using three equations one of which had a negative condition, can be defined by the following two equations:

equations

$$X \in [Ids_1, X, Ids_2] = true \qquad [5']$$

$$X \in \mathit{I\text{-}List} = false \qquad \textbf{otherwise} \qquad [4'6']$$

Again, **otherwise** equations are intended as being equivalent to a series of equations using positive conditions only. The so-called *complement set* of a term built of constructors only (see [Thi84]) can be used to construct this set of alternative positive conditional equations, as will be illustrated in Section 2.3.3. The **otherwise** construct is related to, e.g., the ordering given to clauses in Prolog, or the priority rewrite systems as discussed, e.g., by [BBK87, Moh89].

2.3 Example Specification: The λ-calculus

We will now illustrate the use of the features and concepts just introduced by discussing a small example. The modules to come define the syntax of the λ-calculus, as well as a series of useful operations on λ-terms.

The λ-calculus originated in the 1930s from the work of A. Church as a theory to study functions. Ever since, it has inspired many other important developments, such as Lisp, denotational semantics, or functional programming. By now, λ-calculus has grown into a major topic in programming language theory. It is used to study computation, design and semantics of programming languages, as well as specialized computer architectures [Gor88].

[2] Originally, the **otherwise** construct was proposed by Klint who used the name "default-equation". In the ASF+SDF system, default equations have a tag starting with "[default-...]".

module Lambda-Syntax
imports Identifiers$^{(2.2)}$
exports
  sorts L-EXP
  context-free syntax
    ID → L-EXP {**constructor**}
    "λ" ID+ "." L-EXP → L-EXP {**constructor**}
    L-EXP L-EXP → L-EXP {**constructor**, **left**}
    "(" L-EXP ")" → L-EXP {**bracket**}
  priorities
    "λ" "." < { L-EXP L-EXP → L-EXP }

  variables
    E [0-9]$*$ → L-EXP
    V [0-9]$*$ → ID
    Vs [0-9]$*$→ ID+

equations

$$\lambda\ Vs_1\ Vs_2\ .\ E = \lambda\ Vs_1\ .\ \lambda\ Vs_2\ .\ E \qquad [1]$$

Module 2.5: Lambda-Syntax. Syntax of λ expressions

2.3.1 Syntax of the λ-calculus

The consecutive lines of the context-free syntax section of module Lambda-Syntax (Module 2.5) define λ-expressions to consist of:

1. variables (identifiers) *x, y, ...*;
2. *abstractions* of the form $\lambda\ V_1 \cdots V_n\ .\ E$, where $n \geq 1$. We sometimes refer to $V_1 \cdots V_n$ as the *formal parameters*, and to E as the *body* of the abstraction. The ID+ indicates that we allow a list of one or more variables immediately after the λ.
3. Function *applications*: if E_1 and E_2 are λ-expressions, then so is $E_1\ E_2$. It denotes the result of applying function E_1 to an argument E_2.

The **left** declaration indicates that function application is left-associative, i.e., $E_1\ E_2\ E_3$ means $(E_1\ E_2)\ E_3$. The **priorities** declaration indicates that $\lambda\ V\ .\ E_1\ E_2$ is to be read as $\lambda\ V\ .\ (\ E_1\ E_2\ )$ rather than as $((\lambda\ .\ E_1)\ E_2)$ (i.e., the scope of the variable V extends as far to the right as possible). The **brackets** (and) can be used to override these conventions. In the variables section we have defined V_1, V_2, E_1, E_2, ... which we will use for arbitrary variables and λ-expressions respectively in the modules still to come.

module Variables

Free variables are defined according to [Bar84, Definition 2.1.7].

imports Id-Lists[(2.4)] Lambda-Syntax[(2.5)]

exports

context-free syntax

"FV"(L-EXP) → ID-LIST

get-fresh(ID, ID-LIST) → ID

equations

$$\mathrm{FV}(V) = [V] \qquad [1]$$

$$\mathrm{FV}(E_1\ E_2) = \mathrm{FV}(E_1) \cup \mathrm{FV}(E_2) \qquad [2]$$

$$\mathrm{FV}(\lambda\ V\ .\ E) = \mathrm{FV}(E) - V \qquad [3]$$

$$\frac{V \in \textit{I-List} = \textit{true}}{\textit{get-fresh}(V, \textit{I-List}) = \textit{get-fresh}(\textit{prime}(V), \textit{I-List})} \qquad [4]$$

$$\frac{V \in \textit{I-List} = \textit{false}}{\textit{get-fresh}(V, \textit{I-List}) = V} \qquad [5]$$

Module 2.6: Variables. Fresh and Free Variables

The single equation of the module states that $\lambda\ V_1\ ...\ V_n\ .\ E$ is just an abbreviation for $\lambda\ V_1\ .\ (\ ...\ .\ (\lambda\ V_n\ .\ E))$. Because of this equation we only need to deal with abstractions binding exactly one variable in the modules still to come. Note the use of list-variables Vs_1 and Vs_2 to match sequences of one or more identifiers (as indicated by ID+).

2.3.2 Substitutions

In Section 2.3.3 we will see how a function abstraction $\lambda\ x\ .\ E_1$ can be "called" with actual value E_2 by replacing all occurrences of the formal parameter x in body expression E_1 by the actual value E_2. Before doing so, we have to define substitutions themselves, which is done in module Substitute (Module 2.7). A substitution of expression E in expression E_1 for all free occurrences of variable V is denoted by $E_1[V := E\ ]$. A variable is *free* in an expression, if it is not bound by a λ abstraction. This notion of free variables is made precise by the first three equations of module Variables (Module 2.6). When defining substitutions $E_1[V := E]$ one has to take care that variables free in E do not become bound in $E_1[V := E]$. The specification does so, and defines so-called

module Substitute

imports Variables(2.6)
exports
  context-free syntax
    L-EXP "[" ID ":=" L-EXP "]" → L-EXP
  priorities
    "[" ":=" "]" > { L-EXP L-EXP → L-EXP }

equations

$$V[V := E] = E \qquad [1]$$

$$\frac{V_1 \neq V_2}{V_1[V_2 := E] = V_1} \qquad [2]$$

$$(E_1\ E_2)[V := E_0] = E_1[V := E_0]\ E_2[V := E_0] \qquad [3]$$

$$(\lambda\ V\ .\ E_1)[V := E_0] = \lambda\ V\ .\ E_1 \qquad [4]$$

Simple case where the $\lambda V_1.E_1$ does not require a renaming of bound variable V_1

$$\frac{V_1 \neq V_0,\ \ V_0 \in \mathrm{FV}(E_1) \wedge V_1 \in \mathrm{FV}(E_0) = \textit{false}}{(\lambda\ V_1\ .\ E_1)[V_0 := E_0] = \lambda\ V_1\ .\ E_1[V_0 := E_0]} \qquad [5]$$

Find a suitable renaming for V_1, i.e., a variable not yet occurring in E_0 or E_1.

$$\frac{\begin{array}{l} V_1 \neq V_0,\ \ V_0 \in \mathrm{FV}(E_1) \wedge V_1 \in \mathrm{FV}(E_0) = \textit{true}, \\ V_2 = \textit{get-fresh}(V_1, \mathrm{FV}(E_0) \cup \mathrm{FV}(E_1)) \end{array}}{(\lambda\ V_1\ .\ E_1)[V_0 := E_0] = \lambda\ V_2\ .\ E_1[V_1 := V_2][V_0 := E_0]} \qquad [6]$$

Module 2.7: Substitute. Valid substitutions, following the variable convention *valid* substitutions as defined[3] in [Bar84, Definition C.1]. As a result, the λ-expression $(\lambda\ y\ .\ y\ x\ )[x := y]$, for example, is equal to $\lambda\ y'\ .\ y'\ y$.

It is perhaps instructive to make some observations concerning the style used to define the substitution operation. It is defined by distinguishing the three constructor possibilities for its first L-EXP argument. The variable case is covered by equations [1] and [2], application by [3], and abstraction by [4],[5], and [6]. The variable and abstraction cases need further distinction, according to the variable argument of the substitution (the ID at the second argument position). This distinction is made using conditional equations. It is important to realize that the various conditions identify *disjoint* cases: the variables are either equal or not, and the complex Boolean expression in equations [5] and [6] is either true or false. Also note that the last condition of equation [6] acts like a let expression assigning a value to V_2 rather than a condition imposing a real restriction. Finally, it is easy to see that a substitution term operating on λ-expressions built of L-EXP-constructors only is always equal to a term also built of constructors only. In other words, the equations defining the _ [_ := _] operation are *sufficiently complete*.

2.3.3 Conversions

Conversion rules are ways to transform one λ-expression into another. Module Convert (Module 2.8) defines the so-called α, β, and η-conversions. The most important one is β-conversion, which simulates evaluating a function: $(\lambda\ V\ .\ E_1)\ E_2$ is by β-conversion equal to $E_1[V := E_2]$, i.e., by replacing the formal parameter V by an actual value E_2. (equation [1]). Functions that have the same form apart from the names of the bound variables denote the same function by α-conversion. Thus, $\lambda\ x\ .\ E$ can be replaced by $\lambda\ y\ .\ (E[x := y])$ provided y does not occur freely in E (equation [2]) [Bar84, p. 26]. By η-conversion, it is allowed to eliminate formal parameters from an abstraction if those parameters do not occur in the body. Thus, $\lambda V\ .\ E\ V$ is the same as just E, provided V does not occur in E

If a λ-expression E is not α, β, or η-convertible, (i.e., equations [1], [2], and [3] do not apply) then the **otherwise** equations [4], [5], and [6] guarantee that functions α, β, and η are equal to the unchanged expression E. As this is a relatively simple case, let us see how the **otherwise** construct can be eliminated for equation [5]. Instead of equation [5], we can also write the following four equations:

equations

$$\beta(V) = V \qquad \text{[b1]}$$

$$\beta(\lambda\ V\ .\ E) = \lambda\ V\ .\ E \qquad \text{[b2]}$$

$$\beta(V\ E) = V\ E \qquad \text{[b3]}$$

$$\beta(E_1\ E_2\ E_3) = E_1\ E_2\ E_3 \qquad \text{[b4]}$$

[3] Originally we used the definition of [Gor88, p.73], but that definition allows the result of $(\lambda f.f)[f' := f]$ to be $\lambda f'.f$, which is wrong. The forgotten case seems to be that the fresh variable generated should not be equal to the variable that is being substituted. Barendregt's definition recognizes that no fresh variable needs to be generated in this case.

module Convert
imports Substitute$^{(2.7)}$
exports
context-free syntax
α(L-EXP) $\rightarrow$ L-EXP
β(L-EXP) $\rightarrow$ L-EXP
η(L-EXP) $\rightarrow$ L-EXP

equations

$$\beta((\lambda\ V\ .\ E_1)\ E_2) = E_1[V := E_2] \qquad [1]$$

$$\frac{V_1 = \mathit{get\text{-}fresh}(V_0, \mathrm{FV}(E))}{\alpha(\lambda\ V_0\ .\ E) = \lambda\ V_1\ .\ E[V_0 := V_1]} \qquad [2]$$

$$\frac{V \in \mathrm{FV}(E) = \mathit{false}}{\eta(\lambda\ V\ .\ E\ V) = E} \qquad [3]$$

$\alpha(E) = E$ **otherwise** [4] $\beta(E) = E$ **otherwise** [5] $\eta(E) = E$ **otherwise** [6]

Module 2.8: Convert. α, β, and η conversions

These four cases correspond to the four *complement terms* of the term $(\lambda\ V\ .\ E_1)\ E_2$, occurring in the left-hand side of equation [1]. Equations [b1] and [b2] deal with the alternative constructors for the top application node. Equations [b3] and [b4] have an application as top symbol, but the first argument is different from an abstraction, i.e., it is a variable [b3] or an application [b4]. These four equations cover all possible cases that are not dealt with by equation [1]; i.e., they enumerate all possible "otherwise" situations. See [Thi84] for an algorithm to compute such a set of *complement* constructor terms.

2.3.4 Left-most Reductions

If equation [1] of Convert (Module 2.8) is applicable to an expression E, then E is called a *β-redex*. In general, a λ-expression may contain several β-redexes. Repeatedly applying β-conversion may produce a λ-expression in which no β-redex is present, a *normal form*. Whether a normal form is found may depend on the order in which β-reduction is applied to the redexes. A strategy that always leads to a normal form (if it exists at all) is left-most reduction, which repeatedly reduces the left-most redex

module Reduce
imports Convert[(2.8)]
exports
context-free syntax

$$
\begin{array}{ll}
\mathit{lm\text{-}step}(\text{L-EXP}) & \to \text{L-EXP} \\
\mathit{lm\text{-}red}(\text{L-EXP}) & \to \text{L-EXP} \\
\text{``}\mathit{has\text{-}\beta\text{-}redex?}\text{''}(\text{L-EXP}) & \to \text{BOOL} \\
\text{``}\mathit{is\text{-}\beta\text{-}redex?}\text{''}(\text{L-EXP}) & \to \text{BOOL}
\end{array}
$$

equations

$$\mathit{is\text{-}\beta\text{-}redex?}((\lambda\ V\ .\ E_1)\ E_2) = \mathit{true} \qquad [\text{h0}]$$

$$\mathit{is\text{-}\beta\text{-}redex?}(E) = \mathit{false} \qquad \textbf{otherwise} \qquad [\text{h1}]$$

$$\mathit{has\text{-}\beta\text{-}redex?}(V) = \mathit{false} \qquad [\text{h2}]$$

$$\mathit{has\text{-}\beta\text{-}redex?}(\lambda\ V\ .\ E) = \mathit{has\text{-}\beta\text{-}redex?}(E) \qquad [\text{h3}]$$

$$\mathit{has\text{-}\beta\text{-}redex?}(E_1\ E_2) = \mathit{is\text{-}\beta\text{-}redex?}(E_1\ E_2) \lor \mathit{has\text{-}\beta\text{-}redex?}(E_1) \lor \mathit{has\text{-}\beta\text{-}redex?}(E_2) \qquad [\text{h4}]$$

$$\mathit{lm\text{-}step}(\lambda\ V\ .\ E) = \lambda\ V\ .\ \mathit{lm\text{-}step}(E) \qquad [\text{m1}]$$

$$\mathit{lm\text{-}step}(V) = V \qquad [\text{m2}]$$

$$\frac{\mathit{is\text{-}\beta\text{-}redex?}(E_1\ E_2) = \mathit{true}}{\mathit{lm\text{-}step}(E_1\ E_2) = \beta(E_1\ E_2)} \qquad [\text{m3}]$$

$$\frac{\mathit{is\text{-}\beta\text{-}redex?}(E_1\ E_2) = \mathit{false},\ \ \mathit{has\text{-}\beta\text{-}redex?}(E_1) = \mathit{true}}{\mathit{lm\text{-}step}(E_1\ E_2) = \mathit{lm\text{-}step}(E_1)\ E_2} \qquad [\text{m4}]$$

$$\frac{\mathit{is\text{-}\beta\text{-}redex?}(E_1\ E_2) = \mathit{false},\ \ \mathit{has\text{-}\beta\text{-}redex?}(E_1) = \mathit{false}}{\mathit{lm\text{-}step}(E_1\ E_2) = E_1\ \mathit{lm\text{-}step}(E_2)} \qquad [\text{m5}]$$

$$\frac{\mathit{has\text{-}\beta\text{-}redex?}(E) = \mathit{true}}{\mathit{lm\text{-}red}(E) = \mathit{lm\text{-}red}(\mathit{lm\text{-}step}(E))} \qquad [\text{m6}]$$

$$\frac{\mathit{has\text{-}\beta\text{-}redex?}(E) = \mathit{false}}{\mathit{lm\text{-}red}(E) = E} \qquad [\text{m7}]$$

Module 2.9: Reduce. Left-most reduction

module Let
imports Lambda-Syntax$^{(2.5)}$ Substitute$^{(2.7)}$
exports
  sorts DEF LET
  context-free syntax
    expand(L-EXP, LET) $\rightarrow$ L-EXP
    "(" ID ":" L-EXP ")" $\rightarrow$ DEF {**constructor**}
    "(" "*let*" DEF+ ")" $\rightarrow$ LET {**constructor**}
  variables
    D [*0-9'*]$*$"+"$\rightarrow$ DEF+
    D [*0-9'*]$*$ $\rightarrow$ DEF

equations

$$expand(E, (let\ (V : E_1))) = E[V := E_1] \quad \text{[e1]}$$
$$expand(E, (let\ D^+\ D)) = expand(expand(E, (let\ D)), (let\ D^+)) \quad \text{[e2]}$$

Module 2.10: Let. Abbreviations

[Gor88, p.121]. Reduce (Module 2.9) defines left-most reductions on λ-expressions. The function *lm-step* yields the result of exactly one left-most step. It uses the auxiliary function *has-β-redex?* to find the left-most redex. The function *lm-red* repeats left-most steps until the λ-expression no longer changes. If a λ-expression E has a normal form, then *lm-red*(E) is equal to that normal form. Module Reduce only defines left-most β-reduction. It can easily be extended to cover η-reduction as well, but we omitted this to keep our example simple.

Note that in this module the sufficient completeness is lost: the function *lm-red* operating on a "looping" λ-expression like $(\lambda\ x\ .\ x\ x)(\lambda\ x\ .\ x\ x)$ cannot be eliminated, and hence is not equal to a term built of the three L-EXP constructors only (variable, application or abstraction).

2.3.5 λ-definitions

Besides being a language for reasoning about functions, the λ-calculus is used to represent all kinds of objects. Similar to the way in which in set theory natural numbers can be represented by the sets $\emptyset, \{\emptyset\}, \{\emptyset, \{\emptyset\}\}, ...$, all kinds of objects can be represented by λ-expressions. Module Let (Fig. 2.10) introduces notation for such λ-definitions. For example, in the classical method of Church, a number N is represented by the normal form $\lambda\ f\ x\ .\ f^N\ x$. A way to obtain this is by defining:

$(let\ (zero : \lambda\ f\ x\ .\ x)$
  $(succ : \lambda\ n\ f\ x\ .\ n\ f\ (f\ x)))$

According to these definitions, *succ* (*succ zero*) can be β-reduced to $\lambda\, f\, x\, .\, f\,(f\, x)$. In order to use such definitions in λ-expressions, the names must be replaced by their definitions. This is specified by the *expand* function. It is possible to extend the Let (Module 2.10) to cover *letrec* structures for recursive functions as well (which can be rewritten to a *let* containing a fixed point operator, and an application of this operator to the function defined in the *letrec*). We omitted this to keep the specification as small as possible.

2.4 Generating Environments

Now we move from the ASF+SDF specification formalism to the ASF+SDF tool generator. We discuss various basic tools that can be derived automatically, and consider the composition of integrated, interactive environments from basic tools. We illustrate the various ideas using the generated λ-calculus-environment.

2.4.1 Basic Tools

In Section 2.2.2 we encountered *parsing* as the technique to check whether a string is built according to a given signature. *Parsers* can be generated automatically from the sorts, lexical and context-free syntax, and priorities sections of an ASF+SDF module. The parser generation facility of ASF+SDF does not impose any restrictions (such as being LR, LL, LALR, ... [ASU86]) on the input context-free grammar. The parsing technique used is generalized LR parsing [Tom85, Rek92].

An editor which knows about the intended syntax of the texts entered is called a *syntax-directed editor*. In addition to the text, such an editor maintains an abstract syntax tree of the text. Typical tasks of the editor are to perform checks on the syntactic correctness, to inform the user about the syntactic structure, and to allow *structural editing* involving the replacement of non-terminal symbols (place holders, sort denoters, meta-variables) by grammar productions (context-free functions). The editor developed for ASF+SDF is called GSE (Generic Syntax-Directed Editor). The most important concept in GSE's editing model is the so-called *focus* which designates a subtree of the full abstract syntax tree. Ordinary text editing is allowed within the focus [Koo94].

Term Rewriting Systems (TRSs) can be used to execute ASF+SDF specifications. In Section 2.2.1 we have seen a small example of a rewrite according to the equations of module Booleans (Module 2.1) reducing $true \wedge \neg\, false$ to $true$. The assumption made in ASF+SDF is that all equations can simply be oriented from left to right.[4]

Repeated application of rewrite rules can result in normal-forms, i.e., terms to which no rewrite rule is applicable anymore. When using term rewriting to execute algebraic specifications, it is desirable that the rules are *normalizing* (no non-termination), and confluent (resulting in unique normal forms). The ASF+SDF experience here is in accordance with the OBJ observation that "we have found that experienced programmers usually write rules that satisfy these properties" [FGJM85, p.54]. Notice, however, that sets of rewrite rules describing an operational semantics (an interpreter) of some language usually *cannot* be terminating. E.g., the *lm-red* function of module Reduce (Module 2.9) does not normalize when applied to a looping λ-expression.

Conditional equations are executed by first trying to normalize the various condition sides. Only if the two condition sides of each condition yield the same normal form, the conclusion is applied as a rewrite rule. In the terminology of [BK86], conditional rewrite rules are executed as *join* systems. Finally, as seen in the last condition of equation [6] in module Substitute (Module 2.7), the ASF+SDF system can deal with conditions where at most one side introduces variables not yet occurring in the left-hand side of the conclusion. This typically is used as a "let"-like construct, giving new variables a temporary value. The new variables obtain a value by matching with the normalized non-introducing condition side.

Term rewriting issues are discussed in full depth by [Klo92]. Several interesting details concerning the ASF+SDF rewrite implementation, covering such issues as matching, rewrite strategies, and conditional rewriting are discussed in [Wal91].

2.4.2 The Generated λ-calculus Environment

Instances of the various tools mentioned can be generated from the λ-calculus specification and combined into an integrated λ-calculus environment. An editing session in such an environment is shown in Figure 2.1. It displays four windows, each containing a syntax-directed editor. These editors are extended with buttons to apply specified functionality to the subterm in the focus. The large window is the *Definitions* editor containing the user's favorite λ-definitions. In the three smaller editors λ-expressions can be manipulated. The subexpression in the focus can be changed by the various buttons attached to each λ-editor. There are buttons to α, β, or η-convert the focus, to perform one left-most reduction step, or to reduce the expression in the focus by left-most reduction to its λ normal form. The expand button replaces (within the

[4] The best known examples of equations for which this assumption leads to termination problems deal with commutativity, e.g., $x + y = y + x$. In fact, the reason we have been using *lists* rather than *sets* in module Id-Lists (Module 2.4) is to avoid the equation

$$[Ids_1, X, Ids_2, Y, Ids_3] = [Ids_1, Y, Ids_2, X, Ids_3]$$

which expresses that the order of elements in sets is irrelevant. A solution is rewriting based on *commutative matching* as used in, e.g., OBJ [GWM+92]. We will not concern ourselves with commutative matching in this thesis.

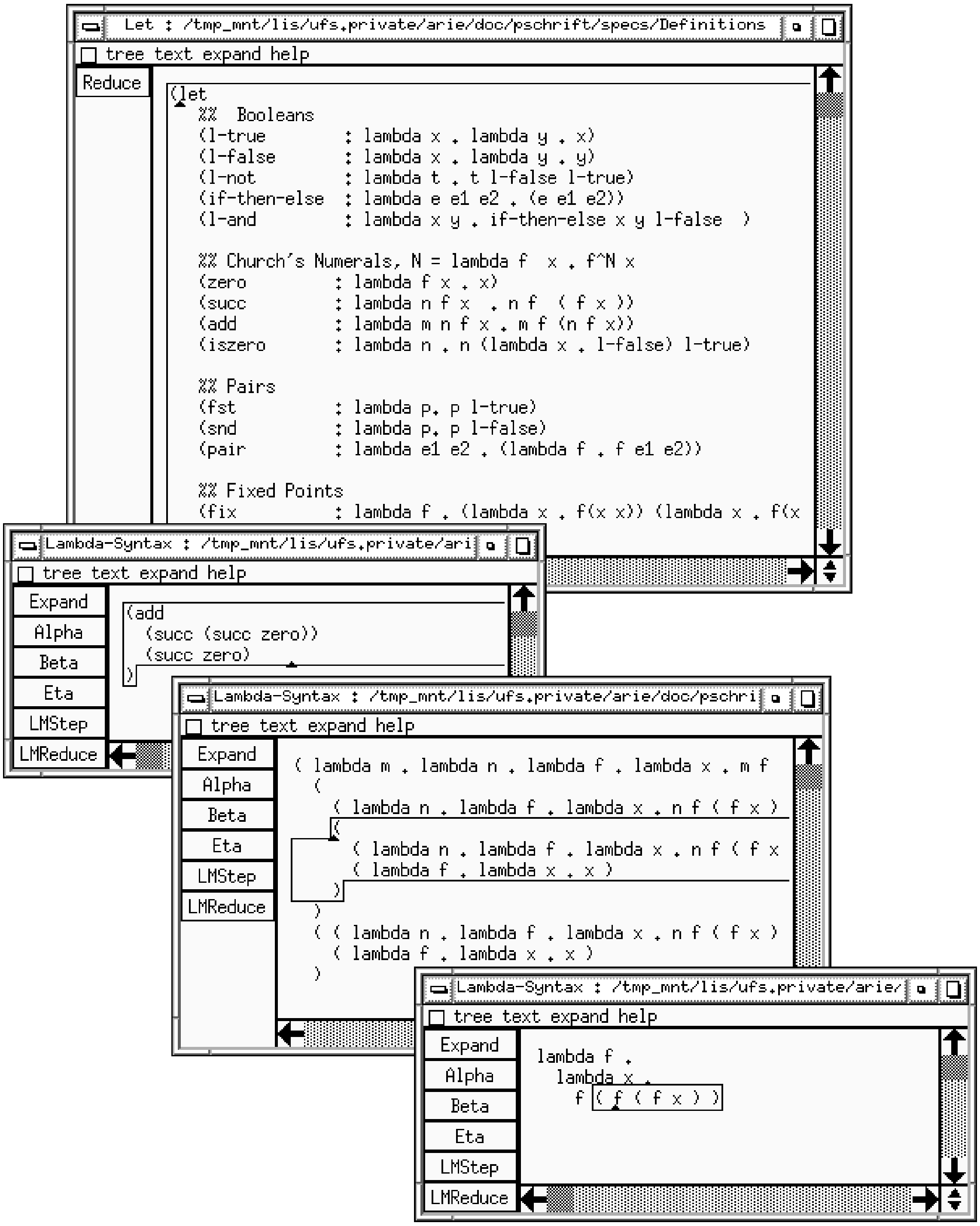


Figure 2.1: Example of the generated λ-calculus environment

focus) all occurrences of λ-defined identifiers by their corresponding definition given in the *let*-construct of the big *Definitions* editor.

As an example of the practical use of such a generated environment, let us consider λ-definitions of numerals. Wadsworth [Wad80] gives several alternative λ-definitions for numbers, and proves various propositions for them. To develop some intuition concerning his definitions, one could edit the λ-definitions in a *Definitions*-editor, and add Wadsworth's numbers:

$$
\begin{array}{l}
(\mathit{let}\,(k : \lambda\ x\ y\ .\ x) \\
\quad (i : \lambda\ x\ .\ x) \\
\quad (\mathit{wzero} : k\ i) \\
\quad (\mathit{wsucc} : k))
\end{array}
$$

Now a term like *wsucc* (*wsucc wzero*) can be entered in some λ-editor. In this editor it is possible to experiment with the Wadsworth numeral representations by clicking the various buttons with the focus at different positions, thus performing α, β, η-conversion, or left-most reduction (steps) on any desired subexpression. The intuition thus gained may help in conjecturing, proving, or refuting statements about Wadsworth's λ-definitions for numbers.

2.4.3 From Tools to Environment

One question still open is how the generated λ-calculus environment should know that the button "Beta" corresponds to function β as defined in module Reduce (Module 2.9), that "LMStep" corresponds to *lm-step*, and most miraculously how button "Expand" knows that its second input argument comes from the Definitions editor and its first from the editor in which it was invoked. In ASF+SDF, this final connection between specified functionality and user-interface events is made using the script language SEAL[5] [Koo94]. SEAL is an abbreviation for Semantics-Directed Environment Adaptation Language. It has various primitives to describe focus movements and replacements, application of functions to terms occurring in editors, etc.

The SEAL description of the λ-calculus environment consists of six button definitions for Expand, Alpha, Beta, Eta, LMStep, and LMRed. Two of these button descriptions are shown in Figure 2.2. The second one is for Beta and is the simplest: It states that if we are on any λ-expression (i.e., when the focus is of sort L-EXP), we can replace the focus by the result of applying function β as defined in module Convert (Module 2.8) to the old focus. The first button definition deals with expand. The focus of a file called "Definitions" is retrieved and set to the root position (i.e., covering the entire text in the editor). The term in this focus is passed as argument to the *expand* function as defined in module *Let*.

The full details of the SEAL language and its implementation are described by Koorn [Koo94]. He also gives a much more elaborate SEAL script for the λ-calculus

[5]Unless one is content with the "environment" generated by default in which case the use of SEAL is not needed. This standard environment just consists of a syntax-directed editor with one button "Reduce" to rewrite the term in the editor to normal form.

Configuration for language *Lambda-Syntax* **is**

button *Expand*
    when focus is L-EXP
    enable *FocusVar* := **focus** ;
        LetDefs := "*Definitions*" . **focus root**;
        focus := *Let* : *expand*(*FocusVar*, *LetDefs*)
    doc : "*Expand names in an expression using file 'Definitions'* "

button *Beta*
    when focus is L-EXP
    enable *FocusVar* := **focus** ;
        focus := *Convert* : β(*FocusVar*)
    doc : "*Beta-reduce the focus-expression*"

Figure 2.2: Definition in SEAL of some of the λ-calculus environment buttons.

environment, dealing for instance with an undo facility as well as enabling and disabling the Alpha, Beta, Eta, LMStep, LMRed buttons, depending on whether or not the focus is an α, β, or η redex, or contains a β-redex.

2.5 The Asf+Sdf Meta-environment

The Asf+Sdf system acts as an environment itself, in that it gives support during the development of the specifications used to generate other environments. For this reason, it is called the Asf+Sdf Meta-environment. We conclude with a brief description of the Meta-environment's main features.

2.5.1 Specification Development Support

The Meta-environment supports syntax-directed editing of both signature parts and equations of Asf+Sdf specifications. The syntax-directed editors for the equations have knowledge of the context-free syntax introduced in the signature part. In addition to syntactic checks on Asf+Sdf specifications, context-sensitive analysis is performed in order to detect such errors as priority declarations referring to undefined functions or the use of undeclared sorts. The equations are inspected for their executability as rewrite rules. For instance, a warning is given if the right-hand side of an equation uses a variable not occurring in the left-hand side.

An example editing session during the development of the λ-calculus specification is shown in Figure 2.3. The two large windows contain the syntax and equation editors for modules Lambda-Syntax and Variables respectively. The smaller window is called "ASF+SDF Meta-environment". It is the main window, used to open new modules, start editing terms, start reducing in debugging mode, etc. It is also used to deal

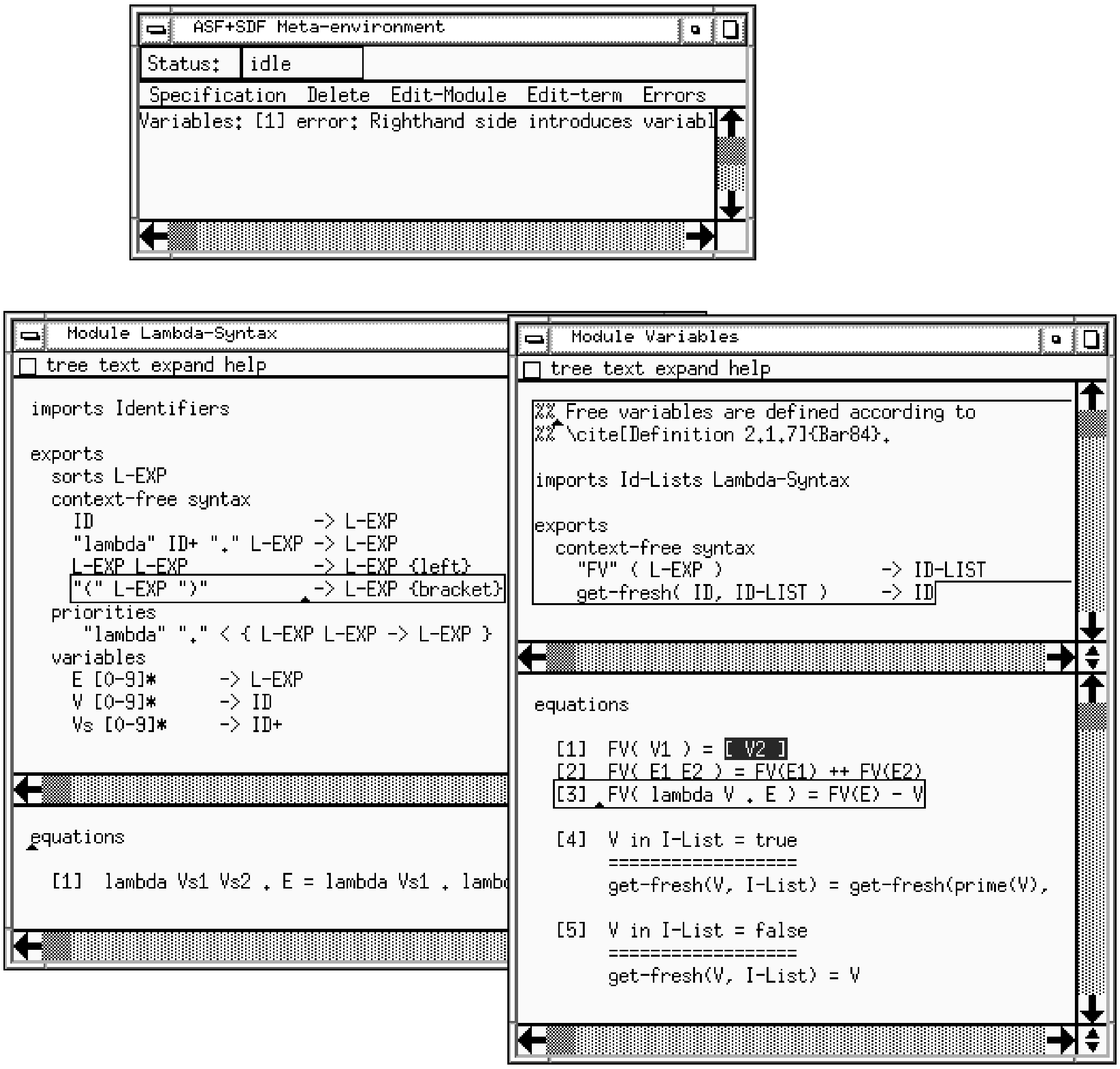


Figure 2.3: Meta-environment while working on the λ-calculus specification

with error handling. All errors appear in this window, and for each error a click on it gives more information concerning that error. In the figure, the specifier has just clicked on the error message displayed. As a result, module Variables is raised, and the relevant text part, in this case the right-hand side of the first equation, is marked via highlighting.

In addition to these analysis features of the Meta-environment, specifications can be executed and tested. A tricky term like $(\lambda\ y\ .\ y\ x\ )[x := y]$ can be entered and reduced to normal form in order to see whether the defined substitution function behaves as intended. Attempts can be made to verify or refute the conjecture that the specification indeed models the behavior one had in mind.

The analysis and testing of specifications is helped by the Meta-environment's most distinctive feature, namely its high level of incrementality. Syntactic analysis of the equations is possible because a parser is generated from the function declarations. In the ASF+SDF Meta-environment, a change to the function declarations, like adding a function, or removing one, does not lead to a regeneration from scratch of the parsers involved. Instead, they are informed about the change and updated accordingly. In other words, parser generation proceeds incrementally. Not only parsers, but also lexical analyzers and term rewriting systems are updated incrementally [Hen91, HKR90, HKR92, HKR94].

A last important feature of ASF+SDF is the support for literate specifications [Vis93], in the sense of the literate programming approach advocated by Knuth [Knu92]. For instance, the text as entered by the specifier for modules Lambda-Syntax and Variables is visible in the windows of Figure 2.3. The literate specification tool automatically transforms this ASCII representation into the formatted representation as shown in Modules 2.5 and 2.6, treating the commented lines (as in module Variables) as LaTeX lines. The use of the literate specification tool invites a specifier to write readable ASF+SDF modules and to produce structured documentation.

2.5.2 Ongoing Activities

The ASF+SDF formalism and Meta-environment have been used for a wide variety of research activities. A typical application is to experiment with descriptions and tools for new languages. For example, ASF+SDF has been used during the development of the aforementioned SEAL language [Koo94], for the formalization and interpretation of the syntax and operational semantics of the Manifold parallel coordination language [RT92] or to obtain tools for the program analysis language PIM proposed by Field [Fie92]. In addition to these applications as a language development tool the ASF+SDF Meta-environment has been used intensively to design and implement new tool generation technology. Besides the already mentioned incrementality, user-interface adaptation, syntax-directed editing, and literate specification, topics recently studied (and implemented) include the ASF2C project aiming at the the translation of equations to highly efficient C code [KW93], coupling of an Emacs-like editor to a syntax-directed environment [KB93], and the derivation of incremental type checkers from static semantic specifications using incremental rewriting [Meu94].

2.5.3 Related Systems

Looking at a micro-historical scale (see also [HK94]), the origins of ASF+SDF can be traced back to the design and implementation of the string processing programming language SUMMER [Kli80]. The wish to have a programming environment for this language, and the intention to use language definitions practically [Kli85, Chapter 4], led to the more general study of the generation of programming environments from language definitions. Experiments with formal language definitions using algebraic specifications guided the design of ASF [BHK89, Chapter 1 and 2], initially implemented using Prolog [Hen88]. In 1985, a cooperation with INRIA (Sophia Antipolis) started in a project titled "Generation of Interactive Programming Environments".

The Centaur system [BCD+89] written in LeLisp became the implementation platform. A first version of the ASF+SDF Meta-environment was operational in the summer of 1990. The rivalry with the other specification language in Centaur, the Metal [KLMM83] and Typol [Kah87] combination, has been a fruitful source of inspiration for the development of the ASF+SDF formalism and system.

In a wider perspective, ASF+SDF builds on several other tool generators based on formal language definitions. Some well-known compiler generator examples (based on denotational semantics) are Mosses' Semantics Implementation System (SIS) [Mos79], or Wand's [Wan84] and Paulson's [Pau82] prototyping systems. More recent support systems for language descriptions are, e.g., [LP87, Tof90, Ber91a, BMW92, Mic93]. Systems aiming at the generation of integrated environments are the earlier mentioned PSG system [BS86] and the Synthesizer Generator [RT89a], as well as the Gandalf system [HN86], the Pan language-based system [BGV92], or the Eli system dealing with executable specifications for language implementations [Kas93].

OBJ [GWM+92] is one of the best-known and most influential algebraic specification languages, so let us take a closer look at differences and similarities between OBJ and ASF+SDF. An important difference between OBJ and ASF+SDF is that OBJ supports parameterized modules, views on modules, and theories describing the behavior of modules (the ASF+SDF *formalism* was designed to support parameterization and renaming, but these have not yet been implemented in the ASF+SDF system). Moreover, the rewrite engine of OBJ is more general in that it implements associative commutative matching, whereas ASF+SDF only implements matching modulo associative lists. However, the rewrite engine of ASF+SDF is, according to [Eke92], a factor four faster than OBJ. On the syntactic side, it should be noted that OBJ is order-sorted, whereas ASF+SDF uses *chain*-functions to achieve the injection of smaller sorts into larger ones. ASF+SDF has no automatic retracts. A significant difference is that ASF+SDF in addition to being an algebraic specification formalism, is intended as input language for a programming environment generator. Therefore, the ASF+SDF formalism supports the definition of lexical syntax and (which is forbidden in OBJ) empty productions (invisible constants) which can be used to model optional constructs. Moreover, it is more flexible concerning *brackets*. OBJ gives special treatment to parentheses (...), which only serve to group and never occur in abstract syntax trees. In SDF, specifiers can choose their own syntax for brackets (e.g., ⟨...⟩ or {...} pairs). Moreover, parentheses (...) can also be used as genuine functions, for instance as tuple constructors. In OBJ this is forbidden (causing, e.g., problems as described by Martin [Mar93]).

Other systems supporting the development of algebraic specifications include Asspegique [BC85, BGM89, Voi86], Act-one [Han87], or the Equation Interpreter [O'D85]. Features not available in ASF+SDF are proof support as in Larch [GG89], narrowing as in RAP [Hus88], completion as in REVEUR4 [BR88] or in RRL [Zha92], or persistency checks as in Perspect [Wie91].

Chapter 3

A Language for Defining Financial Products

We report on the use of formal methods and supporting tools during the development of a language applied in a banking environment. The language designed, called RISLA, aims at defining the nature of the interest products offered by a bank. Its emphasis is on fixing the resulting cash flows (amounts of money coming in or going out on particular dates) for each of the products. RISLA has been formally defined using the algebraic specification formalism ASF+SDF. The development of this specification was supported by the ASF+SDF Meta-environment. RISLA is now being used intensively to handle all interest product transaction of Bank MeesPierson. The role of the formalization using algebraic specifications and ASF+SDF is described, discussed, and evaluated.

3.1 Introduction

A typical bank offers various sorts of *interest products*, i.e., products where a rate of interest, either fixed or depending on the market, is paid over a certain amount (the *principal* amount) of money. Numerous slightly different products exist, each varying, e.g., in the risk taken by the buyer of the product, or in the way it anticipates an increasing or decreasing interest rate. The profit that can be made when dealing in such products is considerable.

The market on which these products are dealt in is dynamic; any bank can create a new kind of product. Whether offering such a new product will result in increased profits or growing losses largely depends on the bank's abilities to offer such a product faster than the competition can.

Offering a new product to a large number of clients, however, is not without consequences for the bank's automated systems. For example, the *financial administration* needs to be able to process the new products (who bought them, what are the interest rates agreed upon, what are the actual amounts over which interest is to be paid, over which intervals should the interest be computed, and so on). Moreover, *management information systems* should provide the bankers with up to date information concerning the risk the bank is taking when, e.g., the central interest rates suddenly start rising.

How can the bank's automated systems be made sufficiently flexible such that they can easily process new kinds of products? CAP Volmac[1], commissioned by Bank MeesPierson[2], started working out the following scenario. For each interest rate product, a high-level description is given, fixing the essence of the product. The basic assumption is that a product can be characterized by describing the *cash flows* it causes. A cash flow is a series of $\langle$amount, date$\rangle$ pairs, indicating an amount of money leaving or entering the bank at a certain date. The kernel of a product description consists of a set of rules defining how to compute cash flows from the initial contract values. These product descriptions suffice to *generate* the necessary software automatically. The software generated from a description consists of a number of COBOL procedures that can be invoked by the bank's automated systems. The COBOL routines can answer specific questions about products entered, as required by the financial administration of management information systems.

At the start of the project, about two years ago, it was very unclear whether this approach could be successful. By now, after having tackled many problems, full implementation work is going on. The system built should be fully operational in the autumn of 1994. It should be able to process all interest rate products of Bank MeesPierson. Early experiments showed that, in this scenario, the time delay caused by updating the automated systems upon the introduction of a new product has been reduced to just a few days, as opposed to several months in the "classical" approach (adapting the systems by hand).

At the heart of this approach are the product descriptions. CAP Volmac designed a small specific language called RISLA[3] for the purpose of defining interest rate products. As neither CAP Volmac nor Bank MeesPierson had much experience with language design, they contacted CWI, the Dutch national center for mathematics and computer science, to assist the development of RISLA. CWI, CAP Volmac, and Bank MeesPierson, agreed to start a two person-month project aimed at further developing the RISLA application language. An ASF+SDF definition of RISLA was to be given, and some prototype tools, such as a syntax-directed editor for RISLA, were to be generated. Moreover, a selection of ten representative interest rate products had to be described using RISLA, in order to illustrate its expressive power.

This chapter describes the role of the ASF+SDF formalism and Meta-environment during the development of RISLA.

3.1.1 Related Work

In addition to developing the new language RISLA, CAP Volmac, CWI, and the University of Amsterdam (UvA), studied whether an *existing* language could be used. As it was conjectured that interest products involved certain "processes", like "pay interest", or "redeem a loan", a language tailored towards process description was

[1] The work on RISLA was in particular initiated by the Orfis group — the Organization for Financial Information Systems; now part of CAP Volmac.

[2] The cooperation originally started with Bank Mees & Hope.

[3] A Dutch/English acronym for Rente (interest) Informatie Systeem LAnguage. Earlier documents use "RPM" as language name, an acronym for Rente Produkt Modelleringstaal.

chosen, namely the Process Specification Formalism PSF [MV90]. In a pilot study, four interest products were defined using PSF. Surprisingly, the process specification facilities were *not* used to define these products; the definitions only involved functions and data structures.

Finally, it was decided to stop experimenting with existing languages. The main reasons for this were that RISLA was by that time sufficiently developed and (i) it seemed that all products could easily be defined in RISLA; (ii) these product descriptions looked the way CAP Volmac and Bank MeesPierson wanted them to look; and (iii) RISLA was so small that the extra time needed to build a specialized compiler was considered worth the effort.

A completely independent, but remarkably similar project is described by Eggenschweiger and Gamma from the Union Bank of Switzerland [EG92]. They describe how ET++, a class library for C++, has been used to implement a swap valuation system.

3.2 Product Descriptions

A RISLA product description defines a class of *contracts*. A contract is established between clients of the bank, and fixes the actual amounts over which interest is paid, the interest rates used, etc. Different products yield different types of contracts.

One of the simplest products is the *deposit*. A deposit is an agreement between a client and the bank that the client is to deposit an amount of money in a given currency for a period of time at the bank, for which he will receive a percentage of interest, to be paid at certain intervals. These are the *parameters* of the product, defined in the product description. An actual contract may fix a deposit to 1 million Dutch guilders starting at 1 January 1995 until 1 January 2000 at an interest of 6.5%, the interest to be paid every year.

Such a deposit will give rise to two streams of cash flows: The *Principal Amount Flow* (PAF) will simply consist of 1 million guilders on January 1st 1995, and −1 million guilders on January 1st, 2000. The *Interest Amount Flow* (IAF) consists of the −65,000 guilders going out every year that the interest is actually paid.

A very simple deposit described using RISLA is shown in Figure 3.1. It defines the PAF and the IAF of a deposit involving Dutch guilders. The next section will elaborate on the syntax and built-in functions used in the definition of the simplified deposit.

Most interest products are variations on the deposit. A typical one is the *Currency Interest Rate Swap*, where the client is another bank. The two banks swap an amount of money in one currency for an equivalent amount in a different currency. One bank pays a fixed interest rate agreed upon in the contract, the other bank pays a periodically reset market rate. Again this product results in PAF and IAF flows.

The software to be generated from a product description includes routines to enter new contracts over that product, and functions (methods) to be applied to contracts yielding, e.g., the PAF or IAF flows.

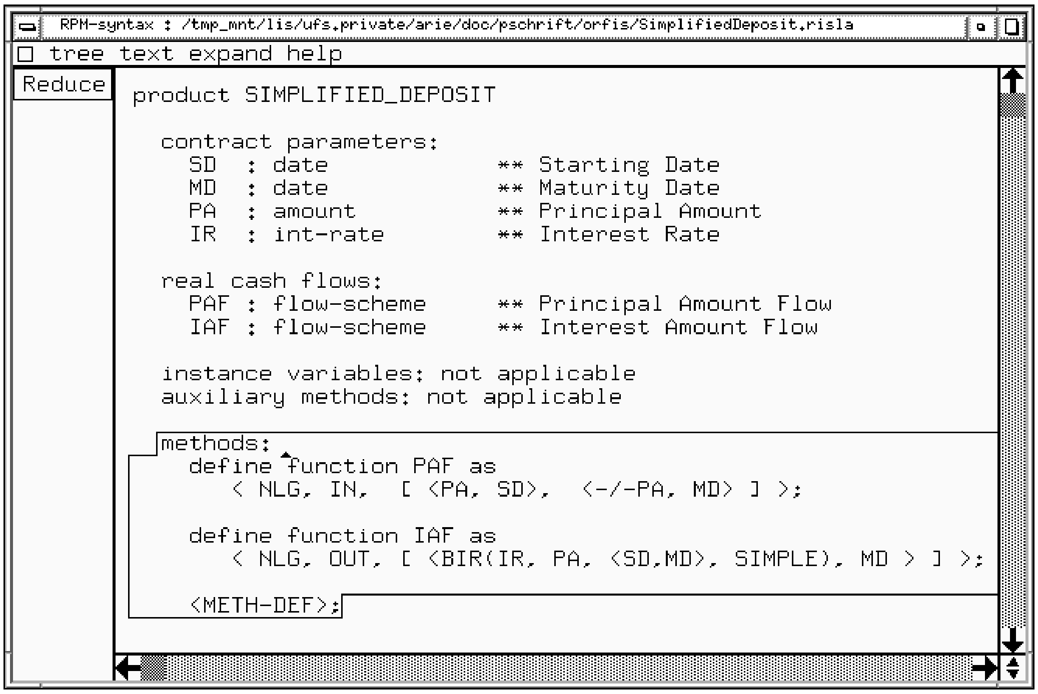


Figure 3.1: Example of a very simple RISLA product definition

3.3 Defining RISLA using ASF+SDF

3.3.1 Before ASF+SDF

At the start of the two-month project, a folder [AG92] concerning the language RISLA was available. This folder contained a large number of product definitions. The notation used looked relatively stable, although small variations were visible between different products, and certain descriptions required ad-hoc extensions of the notation.

Various built-in functions were assumed. Some of these dealt with date manipulation, for instance when computing the number of working days during which interest has to be paid; depending on the currency certain days will not count as they are holidays (e.g., the fourth of July or 14me Juillet). Other functions reflected operations that were frequently needed when manipulating interest computations. These functions were either based on existing COBOL routines, or informally described in natural language.

No context-free (BNF) grammar for RISLA was given. The typing of RISLA product definitions and expressions was implicit; it was assumed that certain operations were only allowed on operands of particular types, but this was not explicitly stated.

3.3.2 RISLA Data Types

A first task when defining a language like RISLA is to get a clear picture of the underlying data types and built-in operations.

A *cash flow* is the most important data type of RISLA. It can be modeled as a list of ⟨amount, date⟩ pairs ordered by date, In ASF+SDF module Flows (Module 3.1) cash flows are specified algebraically, introducing several elementary operations on flows. For example, function *merge* takes two cash flow lists and merges them into one. Function *nett* accumulates all elements with the same date to a single flow element, netting the various amounts.

Related to cash flows are *balances*. Whereas a cash flow represents money leaving or entering the bank, the balance records the amount of money currently available to the bank. In other words, a balance is a list of ⟨amount, interval⟩ pairs, indicating that a certain amount of money will be available for the bank during the given date interval. As for cash flows, an algebraic specification for balances has been written, although we did not include the full ASF+SDF module in this chapter. The interested reader is referred to [AD92, Appendix A.1.2]. An example computation over module Balances is shown in Figure 3.2. It shows the function "bal", which takes a cash flow, an interval, and an initial amount, and computes the resulting balance for the given interval.

Algebraic specifications like the one given for the cash flows of module Flows were given for most data types of RISLA [AD92, Appendix A.1]. For some of the data types, such as "market" or "time", only a signature was given. Overall, 15 modules, covering 25 pages, using about 100 equations were needed to specify these data types (excluding the modules defining standard data types like Booleans and Integers).

Even though constructing these data type specifications was a straightforward task, it had a very clarifying effect. Before, a question like "what is a cash flow" resulted in an answer explaining that it was something which could be used to deal with flows of money resulting from interest products. Later on, the answer became that a cash flow is an element of the sort CASH-FLOW as specified in module Flows. Secondly, the specifications fixed the argument types and result types of many of the built-in functions operating on the data types. In most cases this was again straightforward, but in others various people involved in the project had different opinions about the number of arguments given to functions used in RISLA definitions. In those cases, the formal specification served as a fruitful design aid.

module Flows
imports Dates
exports
sorts CASH-FLOW FLOW-LIST
context-free syntax

"⟨" "*amount*:" INT "," "*date*:" DATE "⟩"	$\rightarrow$ CASH-FLOW
CASH-FLOW "." *date*	$\rightarrow$ DATE
CASH-FLOW "." *amount*	$\rightarrow$ INT
CASH-FLOW "." *amount* ":=" INT	$\rightarrow$ CASH-FLOW
"[" {CASH-FLOW ","}* "]"	$\rightarrow$ FLOW-LIST
ins(CASH-FLOW, FLOW-LIST)	$\rightarrow$ FLOW-LIST
merge(FLOW-LIST, FLOW-LIST)	$\rightarrow$ FLOW-LIST
nett(FLOW-LIST)	$\rightarrow$ FLOW-LIST
split-before(FLOW-LIST, DATE)	$\rightarrow$ FLOW-LIST
split-after(FLOW-LIST, DATE)	$\rightarrow$ FLOW-LIST
inv(FLOW-LIST)	$\rightarrow$ FLOW-LIST
FLOW-LIST "−" FLOW-LIST	$\rightarrow$ FLOW-LIST {**left**}
FLOW-LIST "+" FLOW-LIST	$\rightarrow$ FLOW-LIST {**left**}

hiddens
context-free syntax
sorted-insert(CASH-FLOW, FLOW-LIST) $\rightarrow$ FLOW-LIST
exports
variables

C [*0-9*]*	$\rightarrow$ CASH-FLOW
C [*0-9*]*"*"	$\rightarrow$ {CASH-FLOW ","}*
F [*0-9*]*	$\rightarrow$ FLOW-LIST

equations
Querying and updating a single cash flow

$$\langle amount\text{: } N,\ date\text{: } D\rangle \;.\; date = D \tag{a1}$$

$$\langle amount\text{: } N,\ date\text{: } D\rangle \;.\; amount = N \tag{a2}$$

$$\langle amount\text{: } N_1,\ date\text{: } D\rangle \;.\; amount := N_2 = \langle amount\text{: } N_2,\ date\text{: } D\rangle \tag{a3}$$

Inserting elements in a sorted list

$$sorted\text{-}insert(C, []) = [C] \tag{i1}$$

$$\frac{C_1 \;.\; date \leq C_2 \;.\; date = true}{sorted\text{-}insert(C_1, [C_2, C^*]) = [C_1, C_2, C^*]} \tag{i2}$$

$$\frac{C_1 \;.\; date \leq C_2 \;.\; date = false}{sorted\text{-}insert(C_1, [C_2, C^*]) = sorted\text{-}insert(C_2, sorted\text{-}insert(C_1, [C^*]))} \tag{i3}$$

Insertion of non-empty elements.

$$ins(C, F) = F \qquad \Leftarrow C\,.\,amount = 0 \tag{i4}$$

$$ins(C, F) = sorted\text{-}insert(C, F) \Leftarrow C\,.\,amount \neq 0 \tag{i5}$$

Merging two flow lists.

$$merge(F, []) = F \tag{m0}$$

$$merge(F, [C, C^*]) = ins(C, merge(F, [C^*])) \tag{m1}$$

Making dates unique, netting the amounts.

$$nett([]) = [] \tag{n1}$$

$$nett([C]) = [C] \tag{n2}$$

$$\frac{C_1\,.\,date = C_2\,.\,date}{nett([C_1, C_2, C^*]) = nett(ins(C_1\,.\,amount := C_1\,.\,amount + C_2\,.\,amount, [C^*]))} \tag{n3}$$

$$\frac{C_1\,.\,date \neq C_2\,.\,date}{nett([C_1, C_2, C^*]) = ins(C_1, nett([C_2, C^*]))} \tag{n4}$$

"Arithmetic" operations

$$inv([]) = [] \tag{n5}$$

$$inv([C, C^*]) = ins(C\,.\,amount := -\,C\,.\,amount, inv([C^*])) \tag{n6}$$

$$F_1 + F_2 = nett(merge(F_1, F_2)) \tag{n7}$$

$$F_1 - F_2 = F_1 + inv(F_2) \tag{n8}$$

Taking first and second parts of a flow list

$$split\text{-}before([], D) = [] \tag{b1}$$

$$split\text{-}before([C^*, C], D) = [C^*, C] \qquad \Leftarrow C\,.\,date < D = true \tag{b2}$$

$$split\text{-}before([C^*, C], D) = split\text{-}before([C^*], D) \Leftarrow C\,.\,date \geq D = true \tag{b3}$$

$$split\text{-}after(F, D) = F - split\text{-}before(F, D) \tag{b4}$$

Module 3.1: Flows. An Algebra of Cash Flows

```
bal( [ < amount: 100 , date: 5  >,
       < amount:  50 , date: 10 >,
       < amount: 200 , date: 30 >,
       < amount: -300, date: 50 >   ],
     <start: 3, end: 40>,
     300 )
 =
[ < amount: 300, interval: <start: 3,  end: 5> >,
  < amount: 400, interval: <start: 5,  end: 10> >,
  < amount: 450, interval: <start: 10, end: 30> >,
  < amount: 650, interval: <start: 30, end: 40> > ]
```

Figure 3.2: Example of a balance computation

3.3.3 RISLA Syntax

Another major component of a language definition is its context-free syntax. As no grammar was given in the folder introducing RISLA [AG92], the syntax of RISLA had to be inferred from the (sometimes inconsistent) examples. Together with the author of a major part of the example definitions, we devised an initial syntax of RISLA, similar to the one in Figure 3.3. We tested whether a simple product description fitted well in this syntax, updated the syntax if necessary, and tried the next product. After several iterations, we had defined ten products ranging from the simple Deposit to the Fixed Rate Agreement, and a stable syntax. The interactive setting of the ASF+SDF Meta-environment proved ideal for these purposes.

The basic structure of a RISLA product description is illustrated by the deposit example of Figure 3.1. It starts with the declaration of the *contract parameters*. Next, the exported cash flows are listed (the *real* cash flows). Finally, after the *methods* keyword, the functions are actually defined. For these definitions the expressions are of importance; they consist of elementary numeric operations as well as tuple and list constructors. A function "for each ... new element is ..." may be used to compute a new list given an existing list.

In order to detect as many errors as early as possible, RISLA was made a statically typed language, containing explicit type declarations for, e.g., functions and contract parameters.

3.3.4 Issues Left Open

The two months of the project were spent specifying the underlying data types, developing a syntax for RISLA, and describing ten products in conforming to that syntax. Although we considered them informally, no full ASF+SDF specifications were given of the static semantics (type checking), evaluation, or the translation to COBOL calls.

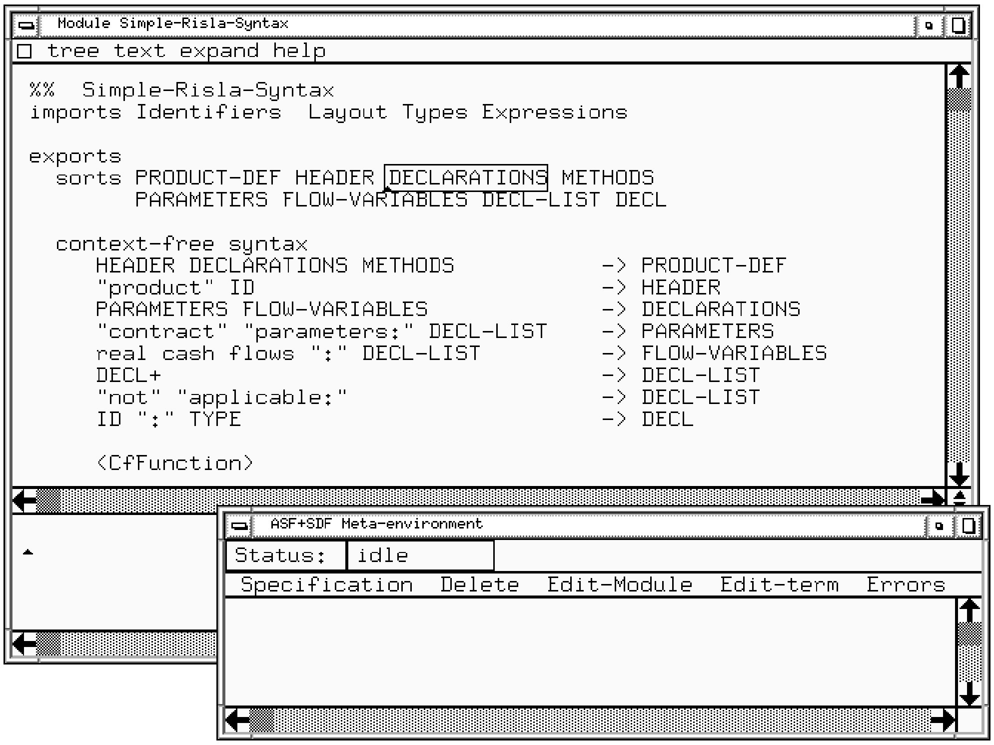


Figure 3.3: Snapshot of an editing session defining the syntax of RISLA

3.4 RISLA at the Bank

The bank's reaction to the two-month formalization effort was positive. The increased understanding of the data types of RISLA, as well as the fixed context-free syntax strengthened the management's faith in the RISLA approach. Work continued by defining the remaining (30) interest products of Bank MeesPierson. This turned out to be easily feasible, triggering some very minor further extensions of the RISLA syntax. At that moment it was decided to base all processing of Bank MeesPierson's interest rate products on RISLA descriptions.

The bank could have decided to use ASF+SDF to construct prototypes of the RISLA to COBOL compiler or a RISLA type checker. There were, however, several reasons not to do this. First of all, one should realize that the idea of using the specific RISLA language already was a small revolution within the bank. Introducing ASF+SDF in addition to RISLA would involve too many new things at the same time, and therefore endanger the acceptance of the RISLA project. Secondly, so far no experience existed with the use of ASF+SDF or algebraic specifications to generate COBOL programs. Lastly, the design phase of RISLA was considered to be completed, so the phase of building prototype tools could safely be skipped.

For these reasons, the implementation of a RISLA to COBOL compiler was started immediately, using Lex, Yacc, and C as the implementation platform.

3.5 Further Formalization

Apparently, it was not obvious to Bank MeesPierson and CAP Volmac that it could be beneficial to use algebraic specifications for the development of the RISLA to COBOL compiler. In order to show the feasibility of using ASF+SDF for these purposes, Res [Res94] provides a specification of a mapping between RISLA and COBOL. He gives a signature for COBOL, and defines functions translating RISLA constructs to COBOL routine calls. He needs about 35 pages of ASF+SDF code for this. In order to turn the specification into a compiler producing executable COBOL code, he also provides a COBOL pretty printer, produced using the pretty print generator described by Van den Brand [Bra93].

In addition to specifying this translation, Res presents a RISLA type checker. Although RISLA is a typed language, the compiler developed by the bank was only moderately concerned with type checking issues. Res discusses an algebraic specification of a static analysis function, capable of detecting undeclared contract data or auxiliary methods, function calls with incorrect argument types, illegal tuple constructions, and so on.

Specifying the type checker (25 pages) and the translation (35 pages) is a tedious task, involving several complicated details. Nevertheless, a specification has several well-known advantages over a C implementation (which has to deal with the same details), which are: (i) better readability, (ii) better maintainability, and (iii) easier reasoning about the compiler's correctness. Once all concepts are clear and all details understood, an implementation in a language like C can be undertaken.

The specifications described by Res show that the initial concerns about the feasibility of a complete formal specification of the RISLA compiler were not justified.

3.6 Concluding Remarks

In the initial phase of the RISLA project, algebraic specifications and ASF+SDF played a very important role. Algebraic specifications were used to describe the fundamental data types of the language, and the ASF+SDF Meta-environment was used to design a context-free grammar for RISLA interactively. It is fair to say that the continuation of the RISLA project would have been doubtful if RISLA had not been subject to a rigorous analysis using ASF+SDF.

ASF+SDF was not used to build prototype tools for RISLA. The main reason for this was that there was little experience with the use of ASF+SDF to generate COBOL, needed to express the interfacing to the bank's running systems. A recently developed specification [Res94] bridges this gap, illustrating the advantages and disadvantages of specifying the translation to COBOL formally.

Chapter 4

The Static Semantics of Pascal

One of the intended purposes of the ASF+SDF formalism is the easy specification of realistic programming languages. In this case study, we describe and judge an ASF+SDF specification of the static semantics of Pascal. With the exception of the goto-statement, the complete ISO standard static semantics is defined. It is discussed how the syntactic freedom of ASF+SDF can be used to create an easy-to-read description of the static semantics of Pascal, which stays as close as possible to the ISO definition.

4.1 Introduction

Although ASF+SDF was intended for the specification of realistic languages, no definition of any large language had been attempted at the time of writing this chapter. This raised the question whether the ASF+SDF formalism scales up smoothly. Do the techniques exploited in [BHK89] to define the toy-language Pico apply to more substantial languages as well? If not, what adaptations are necessary?

In this chapter we attempt to answer these questions by discussing a case study concerning type checking of Pascal, a well-known language of manageable complexity, standardized by ISO [ISO83]. We present some new techniques which ease the specification of larger languages. Moreover, we show how ASF+SDF can be used to obtain a formal yet readable definition, which remains close to the formulation used in the official ISO document (which is written in English).

4.2 Static Semantics in ASF+SDF

A definition of the static semantics of a programming language aims at narrowing the set of all programs generated by a context-free syntax down to those programs that are free of obvious errors, in particular errors that can be detected without executing the program. Typically, these errors involve context-sensitive requirements, e.g., all identifiers used must be declared, procedures must be called with the proper number of arguments, and so on.

One way to characterize the static semantics in an algebraic specification setting is to define a function taking an abstract syntax tree as input and producing a boolean result indicating correctness ("true") or the presence of errors ("false"). Alternatively, the specifier may feel the urge to yield more information than just "good" or "wrong", and define a function mapping to a domain of error messages, indicating *what* is wrong as well. In that case, a program is statically correct if and only if it is mapped to an "ok" element (or to the empty list of error messages). Note that in ASF+SDF error messages simply are functions defined in a signature, which, thanks to the user-definable syntax, can read like English sentences. For example:

```
exports
  sorts ERROR
  context-free syntax
    "Parameter lists are not congruous."      -> ERROR {constructor}
    "Identifier" ID "not declared."           -> ERROR {constructor}
```

Here the first function of sort ERROR is a simple constant; the second takes an argument indicating the name of the undeclared identifier.

Such a mapping has the flavor of a translation between the programming language to the language of errors. It is typically defined inductively over the abstract syntax, i.e., using *syntax-directed translation*. In order to define this mapping, elementary data types representing types, symbol tables, procedure declarations are needed. Algebraic specifications of these data types can be easily defined, the typical operations being functions with boolean results to decide type compatibility, type equivalence, assignment compatibility, or inspection and update functions on the symbol table.

Thus, a specification of the static semantics consists of four ingredients: the abstract syntax of the language, a signature for the error messages, abstract data types for the necessary type operations, and functions mapping syntax to errors using type operations.

Naturally, as we are aiming at *specifying* static semantics, the emphasis must be on readability. The syntactic freedom of ASF+SDF can be exploited to choose meaningful and easy-to-read function names in the specification.

4.3 The Pascal Specification

In this section, we discuss some of the more interesting aspects of the ASF+SDF specification of Pascal. We have selected three smaller subjects. For the full specification[1], which takes about 60 pages, we refer the reader to [Deu91].

4.3.1 Type Compatibility

It is easy to define a data type "context" which simply is a list of all identifiers declared at a certain point in a Pascal program. Basic lookup functions can be introduced to check the existence of identifiers. In order to add the necessary block structure to these contexts, a special element to mark the start of a new block can be used. The entries in the context should indicate the kind of identifier as well (is it a function, constant, type,

[1] The sources of the most up to date version of the specification, documented using the literate specification facility of ASF+SDF, are available by anonymous *ftp* from ftp.cwi.nl, pub/gipe/spec/pascal.tar.Z

procedure, ...), which can be easily achieved by putting the full definition as literally occurring in the Pascal program into the context. For example, a type declaration "**type** *positive* = 1 .. maxint" produces entry `type positive = 1 .. maxint` in the context. The full context starts with the declarations for the so-called required identifiers such as functions `sqrt`, `sqr`, `sin`, `round`, etc. Some required identifiers cannot be defined within Pascal itself, such as types `integer`, and I/O procedures like `readln` taking optional arguments or an arbitrary number of arguments. For these identifiers special context entries are created.

Type equivalence in Pascal is based on the place in the program where the types are declared. Stated in ISO terminology: "Each occurrence of a *new-type* shall denote a type that is distinct from any other *new-type*" [ISO83, p.12]. A way to model this is by using the full context of a particular type to represent that type. This guarantees that two different type denoters like, say "array [1..10] of char" are indeed regarded as different. Again, special entries in the context need to be created for types not directly corresponding to types expressible within Pascal, such as polymorphic types for the empty list and the nil pointer.

With a representation for types available, an abstract layer can be built containing functions to extract component or index types of arrays, to retrieve field types of records, to count the number of elements of ordinal types, to decide type equivalence or compatibility, and so on.

As an example of this, consider Figure 4.1. The signature introduces postfix operations ".nr-of-elements" to yield the number of elements of a Type, and " .index-type" to provide the index type of an array. The equations defining these functions are not shown in the figure. Moreover, the figure introduces type "classification" functions with boolean results, indicating whether a type is a subrange, a string, and so on. Again, the defining equations for these functions are not shown in the figure.

With these functions at hand, it becomes straightforward to translate, e.g., the ISO definition [ISO83, p.19] of compatibility to conditional equations. To that end, a mixfix function "_ and _ are compatible?" to define compatibility is introduced in Figure 4.1 The first equation states that subrange types are compatible provided they range over the same host type. For example: subrange 1..3 is compatible with 2..5, but not with 'a'..'c'. The second equation defines that strings are compatible provided they contain equally many characters. E.g., the type of 'aaa' is compatible with array [6..8] of char.

ISO formulates compatibility as follows: "Types T1 and T2 shall be designated *compatible* if ... T1 and T2 are subranges of the same host type ... T1 and T2 are string-types with the same number of components" [ISO83, p.20], which is surprisingly similar to the two conditional equations of Figure 4.1.

4.3.2 Types for Operators

Defining the types of expressions is a crucial part of the specification of the static semantics of any language. Particularly important are checks involved with the built-in operators such as +, ∗, <, and so on. For many smaller languages, the static (or dynamic) semantics of these built-in operators is defined by giving an explicit equation

```
module TypeOperations
imports SyntaxTypes Booleans Naturals
exports
  context-free syntax
    TYPE and TYPE are compatible "?"  -> BOOL
    TYPE is the same as TYPE "?"      -> BOOL
    "is-subrange-type?"( TYPE )       -> BOOL
    "is-string-type?"( TYPE )         -> BOOL
    "is-ordinal-type?"( TYPE )        -> BOOL
    TYPE ".host-type"                 -> TYPE
    TYPE ".index-type"                -> TYPE
    TYPE ".nr-of-elements"            -> NAT

variables
  T[12]   -> TYPE

equations
  [1]  is-subrange-type?(T1) = true,
       is-subrange-type?(T2) = true
       T1.host-type is the same as T2.host-type? = true
       ================================================
       T1 and T2 are compatible? = true

  [2]  is-string-type?(T1) = true,
       is-string-type?(T2) = true,
       T1.index-type.nr-of-elements = T2.index-type.nr-of-elements
       ===========================================================
       T1 and T2 are compatible? = true

       (* And several other cases for compatibility *)
```

Figure 4.1: Signature of some operations on types, and equations for two of the cases in which types are compatible.

for each of these operators. In Pascal, however, many operators exist and most of these can be overloaded. Giving separate equations for each would lead to many (more than 40) similar-looking conditional equations, each dealing with a particular built-in operator invoked with arguments of particular types.

ISO addresses this problem by giving tables summarizing the various operand types allowed for each operator [ISO83, p.39]. The question whether "real" and "integer" are allowed as operands for the "+" operator can simply be answered by looking for the "+" in the table, and inspecting the allowed operand types.

The same solution is beneficial for the ASF+SDF specification as well, as it greatly improves its readability and compactness. Figure 4.2 shows how the syntactic capabilities of ASF+SDF can be used to make this table as similar as possible to the ISO tables. The signature introduces syntax to write down tables, and the equations give two particular tables to be used when checking arithmetic and relational operators. Checking an arbitrary operator in an expression involves a simple look up in these

```
module TC-Expressions
imports SyntaxExpressions
exports
  sorts TABLE HEADER ENTRY OP-TYPE
  context-free syntax
    "Operator | Operand1  | Operand2  | Result  "
    "---------+-----------+-----------+---------"         -> HEADER

    OPERATOR "|" OP-TYPE "|" OP-TYPE "|" OP-TYPE          -> ENTRY

    HEADER ENTRY*                                         -> TABLE

    "correct-types?"( TABLE, OPERATOR, OP-TYPE, OP-TYPE ) -> BOOL
    "get-result-type"( TABLE, OPERATOR, OP-TYPE, OP-TYPE) -> OP-TYPE

    arithm-op-table                                       -> TABLE
    relational-op-table                                   -> TABLE

equations
  [1]  arithm-op-table =
         Operator | Operand1  | Operand2  | Result
         ---------+-----------+-----------+---------
            +     | int       | int       | int
            +     | int       | real      | real
            +     | real      | int       | real
            +     | real      | real      | real
            +     | set       | set       | set
           ...
         (*  similar for - * / div mod or not and  *)

  [2]  relational-op-table =
         Operator | Operand1  | Operand2  | Result
         ---------+-----------+-----------+---------
            in    | ordinal   | set       | boolean
            <     | string    | string    | boolean
            <     | simple    | simple    | boolean
           ...
         (*  similar for <= > >= = <>  *)
```

Figure 4.2: Exploiting syntactic freedom to create operator tables

tables.

4.3.3 Checking Type Constraints

The core of a static semantic definition states which constraints are to be fulfilled by each language construct. For instance, when introducing the assignment statement, ISO remarks that "the value of the expression of the assignment-statement shall be assignment-compatible with the type possessed by the variable denoted by the variable-access" [ISO83, p. 41].

In the ASF+SDF specification, this task of checking the type constraints for each language construct is performed by a collection of "tc-functions". Figure 4.4 introduces three tc-functions, operating on an expression, statement, and variable-access (the left-hand side of an assignment statement) respectively. The collection of tc-functions is defined inductively (compositionally) over the abstract syntax of Pascal.

A naive approach when defining these tc-functions is to give at least two (conditional) equations for each language construct: one to state that in the correct case no error needs to be produced, and the other to make sure that the proper message is yielded when the construct contains a type conflict. This, however, is undesirable as it doubles the size of the specification. Moreover, many constraints for different constructs are in fact the same. For instance, the if-then-else, while, and repeat-until statements all should satisfy the constraint that the type of their control expression should be boolean.

Therefore, these constraint checking functions are made explicit. We have given all these functions names with the words "should be" occurring in them, in order to suggest that a particular condition should be holding. As an example, the signature of Figure 4.3 introduces three should-be functions, "TYPE should be ordinal in STATE", "TYPE should be boolean in STATE", and "TYPE should be assignment-compatible with TYPE in STATE". Note that all these functions have names consisting of several words, two of which are "should" and "be". All should-be functions take a so-called "state" (an argument of sort STATE), and yield a new state. In order to compute this new state, they first check a particular constraint. If the constraint is met, the state is returned unchanged. Otherwise, the state extended with an informative error message is yielded.

As an example, consider Figure 4.3. We focus on the function "TYPE should be ordinal in STATE" for which two equations are given, which should ensure that the type it is given (its first argument) is an *ordinal* type, e.g., a subrange or an enumeration type. The function which knows whether a particular type is indeed ordinal is called "is-ordinal-type?(STATE)", and we assume it to be defined in module TypeOperations (Figure 4.1). The two conditional equations for the should-be function either yield the unchanged state S (first equation), or the state S extended with the error message "Ordinal type expected." (second equation).

All tc-functions have, besides their syntactic construct, a state as argument, and compute a new state. The state consists of a list of error messages (updated by should-be functions), a context containing the definitions of all identifiers introduced so far, and a *result* field. Whenever an expression is checked, the result field of the state

```
module ShouldBeFunctions
imports TypeOperations
exports
  context-free syntax
    TYPE should be ordinal in STATE                        -> STATE
    TYPE should be boolean in STATE                        -> STATE
    TYPE should be assignment-compatible with TYPE in STATE  -> STATE

    add-error "[" ERROR "]" to STATE                       -> STATE
    "Ordinal type expected."                               -> ERROR {constructor}

variables
  S     -> STATE
  T[12] -> TYPE

equations
  [1]  is-ordinal-type?( T1 ) = true
       =============================
       T1 should be ordinal in S = S

  [2]  is-ordinal-type?( T1 ) != true
       ==============================
       T1 should be ordinal in S =
         add-error [Ordinal type expected.] to S
```

Figure 4.3: The signature introduces three functions whose name contain the words “should be”. For one of these, “should be ordinal in”, the defining equations are given.

contains the type of that expression.

To illustrate the use of states and should-be functions, consider Figure 4.4. The two equations characterize the static semantics of two sorts of statements, the if-then-else and the assignment. In order to check an if-then-else, the sub-constructs (the expression and the then-part) are checked. The constraint to be met by the entire if-then-else is that the type of this particular expression (denoted by S2.result) should be boolean. Likewise, the equation dealing with an assignment checks the left-hand side and the right-hand side, and states that their types (S2.result and S3.result) should be assignment-compatible.

Figure 4.4 should make the advantages of this approach immediately obvious: One equation is given for each language construct, and the style of ISO stating constraints for each language construct can be followed directly.

4.4 Related Work

The Pascal language has been subject to various earlier formalization attempts. Some of these exclusively deal with dynamic semantics, e.g., the axiomatic definition of

```
module TC-Stats
imports ShouldBeFunctions TC-Expressions
exports
  context-free syntax
    expr-tc "[[" EXPR "]]" in STATE            -> STATE
    stat-tc "[[" STATEMENT "]]" in STATE       -> STATE
    var-at-lhs-tc "[[" VAR-ACCESS "]]" in STATE -> STATE

    STATE ".result"                            -> TYPE

variables
  Stmt      -> STATEMENT
  Exp       -> EXPR
  VarAccess  -> VAR-ACCESS
  S[1-4]    -> STATE

equations
 [1]  expr-tc [[ Expr ]] in S1 = S2,
      stat-tc [[ Stmt ]] in S2 = S3,
      S2.result should be boolean in S3 = S4
      ======================================
      stat-tc [[ if Expr then Stmt ]] in S1 = S4

 [2] var-at-lhs-tc [[ VarAccess ]] in S1 = S2,
     expr-tc [[ Exp ]] in S2 = S3,
     S3.result should be assignment-compatible with S2.result in S3 = S4
     ===================================================================
     stat-tc [[ VarAccess := Exp ]] in S1 = S4
```

Figure 4.4: Type checking statements using the "should be boolean in" and "should be assignment-compatible with" functions.

Hoare and Wirth [HW73], or the more recent action semantics of Mosses and Watt [MW93]. A denotational semantics also covering static semantics is given by Tennent [Ten77], and by Andrews and Henhapl in VDM [BJ82, Chapter 7]. A definition using *predicate rules*, akin to Z, is presented by Duke [Duk87].

More often, the static semantics are covered by attribute grammars (AGs). For instance, definitions have been given in the Synthesizer Specification Language (SSL) [RT89a], in the ALADIN formalism of the GAG system [KHZ82], and using extended attribute grammars [Wat79]. None of these attribute grammar formalisms give the specifier the syntactic freedom for function names offered by ASF+SDF. As a result, the ASF+SDF specification is superior in readability compared with the attribute grammar definitions. Concerning the size of the specifications, the description of Watt is remarkably compact. The definitions in SSL and ALADIN are of the same size as the ASF+SDF specification (about 60 pages). Concerning expressive power, Courcelle and Franchi-Zannettacci have proved a one-to-one correspondence between a subclass of algebraic specifications, the so-called *primitive recursive schemes* (see also Chapter 7) and a subclass of attribute grammars, the strongly non-circular attribute grammars

[CF82].

Visser describes an algebraic formalization (in ASF+SDF) of the static semantics of the object-oriented language Eiffel [Mey88, Vis92], a language that is more difficult to type check than Pascal. He uses states in a way similar to ours. He extends them with a *context manipulation language* (CML), supporting sequential composition, if-then-else, and case statements. His approach has the flavor of translating Eiffel to an intermediate language (the context-manipulation commands) which can be evaluated to yield a list of error messages.

In order to avoid the need to pass states around explicitly, (see, e.g., the occurrences of variables S1 to S4 in the equations of Figure 4.4), state-hiding *combinators* could be used. In the approach discussed in this chapter, variables are needed because several components of one state may be needed by different functions. For example, equation [1] in Figure 4.4 assigns a value to S1 in the first condition, and then uses that value in both the second and the third condition. If we introduce a number of combinators that allow one both to sequence various state manipulation as well as to access certain components of states computed in between, we can avoid this unelegant use of variables. For *dynamic semantics*, this combinator approach has been studied extensively in the area of *action semantics* (see also the next chapter).

The specification style of the static semantic definition we discussed can be considered as rather classical: A function is defined inductively, mapping abstract syntax to a list of errors. Recently, Dinesh has conducted experiments with the use of abstract interpretation to characterize the static semantics of programming languages [Din93]. In short, he translates a language to an abstract domain of types, and then evaluates over the new domain using equations like "integer + integer = integer". He has given a specification for Clax, a relatively large subset of Pascal.

4.5 Concluding Remarks

We have presented a specification of the static semantics of Pascal, emphasizing readability and staying close to the ISO definition. We discussed a general layout of a static semantic definition, and presented some techniques which ease the specification of large and realistic languages. The syntactic freedom of the ASF+SDF proved very useful for these purposes.

In addition to supporting the development of high-level specifications of languages, however, the ASF+SDF Meta-environment aims at prototyping tools defined by such specifications. In other words, the specification of the static semantics should produce a reasonably efficient and more or less usable type checker. How feasible is this?

The ASF+SDF Meta-environment can directly execute the equations of an ASF+SDF specification using conditional term rewriting. Doing so indeed yields an executable type checker, which given a Pascal program computes a list of messages. This type checker has been used[2] to check some smaller programs (like quick-sort, not more than five pages). Conducting these experiments revealed three issues:

[2] In 1991, on an 8 Mbyte Sparc 1, execution was not yet possible. By now, we are running the specification successfully on a 32 Mbyte Silicon Graphics machine.

- Executing the specification helped to find several small errors.
- The incrementally generated rewrite machine of the Meta-environment is useful while testing small functions or sub-modules. To obtain a reasonably efficient type checker the the compiler translating ASF+SDF to C [KW93] must be used.
- The output obtained by just executing the equations as rewrite rules is unsatisfactory. Type checking a Pascal program of five pages simply computes a normal form, e.g., "Types are not compatible." What is missing is any information about how this error is related to the original program. Plain rewriting does not reveal any information where the type incompatibility occurred in the five pages.

This last issue seems to be the most interesting of the three. It is intriguing that while reading the specification, one intuitively has a rather clear picture which statements can cause particular errors. In addition to that, when executing the rewrite rules "by hand" for a specific program, one will be rather certain which parts of the source program were responsible for errors. This raises the question whether this information can be extracted *automatically* as well, while executing the rewrite rules.

Questions like these will be studied in full detail in the second part of this thesis. In fact, the issue will be generalized considerably: Part II deals with techniques to establish dependence relations between initial values and resulting data occurring in computations described by equations.

Chapter 5

Tools for Action Semantics

Action Semantics is a recently-developed paradigm for the formal definition of programming languages. ASF+SDF has been used to implement a set of tools supporting the development of action-semantic descriptions. The tools feature editing, parsing, checking, and execution (based on term rewriting) of action-semantic descriptions. Moreover, they include the full specifications of standard Data and Action Notation modules. Non-trivial aspects of the tools deal with mixfix syntax, prevention of syntactic ambiguities, well-sortedness checks, and parser generation. The suitability of ASF+SDF for implementing this tool box is evaluated.

5.1 Introduction

In this chapter we show how ASF+SDF has been used to provide tool support for *action semantics* [Wat91, Mos92]. Before introducing these tools, we briefly explain action semantics itself. An overview of some special terminology is given in Figure 5.1.

The most widespread approach to programming language semantics to date is denotational semantics. A language is given its meaning by mapping programs to mathematical objects. Typically, such a mathematical object is a λ-calculus expression, in which the complete program behavior is encoded. However, it is widely acknowledged that denotational language descriptions suffer from the following three problems:

- The encoding can be notoriously difficult: several clever tricks are needed to represent, e.g., stores, goto's, or parallelism;
- Denotational definitions easily become unreadable;
- Denotational semantics does not scale up to realistic (large) languages.

In *action semantics* [Wat91, Mos92] an attempt is made to overcome these difficulties. Instead of encoding directly in the λ-calculus, the meaning of programs is expressed using so-called *actions*. The problems mentioned are coped with by:

- Making fundamental concepts such as stores, cells and values explicit, rather than encoding them in λ-terms;

- Choosing well-considered names which ensure that actions read like English. For instance, the denotation of an assignment `x := 3` is the action `give decimal "3" then store the given value in the cell bound to "x"`.
- Providing modularity and data hiding (in combinators) to support the construction of large language definitions.

Action semantic descriptions have been given for several (realistic) languages, including Pascal [MW93] and Standard ML [Wat87].

5.1.1 Actions

In Action Semantics, the meaning of a language is defined by mapping that language to actions. Actions are written in the *Action Notation* language (AN). We do not need to know very much about Action Notation for the purposes of this chapter.

To give a small impression of Action Notation, however, consider the ASF+SDF module Actions-in-ASF-plus-SDF (Module 5.1), defining a subset of AN. A sort ACTION is introduced, along with a subsort for primitive actions. Actions are written using constants like "complete" or "rebind". They can be combined using combinators like "and then" or "moreover". Using such combinators, sentences like "bind x to 3 and then complete" can be built.

These actions are used to give meaning to programming languages. The meaning of *actions*, in turn, is defined by an operational semantics of Action Notation, given in [Mos92, Appendix C]. This operational semantics defines how every action can be *performed*. For instance, the performance of "bind x to 3" will have a certain effect on the a list of *bindings*, changing the value associated with the token "x" to the new value "3". The details of action performance are fully formalized by the operational semantics. They are, however, of little importance for the understanding of the rest of this chapter.

5.1.2 Action-Semantic Descriptions

An action-semantic description (a.s.d.) of a language is divided into three parts. First of all, the *abstract syntax* of the language is defined. Secondly, the so-called *semantic entities* are specified, which are used to specialize the general Action Notation to the specific needs of the language defined. For instance, when describing the action semantics of Pascal, a definition of Pascal's built-in notion of a file type should be given. Finally, the *semantic functions* are defined, which map programs (constructed using the abstract syntax) to actions (built from AN extended with the semantic entities).

5.1.3 A Formalism for Writing a.s.d.s

In order to write down a.s.d.s, Mosses has developed a formalism referred to as the "Meta Notation" [Mos92, Appendix F]. It is this Meta Notation that we are building tools for in this chapter.

module Actions-in-ASF-plus-SDF

imports Layout
exports
  sorts ACTION PRIM-ACTION
  context-free syntax

PRIM-ACTION	$\rightarrow$ ACTION	
complete	$\rightarrow$ PRIM-ACTION	
unfolding ACTION	$\rightarrow$ ACTION	
ACTION *or* ACTION	$\rightarrow$ ACTION	{**left**}
ACTION *and then* ACTION	$\rightarrow$ ACTION	{**left**}

imports Yielders-in-ASF-plus-SDF
exports
  context-free syntax

bind YIELDER *to* YIELDER	$\rightarrow$ PRIM-ACTION
rebind	$\rightarrow$ PRIM-ACTION
furthermore ACTION	$\rightarrow$ ACTION
ACTION *moreover* ACTION	$\rightarrow$ ACTION

exports
  priorities

{ unfolding, furthermore, bindto } > { or, andthen, moreover }

hiddens
  variables
    $A\rightarrow$ ACTION

equations

$$\textit{furthermore}\ A = \textit{rebind moreover}\ A \qquad [1]$$

Module 5.1: Actions-in-ASF-plus-SDF. Action Notation defined using ASF+SDF

The Meta Notation (MN) is very similar to an algebraic specification formalism. Arbitrary data types can be defined, as needed, e.g., for the specification of the *semantic entities*. Moreover, special facilities are offered to define *abstract syntax*. Conditional equations (positive horn clauses) can be used to state equalities and define functions, for instance the *semantic functions*. Modularization constructs ease the construction of large specifications.

MN has been used to provide a specification of the "action" abstract data type, in the same manner as the ASF+SDF formalism was used in module Actions-in-ASF-plus-SDF (Module 5.1). In fact, this ASF+SDF module was derived from the MN module shown in Figure 5.2 (which we will study in more detail in Section 5.2.1). It defines a sort ACTION as well as certain equalities over terms over the sort ACTION. The full MN definition of AN is given in [Mos92, Appendix B]. In addition to AN, so-called "Data Notation" modules exist, a collection of MN modules defining basic data types such as integers, strings, and trees.

Summarizing, the very same MN formalism is used to define AN and DN, syntax, data types occurring in languages, and mappings from syntax to AN. The MN formalism is discussed in more detail in the next section, where we will also see that MN is not a conventional algebraic specification language, as it is based on the notion of *unified algebras*.

5.1.4 The ASD Tools

The tools described in this chapter aim at supporting the development of a.s.d.s. They were baptized the ASD tools, ASD being an abbreviation for "Action-Semantic Descriptions".

The tools support a version of the MN which is as close as possible to the formalism used in [Mos92]. The tools can be used to perform various checks modules written in this formalism. For example, they can be used to verify that the terms used in equations are written only using the symbols that were introduced. Moreover, the tools support, to a certain extent, the execution of MN modules, in the sense that certain equations are executed as rewrite rules.

In addition to that, the tools give the possibility to generate syntax-directed editors and parsers from abstract-syntax definitions written in MN. Thus programs can be parsed, and an abstract-syntax tree can be constructed. Together with the possibility to execute specifications using rewriting, this provides ASD users with the possibility to compute the action denotations of actual programs.

In addition to providing support for writing MN specifications, the ASD Tools include a library of MN modules. Naturally, the full AN and DN modules are provided. Moreover, a.s.d.s of several languages are given, such as the toy-language Pico (used for tutorial purposes), ISO Pascal [MW92], and the subset of Ada called Ad [Mos92, Appendix A].

Currently, the ASD Tools are *not* intended for semantic prototyping based on action-semantic descriptions of source languages. Such facilities are offered by already existing systems such as those discussed in Section 5.1.6. Rather, ASD can be used to check action semantic descriptions written in the style of [Mos92]. As an example, the existing definitions of Ad [Mos92] and Pascal [MW93] have been passed through ASD. The LaTeX sources of these definitions could be translated automatically to the ASCII representation supported by ASD. The checks that could subsequently be performed on these definitions revealed about one (albeit sometimes very minor) error per two pages (see Section 5.5.3 for a more detailed account).

5.1.5 Why ASF+SDF?

The striking similarities between ASF+SDF and MN were the main reason to start investigating how ASF+SDF could be used to build the ASD Tools. Like ASF+SDF, the MN is based on algebraic specifications. Like ASF+SDF, the MN supports full syntactic freedom for the choice of function names. Just as ASF+SDF, the MN can potentially be executed using term rewriting. Finally, both ASF+SDF and MN can be used to define languages, where these definitions can be used to generate tools.

These similarities suggest that it should be straightforward to build tools for MN using ASF+SDF. This, however, was not the case: some of more interesting problems are covered in considerable detail in this chapter. In particular, we have compiled a list of recommendations and suggestions for improvement, which is presented in Section 5.7.

5.1.6 Related Work

No tools supporting the Meta Notation exist at the moment. However, several compiler generators are available taking Action Notation as their point of departure. For example, the Cantor system has been used to obtain a compiler translating a subset of Ada to Sparc machine code. The compiler was generated from an action semantic description of that subset of Ada [Pal92a, Pal92b]. The Cantor system puts an emphasis on generating provably correct compilers. The OASIS system, written in a mixture of Perl, Scheme, C, C^{++}, Flex, and Bison, aims at generating efficient Sparc code [Ørb94]. Ørbaek reports on generated Sparc code that is only a factor 2 slower than the Sparc produced by hand-written compilers. The Actress system [BMW92] can derive compilers which translate to C; it can, moreover, deal with type checking issues and analysis of action-semantic descriptions. An extension of Actress which focuses on rigorous techniques for typing and binding-time analysis is described by Doh and Schmidt [DS93]. Ruei and Slonneger describe an ML-implementation for a considerable subset of Action Notation, mainly intended for teaching purposes [RS93]. Recently, Watt has experimented with the use of ASF+SDF to describe and execute the transformation and interpretation of Action Notation [Wat94].

5.2 MN: The Meta Notation

In this section, the MN formalism and the unorthodox framework of unified algebras are explained informally in terms of concepts occurring in many-sorted algebraic

Action Semantics	The semantics of a programming language L is defined as a mapping from L to Action Notation.
Unified Algebras (UAs)	A derivative of algebraic specifications.
Meta Notation (MN)	A (modular) formalism for writing UA specifications.
Data Notation (DN)	A collection of MN modules defining standard data types such as numbers, strings, lists, and so on.
Action Notation (AN)	A collection of MN modules introducing a sort `action` and several operations over the sort `action`.
An action	An action is a term of sort `action`.
Abstract Syntax for L	A collection of MN modules defining the syntax of L. The MN provides a special "grammar" construct for this.
Semantic Entities for L	A collection of MN modules extending (instantiating) Action Notation with some L-specific data types.
Semantic Functions for L	A collection of MN modules defining a compositional mapping from the abstract syntax to the semantic entities.
Action-semantic description (a.s.d.) for L	The three MN-modules for the Abstract Syntax, Semantic Entities and Semantic Functions of L.
Action performance	The process of interpreting an action. Formalized by a collection of MN modules describing the Structural Operational Semantics of Action Notation.

Figure 5.1: Action Semantics Terminology

```
module: Action Notation .

  module: Basic.
    introduces: action, primitive-action, complete,
                unfolding _,  _ or _,  _ and then _.

     (*)   action        >= primitive-action .
     (*)   complete      :  primitive-action .
     (*)   unfolding _   :: action ->  action (total)   .
     (*)   _ or _        :: action, action ->  action
                            (associative, commutative).
     (*)   _ and then _  :: action, action ->  action
                            (associative, unit is  complete) .
  endmodule: Basic.

  module: Declarative.
    includes: Basic, Yielders.
    introduces:  bind _ to _, rebind, furthermore _,  _ moreover _ .
     (*)   bind _ to _   :: yielder, yielder -> primitive-action .
     (*)   rebind        :  primitive-action.
     (*)   furthermore _ :: action -> action (total).
     (*)   _ moreover _  :: action, action -> action .

    (1:)  furthermore A:action = rebind moreover A.
  endmodule: Declarative.

endmodule: Action Notation.
```

Figure 5.2: Modules introducing a small subset of Action Notation

specifications. A detailed description of unified algebras and MN can be found in [Mos89, Mos92].

5.2.1 Abstract Data Types

As a first example of a unified algebraic specification, consider the module in Figure 5.2. At first sight, we can read the module as an ordinary many-sorted algebraic specification (in this case as Actions-in-ASF-plus-SDF (Module 5.1)). For instance, submodule `Basic` of larger module `Action Notation` simply contains a signature, introducing function symbols `unfolding _`, `_ or _`, and `_ and then _`, which are given functionalities indicating that they take arguments of sort `action` and return a value of sort `action`. Moreover, a constant `complete` of sort `primitive-action` is introduced. These two sorts, `action` and `primitive-action`, are introduced as well, the latter being declared as a subsort of the former by the relation `action >= primitive-action`.

The second submodule, `Declarative`, not only introduces symbols and gives them

```
module: Semantic Entities.

  introduces: value .

  includes: ActionNotation .

  (*) value    = integer | string | cell .
  (*) datum   >= truth-value | value .

  (*) token    = string .
  (*) bindable = cell .
  (*) storable = value .

endmodule: Semantic Entities.
```

Figure 5.3: Semantic Entities specializing action notation modules

functionalities. It also contains an equation expressing the relation between the `furthermore _` and `_ moreover _` function, stating that a `furthermore _` operating on an arbitrary term `A` of sort `action` is equal to a `_ moreover _` applied to the constant `rebind` and action `A`.

Quite often unified algebra specifications can be read as normal many-sorted specifications. The module we just studied, `Action Notation`, simply defines an abstract data type `action`. In Subsection 5.2.4 we will study some of the differences with normal algebraic specifications. We will also see how operations like `_ and then _` are interpreted as sort constructors.

5.2.2 Modularization

The MN encourages the development of *modular* specifications. The most distinctive modularization features of MN are *mutually recursive* module dependencies (cyclic imports) interpreted as module union, and *nested modules* as we have seen in Figure 5.2. Moreover, renaming of symbols is possible by specifying so-called *translations*. The Meta Notation furthermore distinguishes between *included* modules, the symbols of which are exported by the including module, and *needed* modules, the symbols of which are not visible in other modules importing the needing module (a hidden import). Deeply nested modules can be referred to by using a "directory-like" notation, e.g., **needs:** `Action Notation / Declarative`.

Parameterization of modules is achieved by exploiting subsorts, using a mechanism called *specialization*. For instance, in order to define generic lists, an empty sort `item` can be used to represent the elements of the lists. Whenever this sort is used for, say, lists of characters, the sort `item` is "specialized" to include the sort `character`, which is expressed by subsort relation `character =< item`. See [Mos92, p.47, E.6, E.7.1] for more details on specializations and instantiated lists.

When writing an action-semantic description of a programming language, this mechanism is used to specialize the general action notation to the specific needs of

```
module: Abstract Syntax.

  grammar:
  (*)   Statement  = [[ Identifier ":=" Expression ]] |
                     [[ "while" Expression "do" Statement ]] |
                     [[ "begin" Statements "end" ]] .
  (*)   Statements = < Statement < ";" Statement >* > .
  (*)   Expression = Numeral | Identifier | [[ "(" Expression ")" ]] |
                     [[ Expression Operator Expression ]] .
  (*)   Operator   = "+" | "-" | "*" | "<>" | "and" | "or" .
  (*)   Numeral    = [[ digit+ ]] .
  (*)   Identifier = [[ letter (letter | digit)* ]].
  endgrammar.

endmodule: Abstract Syntax.
```

Figure 5.4: Abstract syntax for a simple programming language

the language defined. Figure 5.3, e.g., specializes the sorts `datum`, `token`, `bindable`, and `storable`, using a new sort `value`. A `value` is defined to be built from `integers`, `strings`, and so-called `cells`. The sort `value` is included, along with the sort `truth-value`, into the sort `datum`. In general, instantiating action notation is part of the action-semantic module defining the semantic entities. For more complicated languages, the semantic entities module will have to define various new data types as well (not part of Action Notation or Data Notation).

5.2.3 Abstract Syntax

The Meta Notation provides special notation for defining signatures reflecting the abstract syntax of programming languages. As an example, module Abstract Syntax in Figure 5.4 gives the syntax of a small imperative language. Syntax is built from trees and characters, and defined using `*` and `+` for repetition (zero and one or more, resp.), `?` for optional constructs, and `|` to specify alternatives. Brackets `<...>` build flat associative lists of trees, the repetition of which yields flat lists again. For example, `< Statement < ";" Statement >* >` builds a flat list of statements separated by semicolons. Sub-sorting can be used to identify that, for instance, expressions are built from numerals and identifiers. The brackets `[[...]]` correspond to the interior nodes in the trees defined by the grammar.

Grammar clauses do not distinguish between lexical and context-free syntax. Both are defined using the same regular expressions. The sorts `digit` and `letter` (see the specification of `Identifier` in Figure 5.4) are defined in module `Data Notation / Characters` [Mos92, Appendix E.5]. This module, as well as the module `Data Notation / Syntax` [Mos92, Appendix E.8] are implicitly imported in each module containing a grammar clause. This clause, moreover, abbreviates the explicit introduc-

```
module: Natural Numbers .

  module: Basics.
    introduces: natural, 0, succ _ .
    (1:)  0 : natural .
    (2:)  succ _ :: natural -> natural (total, injective) .
    (3:)  natural = 0 | succ natural (disjoint) .
    closed.
  endmodule: Basics.

endmodule: Natural Numbers.
```

Figure 5.5: Natural numbers in unified algebras

tion of the "non-terminal" sorts at the left-hand sides of the grammar rules [Mos92, Appendix F.2.1].

In Subsection 5.2.4 we will return to abstract syntax definitions, and see how a grammar clause can be interpreted as a normal unified algebraic specification.

5.2.4 Unified Algebras

The key observation of unified algebras is that sorts are the weakest developed notion of many-sorted algebraic specifications. For instance, built-in sort operations like sort union or sort intersection are typically not supported, let alone user-defined operations on sorts. These issues are addressed by Unified Algebras, expressed by the following three statements:

- A sort represents a set of values. We will need standard functions `|` and `&` to denote set union and intersection, as well as the symbol `nothing` to denote the empty set. Different sorts can be classified according to the subset relation, and structured into a power set lattice with subset ordering and `nothing` as bottom. We often talk about *subsorts* rather than subsets.

- One term corresponds to a singleton sort containing just that term. The most natural candidate name to denote that sort is that term itself. Thus, "`3`" can be regarded as a sort containing one element. Elements will be referred to as *individuals*.

- Many operations on individuals naturally extend to operations on sorts by taking the union of the application of that operation to each of the individuals in the sort. For instance, we expect `succ(3)` to be equal to 4, and `succ(3|4|5)`, i.e., `succ` operating on the set $\{3, 4, 5\}$, to be equal to `4|5|6`.

In UAs, the introduction of a symbol does not mark it as either a sort symbol or an individual symbol, nor does it impose any restrictions on argument types or result types. The set of symbols, together with built-ins `nothing`, `_|_`, and `_&_`, induces a set of `terms`. The subsets of this set are ordered into a lattice, with `_|_` acting as join, `_&_` as meet, and `nothing` as bottom. Functions operating on such sets are required to be monotonic with respect to subsort inclusion. A specification will try to narrow this huge lattice down by grouping terms into equivalence classes, derived from equality relations between terms, functionality restrictions for certain symbols, explicit subsort relations and individual assertions. Certain commonly-occurring assertions may be abbreviated by adding *attributes* such as *total*, *associative*, or *injective* to functionality specifications.

A common example to illustrate the use of UA specifications is the datatype of the natural numbers, as defined in Figure 5.5. The introduced symbols `0` and `succ _` act as constructors. Together with `_ | _` they can be used to write down subsets of natural numbers. The given individual assertion `(1:)`, functionality `(2:)`, and (recursive) sort equation `(3:)` serve to make the constant `natural` equal to the term `0 | succ 0 | succ succ 0 | ...`. Moreover, the given attributes *total* and *injective* guarantee that `succ _` is one-one; as a result, `succ _` works element-wise on sorts, such that, e.g., `succ (3|4) = 5|6`.

The natural number example is slightly unsatisfactory in the sense that one is really interested in only one sort, namely just `natural`. A more convincing example, involving many different sorts and operations on sorts, might be the definition of abstract syntax trees. The standard module `Data Notation / Syntax` [Mos92, Appendix E.8] introduces a sort `syntax-tree`. This sort contains all possible syntax trees built from strings in the leaves and `[[...]]` operators as internal tree nodes. It has many subsorts, containing, e.g., just single strings such as `"*"`, or several strings like `"*"|"+"|"-"`, or entire trees such as `[[ "1" "+" [[ "2" "*" "3"]] ]]`. Grammar specifications in UA Meta Notation, as, e.g., occurring in Figure 5.4 are nothing but the identification of subsorts of the sort `syntax-tree`. This is done using simple sort equations, helped by sort operations such as `*` and `+` as defined in module `Data Notation / Tuples` [Mos92, Appendix E.2]. For example, the line `Operator = "*"|"+"|"-"|...` just gives an explicit name to the sort `"*"|"+"|"-"|...`. The equation `Numeral = [[ digit+ ]]` identifies the sort Numeral with all trees having one or more digits as its leaves.

This mechanism captures the intuition that every sort representing the abstract-syntax of a (programming) language should be subsort of one big sort representing all possible trees. This has the advantage that generic tree functions are possible, i.e., functions that can be applied to the abstract syntax tree of any language. Examples of such generic functions are addressing functions which take an arbitrary tree as well as a path in that tree and yield the subterm at the end of that path, functions taking a tree and producing a textual representation (pretty-print) of the tree, or functions counting the number of interior nodes in an abstract syntax tree. The abstract syntax tree mechanisms supported by, e.g., OBJ [GWM+92] or SDF [HHKR89] are not capable of dealing with such generic tree functions.

```
module: Semantic Functions.

  needs: Abstract Syntax, Semantic Entities.

  module: Executing Statements.

    needs: Evaluating Expressions.

    (*)    execute _ :: Statements -> action.

   [1:]    execute [[ I:Identifier ":=" E:Expression ]] =
             evaluate E then store the given value in the cell bound to I .

   [2:]    execute [[ "while" E:Expression "do" Ss:Statements "od" ]] =
             unfolding
             ( evaluate E then give not (it is 0) then
               ( ( check (it is true) and then execute Ss and then unfold )
                 or  check (it is false) ) ) .

  endmodule: Executing Statements.

endmodule: Semantic Functions.
```

Figure 5.6: Some semantic equations mapping a language to its actions

5.3 An Example Action-Semantic Description

Now that we have a sufficient understanding of the Meta Notation, we can show how it is used when defining the semantics of programming languages. We will briefly discuss a subset of the example given by Mosses [Mos92, Chapter 2].

Recall that an action-semantic description consists of a definition of the abstract syntax, the semantic entities, and the semantic functions.

An MN module defining the syntax for a small programming language was given in Figure 5.4. Likewise, an MN module defining the semantic entities needed for this language was shown in Figure 5.3.

The connection between these two is made by the module defining the semantic functions. Some of the semantic functions for this language are defined in the MN module shown in Figure 5.6. It introduces a function `execute _`, which maps values of sort `Statements` to terms of sort `action`. The function is defined by distinguishing the various `Statements` as defined in the grammar of module `Abstract Syntax` (Figure 5.4). In the example, the defining equations for the assignment and while-statement cases are given. The semantic functions are required to be compositional, i.e., they should express the meaning of a larger construct in terms of the semantics of its components. Also observe that the equations given in such modules defining semantic functions are typically easily executable using term rewriting.

5.4 An Overview of ASD

ASD is capable of performing the following tasks:

- Parsing and syntax-directed editing of Meta Notation modules. Typical errors to be caught are misspellings of keywords or forgotten punctuation.
- Checking whether the symbols used in equations are indeed introduced and used in accordance with the specified arity.
- Generating parsers from grammar sections. The parsers can analyze programs written in concrete notation (e.g., actual Pascal programs), and produce abstract syntax trees in the `syntax-tree` format of module `Data Notation / Syntax` (see [Mos92, Appendix E.8], as well as Section 5.2.4).
- Executing certain axioms as rewrite rules. In particular, semantic equations mapping abstract syntax trees to action notation can be executed to yield the action term corresponding to the source program.
- Verifying, as far as possible, that terms occurring in axioms are sorted in accordance with the functionality axioms. Note that well-sortedness in Unified Algebras is undecidable in general (sort assertions may be conditional on equations involving arbitrary terms).

Moreover, ASD comes with the libraries of Action Notation and Data Notation as specified in [Mos92, Appendix B,E], as well as the action-semantic descriptions of Ad [Mos92, Appendix A] and Standard Pascal [MW93].

5.4.1 Context-free Syntax for the Meta Notation

In order to get an ASCII representation of the Meta Notation used in [Mos92], the abstract syntax given in [Mos92, Appendix F.3] needs to be adapted to an SDF grammar. The full SDF definition is given in Appendix A. The most visible differences involve making the abstract syntax into a less ambiguous concrete syntax. As SDF cannot easily deal with the use of indentation or section numbering to indicate the level of nesting, the grammar is extended with several delimiters for disambiguation, such as "module:", "endmodule:", "grammar:", "endgrammar.", and conventional parentheses. The original definition recognizes variables as such since they were written in *this italic font*. Variables in the ASD Meta Notation are just alpha numeric words. They may be introduced either in-line (as in [Mos92], see, e.g., Figure 5.6), or using a new keyword **variables:**.

Tags like • and (N) with N a number, are used in [Mos92], although not specified as part of the MN abstract syntax. The SDF grammar for MN includes tags `(*)`, `(`N`:)`, `[`N`:]`. Specifiers can use the tags to indicate whether axioms are to be interpreted as dealing with syntax, or whether they can be executed as rewrite rules.

The more interesting problems when constructing the context-free grammar for the Meta Notation deal with symbols and terms. Symbols occur, e.g., after the keyword **introduces:**. All kinds of symbols are allowed, not only prefix operations, but arbitrary infix operations like unfolding _, bind _ to _ (Figure 5.2), as well as components from # _ to # _ _ (occurring in Data Notation / Tuples, [Mos92, Appendix E.2]). Can we find a context-free syntax for symbols, which is general enough to recognize all intended symbols as symbols, and restricted enough not to consider, e.g., a complete MN module as just one symbol? The solution chosen in ASD is to restrict symbols to *well-balanced* symbols only and to disallow certain characters to occur in symbols. Thus, brackets [...], (...), {...}, and <...> are only allowed to occur as pairs. Moreover, characters occurring in the MN itself, such as =, :, or . are not allowed to occur in symbols, and a character like comma (,) is only allowed to occur inside parentheses. Although these restrictions were not fully stated before, it turns out that all symbols used in, e.g., [Mos92, MW93] conform to this syntax.

Finding a good syntax for terms is even more challenging. Terms are similar to symbols, but the place holders need to be replaced by actual terms. Furthermore, terms may use *strings* and quoted *characters*. Moreover, they can introduce *variables*, for instance in a term succ(x:natural), which forces variable x to be an individual of sort natural. In addition to that, built-in operations _|_ and _&_ need to be supported. Avoiding ambiguities is a problem, in particular for larger terms containing in-line variable declarations of the form $t_1 : t_2$. To make these declarations parse as natural as possible, several typical symbols used to denote sorts (in t_2), such as _?, _+, _*, _[_], and _^_ are given a special treatment.

We should emphasize that this defines a *context-free* syntax of the Meta Notation. A property not yet checked by such a grammar is that symbols used in terms actually correspond to symbols introduced. For instance, a module like

```
module: Bool.
introduces: true, false, not _.
(1:) not tru = fal.
endmodule: Bool.
```

will parse without problems, even though the words tru and fal are obviously spelling errors. The next sections explain how we will catch such errors.

The SDF definition of MN serves two purposes. As a context-free grammar it can be directly used to obtain a parser and a (primitive) syntax-directed editor for the Meta Notation. As a signature, it provides an abstract syntax tree representation that can be used by functions that operate on Meta Notation modules.

5.4.2 Generating ASF+SDF modules

As we aim at re-using as much as possible from the ASF+SDF system, we will try to translate Meta Notation modules to ASF+SDF modules. Given the signature for MN modules as well as an SDF definition of the ASF+SDF formalism[1] (see, e.g., [HHKR89, Hen91]), we can introduce functions mapping MN modules to ASF+SDF modules. We will define these functions equationally using the ASF+SDF formalism. This specification can operate as an actual translator by executing the equations as

[1] Note that, in order to describe terms occurring in the equations part of ASF+SDF specifications, we can restrict ourselves to the term syntax used for the context-free syntax of ASD.

rewrite rules. The detailed description of the equations performing the translations needed for ASD fall out of the scope of this chapter. Interested readers are referred to the documentation describing the ASD implementation (written using ASF+SDF's literate specification facilities) that is part of the ASD release [DM94].

In addition to the equations describing the translations, we need a small amount of Lisp code to (1) dump the resulting ASF+SDF target code on a file, (2) to load generated modules into ASF+SDF Meta-environments in order to execute the generated code[2], and (3) to implement a small user interface to attach specified functionality to, e.g., buttons. This Lisp code is completely hidden to the ASD specifier, who interacts only with the graphical user interface.

A last issue we would like to mention is that each translator function operates on a single module (which can contain submodules). Thus, the translation is *modular*, i.e., the translator does not require global information concerning the entire specification in order to translate one module.

5.4.3 Four Basic Processors

We will consider four processors operating on MN modules, performing the following tasks: symbol checking, parser generation, term rewrite system generation, and sort checking of axioms. Each processor is implemented by generating, from an MN module, a corresponding ASF+SDF module performing the task of that processor for this particular module. In more detail, assuming M is an MN module:

- The EDIT processor aims at giving feedback while editing M, warning a specifier when he is using a symbol not actually introduced in M, or falling in an ambiguity. It operates by generating an SDF definition containing an M-specific grammar for terms and symbols, allowing only symbols actually introduced in M to be used in terms. Substituting this grammar for the general term syntax discussed earlier in Section 5.4.1, gives a context-free syntax for an MN module which knows only the symbols that were actually introduced.

  If we wish to distinguish between general and specific context-free syntax for MN, we will refer to the former as MN-Intro, and to the latter as MN-Edit.

- The ABSTRACT processor is concerned with abstract syntax as defined in grammar sections occurring in M. First of all, ABSTRACT will generate an SDF module M-ABSTRACT containing the grammar of M corresponding to the abstract syntax notation used in MN, which is also the format used by semantic equations. Secondly, as grammar definitions in MN are deliberately chosen to be as close to the concrete syntax as possible, it is possible generate a more concrete grammar as well. Finally, an ASF+SDF definition is derived containing functions mapping terms over the concrete signature to terms in abstract notation.

- The REWRITE processor handles axioms that can be executed as term rewriting rules. The specifier can explicitly indicate such axioms by giving them an `[`N`:]`

[2] The algebraic specification formalism ASF+SDF does not have an *assert*-like (as in Prolog) facility to load rules created by the specification. Therefore loading is done using external Lisp code.

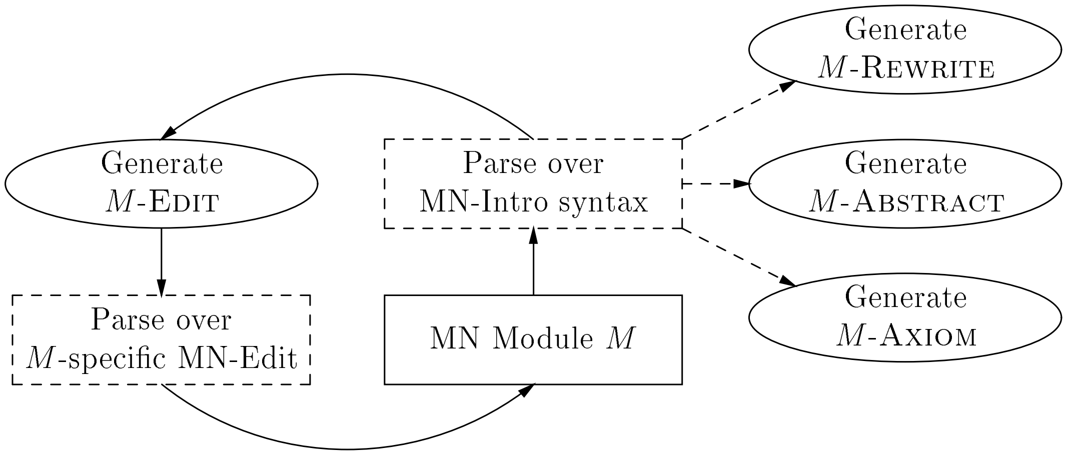


Figure 5.7: Overview of the four ASD processors

tag (rather than `(`N`:)`). It operates by generating a module M-REWRITE containing equations in ASF+SDF format of the `[N:]`-tagged equations occurring in M.

In order to apply rewrite rules as intended, ASF+SDF needs sort information for terms. Therefore, M-REWRITE contains a sorted signature which is derived from introducing clauses, functionalities, and `(*)`-tagged sort equations occurring in M.

- Finally, the AXIOM phase aims at checking the well-sortedness of all axioms. To that end, it generates an ASF+SDF module M-AXIOM containing *dummy* equations in ASF+SDF, i.e., equations not intended to be meaningful or executable, but which can be sort-checked by ASF+SDF. The AXIOM phase uses the same sorted signatures as the REWRITE phase.

A pictorial overview of the four processors is given in Figure 5.7. During development of ASD specifications, the main cycle involves editing and checking symbols (using EDIT). Once the editing is finished, the other processors can be invoked. All generators operate on abstract-syntax trees over the MN-Intro signature.

The implementations of all four processors have one so-called *canonification* phase in common. The full Meta Notation is rather rich, and can be reduced to a kernel subset. For instance, $\leq$ can be expressed using $\geq$, `introduces: s1, s2.` is identical to `introduces: s1. introduces s2.`, etc. The rewrite rules performing this reduction to canonical form are based on the equations given in [Mos92, Appendix F.2.2]. The canonification phase also takes care of eliminating nested modules.

5.4.4 Ambiguities

Before proceeding to explain the various processors in full detail, let us take a look at a frequently recurring problem in ASD: syntactic ambiguities.

Since SDF does not impose any restriction on the context-free grammar rules used, specifiers are allowed to write ambiguous grammars. They can define *priorities* between grammar rules in order to resolve certain ambiguities, but these priorities need not cover all cases. Parsing in SDF is implemented by forking into different parsers if an ambiguity occurs, which can potentially lead to an explosion of information to be dealt with. At the end, once all the possibilities are known, the priority declarations are used to reject certain parses. If several possibilities remain, an interactive dialogue is started asking a user to choose one.

In itself this freedom and the possibility to deal with ambiguity is exactly what is needed to implement a syntactically complicated language like the Meta Notation. However, in combination with another property of SDF, the possibility of *lexical* ambiguities, this can have disastrous effects. An initial version of the term syntax as discussed in Section 5.4.1 was not careful enough with lexical syntax. This led to an explosion of possibilities at the lexical level, with an explosive effect on parsing time. For instance, a sentence (with a unique parse at the end!) consisting of 10 words would parse in 15 seconds, while concatenating the same sentence to itself giving one of 20 words would not require the expected 30 seconds maximally, but 15 *minutes*!

Most lexical analyzers yield one stream of lexical tokens. They use disambiguation rules like *longest match* — if there are several potential tokens choose the longest one — to restrict the number of possibilities. SDF uses lexical disambiguations rules as well, but these are less restrictive, in order to be able to cope with lexically difficult languages like FORTRAN. Instead of *longest match*, a rule called *longest match per sort* has been adopted [HHKR89, Section 4.9]. For example, if we have alpha-numeric identifiers and numbers, the lexical analyzer of SDF will recognize that a character sequence like `abc12` can be interpreted as the identifier `abc` followed by the number `12`, or as just the single identifier `abc12`. In SDF, the parser will subsequently make a selection between these alternatives.

Once one is aware of these problems, one can take measures to avoid them; the current version of the term syntax is carefully constructed to eliminate this problem.

5.5 Generating EDIT modules

A typical use of the EDIT processor involves the following sequence of events (see also the EDIT cycle in Figure 5.7):

1. An MN specifier opens a syntax-directed editor for an MN file. It is parsed over MN-Intro.

2. A corresponding SDF file is generated containing context-free grammar rules for symbols introduced. For instance, if the MN module contains the line

   introduces: `not _`

   the generated EDIT module contains, in essence, the following SDF rule:

   exports context-free syntax *"not"* TERM → TERM

3. The generated EDIT modules are loaded into a separate instance of the ASF+SDF system. This generated system only knows about EDIT syntax and the specific TERM sort; not about the general term syntax of Section 5.4.1.

4. The syntax-directed editor is attached to the new system, and its grammar is changed into the new EDIT grammar.
5. Parsing according to this new syntax can be used to detect the erroneous use of symbols.
6. If the user wishes to introduce other symbols, or change existing ones, this process is repeated. The syntax to be used by the editor is switched back to MN-Intro. A new EDIT file is generated and loaded. Finally, the syntax of the editor is again switched back to the EDIT-specific grammar. The user can simply start this process by one button-click.

Note that symbols introduced in included or needed modules are catered for as well; **includes** or **needs** clauses are translated to imports of the corresponding generated EDIT module.

ActionNotation-EDIT (Module 5.2) is a detailed example of a generated EDIT module. Its details, the WORDs, and the U-, R-, and G-TERMs, will be discussed in the next two sections.

5.5.1 Priorities

Concerning ambiguities in terms, in [Mos92, p. 41] it is noted that "it is convenient to have the following *precedence rules* in terms: outfixes have the weakest precedence, followed first by infixes, then by prefixes, and postfixes have the strongest precedence. Moreover, infixes are grouped to the left." Outfix symbols include, for example, `sum(_,_)` as it is closed at both ends.

SDF's facilities to deal with ambiguities include *priority declarations*, each expressing the relative precedence of two context-free functions. In order to implement the above precedence specification, one can introduce three "dummy" context-free functions called, e.g., `"infix"`, `"prefix"`, and `"postfix"`. Each normal context-free function is given a priority relative to one of these three, which by transitivity takes care of all priorities. Moreover, all infix context-free functions are made left-associative using SDF's **left** attribute.

This scheme, however, has the following disadvantages. As a typical MN specification will introduce many non-outfix symbols, the generated EDIT modules will contain many priority declarations. It turned out that the SDF implementation implicitly assumes a more "normal" use of priorities, where say at most 10 priorities are declared. A full generated EDIT specification, however, could easily include more than 100 priorities. Just loading the modules containing this many priorities takes an excessively long time.

A second potential disadvantage of using priority declarations is that the parsing time increases. Recall that SDF parsers may fork for each ambiguity, using priority information at the end to eliminate certain choices. Therefore, ambiguities always make parsing slower, even if they can be solved.

module ActionNotation-EDIT
imports MetaNotation-EDIT
exports
lexical syntax

"*action*"	$\rightarrow$	WORD
"*primitive-action*"	$\rightarrow$	WORD
"*complete*"	$\rightarrow$	WORD
"*unfolding*"	$\rightarrow$	WORD
"*or*"	$\rightarrow$	WORD
"*and*"	$\rightarrow$	WORD
"*then*"	$\rightarrow$	WORD
"*bind*"	$\rightarrow$	WORD
"*to*"	$\rightarrow$	WORD
"*rebind*"	$\rightarrow$	WORD
"*furthermore*"	$\rightarrow$	WORD
"*moreover*"	$\rightarrow$	WORD
"*A*"	$\rightarrow$	WORD

context-free syntax

"*action*"	$\rightarrow$	G-TERM
"*primitive-action*"	$\rightarrow$	G-TERM
"*complete*"	$\rightarrow$	G-TERM
"*unfolding*" R-TERM	$\rightarrow$	R-TERM
U-TERM "*or*" R-TERM	$\rightarrow$	U-TERM
U-TERM "*and*" "*then*" R-TERM	$\rightarrow$	U-TERM
"*bind*" U-TERM "*to*" R-TERM	$\rightarrow$	R-TERM
"*rebind*"	$\rightarrow$	G-TERM
"*furthermore*" R-TERM	$\rightarrow$	R-TERM
U-TERM "*moreover*" R-TERM	$\rightarrow$	U-TERM

hiddens
context-free syntax
"*A*" $\rightarrow$ G-TERM

Module 5.2: ActionNotation-EDIT. Generated module

Therefore, we decided to choose a solution that aims at avoiding ambiguities and priority declarations altogether. The single sort TERM can be split into several subsorts corresponding to the four levels of precedence needed (see, e.g., the term, factor, and expression example in [ASU86, p.32]). Sort G-TERM contains grouped terms, i.e., just outfixes. It is a subsort of L-TERM, which contains terms expecting an argument at their left (e.g., `furthermore _`) but not at their right. Sort L-TERM is a subsort of R-TERM for functions expecting an argument at the right, which in turn is a subsort of U-TERM containing all other symbols. Carefully giving the various argument positions the proper sorts eliminates the ambiguities. For example, the right-most argument of an infix operator is required to be of sort R-TERM. This scheme eliminates ambiguities during parsing as early as possible, and avoids the need for loading too many priorities.

5.5.2 Name Clashes

It is possible to use the same symbol checking mechanism to test whether the variables used are actually declared, either inside a term or using the special **variables:** construct. Each variable declaration $x : s$ or $x \leq s$ leads to a context free grammar rule "x" $\rightarrow$ TERM. Such single letter words, however, involve a certain risk. What to do if two consecutive words form a larger symbol as well? For instance, using variable `l` and `t`, and symbols `_ is _` and `list`, the character sequence `list` can be read as `list` or as `l is t`.

In the presence of lexical longest match per sort, we initially expected that the longest keyword, `list`, would be chosen. However, all context-free keywords are considered as belonging to different sorts, so both alternatives are lexically allowed.

In order to avoid these ambiguities, we decided that we wanted the longest word in these cases. In order to implement that using SDF, we used its REJECT facility [HHKR89, Section 4.8]. Each generated Edit module contains a hidden lexical section where each word occurring in a symbol is declared as being of a sort "WORD". One module imported by all Edit modules contains the line

exports lexical syntax WORD WORD $\rightarrow$ REJECT

which states that tokens built from several consecutive WORDs are to be discarded, or, paraphrasing, that the accepted tokens must be built from exactly one WORD.

5.5.3 Using Edit for DN, AN, Ad, and Pascal

In addition to these four processors manipulating MN modules, ASD comes with the Data and Action Notation libraries, as well as the example definitions of Ad and Pascal. The ASD versions of these specifications were easily obtained by using a translator[3] mapping the LaTeX sources (disciplinedly written using a set of macros) to the ASCII version supported by ASD. This allowed us to check the literal texts as they occur in [Mos92, MW93] using the new ASD tools.

[3] The LaTeX to ASD translator was written in Asf+Sdf and is part of the ASD distribution.

Doing so gave us a nice opportunity to check the effectiveness of the ASD tools. The specifications of DN, AN, and Ad were published in a book, and therefore carefully proof-read by various people. They were used in several student courses. Great care was taken to make these definitions as error-free as possible. Moreover, most spelling errors occurring in them were caught by a clever use of programs like "makeindex" and "ispell", and therefore removed before publication.

Nevertheless, the ASD tools found about one error on every two pages. Most of these were of a rather trivial nature, for instance forgotten punctuation, omitted words in function names (using `yielded` rather than `yielded by`), or words put in the wrong order (e.g., `[...] map` instead of `map [...]`). Other errors were easily made because function names were very similar. For instance, AN introduces the symbol `agent`, and the Ad definition moreover used `task` as well as `task-agent`. At various spots, agents, tasks, and task-agents were mixed up. Similarly, in a context dealing with `redirect ...`, the word `redirection` was used instead of the correct `indirection`. Performing the EDIT checks also revealed several errors in the import structure; some required imports were forgotten, and some symbols that were introduced privately were used outside their module.

Some errors that could have caused more confusion are the use of `give the datum stored in c` instead of the intended `give the uninitialized stored in c`, or the use of `map [token to variable]` instead of the correct `yielder[of map [token to variable]]`.

Rather surprising were the large number of extra brackets that had to be inserted for disambiguation, not counting the brackets that were added (automatically) because indentation was not used anymore for disambiguation. A fairly typical example is

```
the negation of the integer yielded by x
```

built from functions `negation of _`, `the _`, `the _ yielded by _`, and words `integer` and `x`. This sentence can be grouped in two different ways:

```
the (negation of the integer) yielded by x
the negation of (the integer yielded by x)
```

The natural reading is the second grouping, and one easily forgets the necessary brackets. This pattern occurred very often, and in all cases brackets had to be added to enforce the latter binding.

Other ambiguities arose because too many function symbols were too similar; in the Pascal description, e.g., the symbols `uninitialize _`, `uninitialize all _`, and `all _` were used. In other cases, extra brackets had to be added since the left-associativity of infixes was not the intended grouping. For instance, `x is y or z` easily reads like a correct expression, but what is usually meant is `x is (y or z)`.

5.6 Generating REWRITE modules

The EDIT processor discussed in the previous section guarantees that all symbols and variables used in a set of modules M are declared properly if and only if all modules pass through the EDIT processor without errors. Similarly, the EDIT processor guarantees that *all* ambiguities occurring in terms are detected and solved.

The REWRITE and its related ABSTRACT and AXIOM processor, by contrast, are of a more approximate nature. Many unified algebraic specifications are written according to certain conventions. These are exploited and used to implement MN modules as many-sorted ASF+SDF modules.

5.6.1 Functionalities and Sort Equations

A first decision to be made is which symbols should correspond to (i) sorts in the generated many-sorted signature, (ii) constants or functions acting as elements, and (iii) sort operations. We will illustrate our decisions using a small subset of the Truth-Values modules, Figure 5.8, of which the corresponding generated REWRITE version is displayed in Module 5.3.

In principle, every alpha-numeric word introduced is interpreted as a sort. In our example, `truth-value` is such a word. It causes an SDF sort declaration to be generated, and some basic context-free syntax corresponding to built-in operations of MN, such as `(...)`, `_|_`, and `nothing`. The first six context-free rules of module Truth-Values-REWRITE are generated just by the introduction of `truth-value`. The *individual* attribute is used to conclude that it is not necessary to generate full sorts for `true` and `false`, but that they can just be declared as constants of sort U-truth-value.

Functionality assertions like those for `not _` and `any _` are used to generate SDF context-free functions. The argument or result sorts specified in such functionality axioms can be simple alpha-numeric words (as for `not_`), or they can include *sort-operations*, as for `any_`. A limited set of sort operations is supported by the ASD REWRITE processor: only those corresponding to so-called tuple operations, consisting of `_?`, `_*`, `_+`, `(_,...,_)`, and `_^`$\{N\}$. All these are implemented by generating a new SDF sort. In the example this is shown for the `_*` operator, introducing a new sort `U-truth-value-STAR`, as well as context-free syntax for it. Finally, although not shown in the example, `(*)`-tagged equality or subsort equations are used to generate SDF chain rules (injections).

The problematic issues involve translations and generic operations. Neither of these are supported by SDF. An ASD user can circumvent this by explicitly giving the functionality axioms corresponding to the expansion of the translation or to the instantiation of a generic function. Overloading is supported by SDF.

A final issue concerns the ubiquitous danger of ambiguities in sentences parsed over generated SDF rules. The brackets `(...)` generated for each sort, and the injections giving different trees for the same sentence are potential sources of ambiguities. Therefore, a priority between the `(...)` brackets of the two sorts is generated for injections (shown in the example as well). Moreover, the ASD user can augment his specification with $t : s$ or $t \leq s$ assertions, which force the term t to be of sort s. In our example, the generated function U-truth-value ":" "*truth-value*" → U-truth-value can be used for these purposes by the corresponding ASF+SDF term generated for t.

```
module: Truth-Values.
  introduces: truth-value, true, false, not _, any _.
  (*) truth-value = true | false (individual).
  (*) not _ :: truth-value  -> truth-value.
  (*) any _ :: truth-value* -> truth-value.
endmodule: Truth-Values.
```

Figure 5.8: Subset of symbols and functionalities for Truth-Values

module Truth-Values-REWRITE
imports Z-LAYOUT
exports
  sorts U-truth-value U-truth-value-TUPLE-STAR
  context-free syntax

U-truth-value "*&*" U-truth-value	→ U-truth-value	{**left**}
U-truth-value "|" U-truth-value	→ U-truth-value	{**left**}
"*nothing*"	→ U-truth-value	
"(" U-truth-value ")"	→ U-truth-value	
"*truth-value*"	→ U-truth-value	
U-truth-value ":" "*truth-value*"	→ U-truth-value	
"*true*"	→ U-truth-value	
"*false*"	→ U-truth-value	
"*any*" U-truth-value-TUPLE-STAR	→ U-truth-value	
"*not*" U-truth-value	→ U-truth-value	
"(" U-truth-value-TUPLE-STAR ")"	→ U-truth-value-TUPLE-STAR	
{U-truth-value ","}*	→ U-truth-value-TUPLE-STAR	

  priorities

{ "("U-truth-value ")" → U-truth-value } >
{ "("U-truth-value-TUPLE-STAR ")" → U-truth-value-TUPLE-STAR }

Module 5.3: Truth-Values-REWRITE. Generated module for Truth-Values

module Truth-Values-YIELDER
imports Truth-Values-REWRITE Z-Yielder-YIELDER
exports
context-free syntax
"*not*" R-yielder → R-yielder
"*any*" R-yielder → R-yielder
priorities

{ "*not*" R-yielder → R-yielder } < { "*not*" U-truth-value → U-truth-value },
{ "*any*" R-yielder → R-yielder } < { "*any*" U-truth-value-TUPLE-STAR → U-truth-value }

Module 5.4: Truth-Values-YIELDER. Extensions of *data-operation*.

5.6.2 Data Operations and Yielders

The Action Notation library uses two sorts `yielder` and `data` which require special treatment. The sort `data`, built from `datum*`, is left open, to be specialized in the semantic entities module (see, e.g., Figure 5.3). It typically contains all data types occurring in the programming language defined, such as strings, integers, files, and so on. The sort `yielder` consists of expressions to compute the data values needed by actions for this language. The yielders include language independent operations on bindings and stores, as well as all specific data operations.

In order to specify this `data`-part of the `yielder`-sort of AN, the appendices of [Mos92] make an excursion to *meta*-meta-notation: a special italic *data-operation* is introduced, with a suggestive functionality including "..." to indicate its number of arguments [Mos92, Appendix B.1.2]. In other words, "every *data-operation* (i.e., operation specified for arguments included in `data`) is extended to arguments of sort `yielder`" [Mos92, Appendix D.1.2].

Checking whether a sort is a subsort of `data` requires information from all modules in the specification, which conflicts with ASD's approach of processing one module at a time. Therefore the following heuristic is used: Every functionality not using the sorts `action` or `yielder` is assumed to be a functionality of a data-operation. Again, extra priorities are generated in order to avoid potential ambiguities between the data-operation itself and its yielder version. See Module 5.4 for the generated YIELDER module for MN module `Truth-Values`.

5.6.3 Grammar Sections

Grammar sections provide the interface between actual programs and their semantic descriptions. Let us study the example of the rule from Figure 5.4:

```
Statement = [[ Identifier ":=" Expression ]]
```

All generated ASF+SDF corresponding to this production is shown in Module 5.5. The first context-free syntax declaration in that module corresponds to the concrete syntax, as used by, e.g., Pascal programmers. The second line gives a syntax for the format

module Generated-from-Grammar
imports Z-LAYOUT
exports
  sorts C-Identifier C-Expression C-Statement
  sorts A-Identifier A-Expression A-Statement
  context-free syntax

C-Identifier ":=" C-Expression	$\rightarrow$ C-Statement
"⟦" A-Identifier "\\" :=\\"" A-Expression "⟧"	$\rightarrow$ A-Statement
"*map-c2a-Statement*" "\\"" C-Statement "\\""	$\rightarrow$ A-Statement
"*map-c2a-Identifier*" "\\"" C-Identifier "\\""	$\rightarrow$ A-Identifier
"*map-c2a-Expression*" "\\"" C-Expression "\\""	$\rightarrow$ A-Expression

hiddens
  variables
    C-Identifier-V1 $\rightarrow$ C-Identifier
    C-Expression-V2 $\rightarrow$ C-Expression

equations

$$\begin{aligned} &\textit{map-c2a-Statement}\text{''}\ \textit{C-Identifier-}V_1 := \textit{C-Expression-}V_2\text{''} \ = \qquad [1]\\ &\qquad [\![\ \textit{map-c2a-Identifier}\text{''}\ \textit{C-Identifier-}V_1\text{''}\\ &\qquad\ \text{``:=''}\\ &\qquad\ \textit{map-c2a-Expression}\text{''}\ \textit{C-Expression-}V_2\text{''}\ ]\!] \end{aligned}$$

Module 5.5: Generated-from-Grammar.

expected by action semantics, and hence expected by, e.g., the REWRITE phase. Next, for each sort involved, a function is introduced mapping the former (concrete) version to the latter (abstract) version. Each function is defined inductively over the concrete syntax, as shown in the equations section for the C-Statement function. The names of these mapping functions are non-standard, since they contain the "..." delimiters as part of their name rather than the more conventional (...) brackets. This was needed in order to *specify* the *generation* of these functions within ASF+SDF; only within strings arbitrary text can be written.

The operations allowed to specify grammars are richer than those allowed in SDF. It is, however, relatively easy to generate extra sorts and extra context-free syntax simulating the effect of `_?`, `_*`, `_+`, `|`, or `<...>`. Not all flat lists built by the latter operation can be implemented using SDF, but one of its important uses, e.g., `<Stat <";" Stat>*>` can be implemented using SDF's {Stat ";"}+. Lexical syntax is assumed to be defined in rules using the sort `character` and its subsorts such as `lowercase-letter` or `digit`, or in rules using character constants `'...'`.

Note, finally, that the concrete syntax generated again suffers from the danger of ambiguity. Unlike SDF, the MN grammar specifications do not allow writing priority declarations.

5.6.4 Rewrite Rules and Axioms

If certain axioms in an ASD MN module M are intended to be executable as a rewrite rule, a specifier can indicate this by giving the axiom an `[`N`:]` tag. Such axioms are copied directly into ASF+SDF module M-REWRITE, so that the ASF+SDF Meta-environment rewrite engine can be used to execute these rules. The only change made to the rules is that in-line variable declarations are taken out of it, and passed to a hidden SDF variables section.

It is necessary to rewrite over a *sorted* signature, and not, e.g., over the unsorted EDIT signature. For instance, the distinction between identifiers and literals exploited by the rules

```
[1:]  evaluate I:Identifier = give the entity ...
[2:]  evaluate L:Literal = give the value of ...
```

(occurring in the semantic functions of Ad) would be lost in an unsorted framework. As a consequence, all equations to be executed must be well-sorted (parse-able) over the signatures derived as described in Section 5.6.1.

This sort checking proved a useful device to find errors in ASD specifications. In order to allow sort checking on non-executable axioms as well, the AXIOM phase produces an ASF+SDF module containing *dummy* equations of (N:)-tagged axioms. These dummy equations should parse properly, but are not intended for execution.

5.7 Using ASF+SDF

Although ASF+SDF has been used intensively in various large specifications, there does not seem to be a single application that exploits so many features in such an intensive way. In general, it was impressive to see that ASD could indeed be implemented so easily, although some issues turned out to be more problematic than expected. Here we list some of the more noteworthy difficulties (in arbitrary order):

- The lexical disambiguation rule *longest match per sort* is (i) not needed for several languages, including the MN, and (ii) can have a disastrous effect on parsing time due to extra ambiguities. It should be possible to switch *longest match per sort* on or off explicitly.

- All keywords (literals) can be considered to belong to one lexical sort, instead of assuming each of them to be of a separate sort (see Section 5.5.2). This would make *longest match per sort* applicable to keywords as well.

- A priority declaration like "$*$" > "+" forbids a "+" operator to occur as a direct child of a "$*$" node; only bracketed "+" operations can be arguments of "$*$". This restriction holds for *all* arguments of "+", which in this case is exactly what is needed. However, for a symbol "open only at one side", like "[max_]_", one will be interested in priorities for the *open* (second) argument position only. As priorities apply to all arguments, priority conflicts can occur in unambiguous trees.

  One would like to be able to specify to which argument positions a priority declaration applies.

- An example of the previous problem is the *dangling-else* problem [ASU86, p.174], which is claimed to be solved by priorities in the SDF reference manual [HHKR89, Section 6]. Indeed, the given priority:

  "if" EXPR "then" STAT "else" STAT > "if" EXPR "then" STAT

  makes sure that an "if-then" occurring at the second argument position (first STAT) is rejected. With the given priority, however, an "if-then" at the *third* position is not allowed either. Consequently, the following unambiguous and correct sentence, which is allowed in ISO Pascal [ISO83] is rejected by SDF owing to a priority conflict: `if E1 then skip else if E2 then skip`.

- Loading an ASF+SDF module using priority declarations in a system already containing many priorities takes an excessively long time.

- As a last disambiguation phase, the SDF parser uses the (not yet documented) rule of choosing the alternative with fewest injections occurring in it.

  This turned out to be a crucial disambiguation rule. However, with grammars containing many injections (see, e.g., ASF+SDF module Terms (Module A.3) in the appendix), it sometimes made unintended choices.

It would be useful if it were possible to trace the decisions made by the parser, or to get a warning if the numbers of injections are actually used.

- Generalizing the previous point, an SDF debugging tool could be helpful. Its main use would be the detection of sources of ambiguities. Its functionality could include the display of trees, to show ambiguities, to indicate which priority declarations played a role, or even to help the specifier find rules that could cause ambiguities (an undecidable property, though).

- The LeLisp interface functions to a running ASF+SDF Meta-environment must be documented in an *ASF+SDF Programmer's Manual*.

- Currently, no clean facility is offered to load generated ASF+SDF modules into a running system. One way could be to extend ASF+SDF with a Prolog like *assert* facility, which would add a list of equations to a running system.

  A more controlled way could be to extend the *Semantics-Directed Environment Adaptation Language* SEAL [Koo94] which is part of the ASF+SDF Meta-environment. SEAL is used to attach functionality to user-interface events, and to combine terms occurring in different syntax-directed editors to new editors or files. However, currently no support for manipulating modules is given.

- The incremental behavior of the ASF+SDF Meta-environment is probably responsible for the long loading times for ASF+SDF modules, even if they do not contain priority declarations. Once loaded, however, reloading modules is very efficient, as only the actually changed lines are processed. This is particularly relevant for the — interactive — EDIT phase.

- A limited form of memo functions would be useful to avoid re-computation of intermediate results when, for instance, extracting several subcomponents from one large computed result. The ASD implementation now uses some Lisp code to achieve this effect.

- The current role of injective functions like "context-free syntax ID → EXPR" is very unclear. The remarks in the SDF reference manual suggest that injections are treated as subsorts [HHKR89, Section 5.4], which would be ideal for the implementation of ASD. However, the actual ASF+SDF implementation sometimes treat injections as *functions* which just happen to be invisible.

- *Parameterization* (not implemented in ASF+SDF) was not necessary to implement ASD. *Renamings* would have made it possible to implement *translations*, and *hidden imports* to support *needed* modules.

- The ASF+SDF formalism supports the use of concrete syntax in equations. In the implementation of ASD, the equations describing the translation to equations could easily have an ambiguous concrete syntax. Such ambiguities would have been absent if it was possible to use abstract syntax instead.

It might be possible to combine the advantages of both when the specifier can indicate that for certain function symbols he or she only wishes to use abstract syntax (when used in an equations section). This boils down to mixing functions written in normal concrete notation with other functions written in abstract notation.

The so-called ASFIX fixed exchange format for terms proposed in [Kli93b] can serve as a starting point for work in this direction.

- Many functions occurring in the ASF+SDF specifications defining the mapping from MN to various ASF+SDF modules, are defined by simple induction over some structure, e.g., the MN-Intro syntax.

  The latter was subject to quite a lot of changes, and all functions had to be updated over and over again.

  It would have been useful if the notion of "inductive definition", or "primitive recursion" would have been more strongly supported by the ASF+SDF Meta-environment. In particular, if the constructors of a data type are changed, it would be useful to inform the specifier which functions and more specifically which equations are affected.

5.8 Concluding Remarks

In this chapter, we have discussed a complex set of tools supporting the development of action-semantic descriptions. The complications in the tools mainly arise from:

- the arbitrary syntax as supported by the Meta Notation;
- the Lisp programming needed to load generated ASF+SDF modules into a running ASF+SDF Meta-environment;
- the powerful features of UA to define sort operations; (the implementation supports strings, trees, and tuple (list) operations); and
- (for the REWRITE and AXIOM processor only) the special treatment that needs to be given to the sorts `yielder` and `action`, in order to deal correctly with *data-operations*.

The potential usefulness of the ASD tools has been illustrated by their success in spotting errors in existing MN specifications or language definitions. The current version, however, is a prototype, and the performance is not always as one would wish. Several improvements are still possible; we have, for example, plans to minimize the number of priority declarations generated, thus further reducing the load time of generated modules.

Using ASF+SDF to build the ASD tools allowed us to experiment with various implementation schemes. All these schemes have led us through the most remote corners of ASF+SDF, which resulted in a list of issues in which the ASF+SDF formalism or system could be improved.

Part II

Origin Tracking

Chapter 6

Principles of Origin Tracking

Algebraic specifications of programming languages can be used to generate language-specific programming support tools. Straightforward tools can be obtained by executing these specifications as term rewriting systems. More advanced tools can be constructed if the term rewriting machinery is extended with *origin tracking*. Origin tracking is a technique which automatically establishes a relation between subterms in a result (normal form) or intermediate value and their so-called *origins*, which are subterms in the initial term. Origin tracking can be used to associate positional information with messages in error reports, visualize program execution, and to construct language-specific debuggers. We give a definition of origins, and present a method for implementing origin tracking.

6.1 Introduction

Term rewriting is a convenient technique to execute the algebraic language specifications as we encountered them in Part I of this thesis. Is it, however, a sufficiently advanced technique? Do we extract the best possible tools from the specifications if all we can do is a simple reduction of a term to its normal form?

In this second part of the thesis we study a particular technique which aims at generating significantly more sophisticated tools from algebraic language specifications. The underlying idea of this technique, called *origin tracking*, is that we can derive better tools if we have a better understanding *how* the initial term was reduced to its normal form. Which rewrite rules were used? Which new function symbols were created by which rule? For a single reduction step, were there any subterms occurring in the redex that literally recur in the contractum? And how are subterms propagated over multiple reduction steps?

Answers to questions like these will be given by the *origin tracking* technique. For a reduction $t \rightarrow^* t'$, it aims at establishing relations between subterms occurring in the result t', and subterms — their so-called *origins* — in the initial term t. The reasons for studying these relations are of an application-oriented nature. We are in particular seeking solutions for the following three problems:

- Associating positional information with messages in error reports.

  In Chapter 4, we studied a specification of the static semantics of Pascal. Executing this specification using plain term rewriting just yields normal forms representing error messages. Can we use the same specification to obtain a type checker that also explains *where* in the Pascal program the source of the error is?

- Visualizing program execution.

  A specification of the dynamic semantics of a language, e.g., an operational semantics, can be executed using plain rewriting in order to obtain an evaluator of the language in question. Can we use the same specification to obtain an *animator* for that language, i.e., an evaluator which indicates the statement that is currently being executed in some distinctive way?

- Constructing language-specific debuggers.

  A specification of a translation between a programming and an assembly language can be executed using term rewriting in order to get a prototype compiler. If we detect an error while running the compiled program, we would like to perform debugging activities (setting breakpoints, inspecting values of variables) in terms of the *source* program rather than in those of the assembly language. Can we use the translation specification to generate such a source-level debugger?

Origin tracking is the key to a positive answer to these three questions. In this chapter, the principles of origin tracking are presented. Origins are introduced by a small example, after which they are defined more rigorously, and analyzed in full detail. It is explained how origins can be extended to conditional TRSs, implemented, and applied to the generation of tools from language definitions. In the next chapter, a specialized definition is given for specifications in the so-called "Primitive Recursive Scheme" format. In Chapter 8, finally, an extension of origin tracking to higher-order specifications is described.

6.2 Related Work

The study of origins was pioneered by Bertot [Ber90, Ber92, Ber93]. He investigated applications of origin tracking to source-level debugging given a specification in natural semantics style [Kah87, Ber90]. Furthermore, he considered the relation between origins for the λ-calculus and for TRSs [Ber92], and introduced a formal framework for reasoning about origin functions [Ber93]. Bertot focused on orthogonal, unconditional TRSs, where an origin consists of at most one subterm occurrence. His research was conducted in the framework of the Typol [Des88] language implemented in the Centaur [BCD$^+$89] system. This system uses *subject-tracking* in order to maintain origin-like information.

The idea to study relations between the subterms of an initial term and those of the normal form is not new. It was applied successfully in the search for optimal reduction strategies, i.e., a strategy that indicates which redex to contract in order to minimize the total number of reduction steps [HL91, Mar91]. So-called *residual map*, or *descendant relations* are being used to keep track of subterm recurrences, and in particular of *redex* recurrences. The origin relation can be considered as being an extension of these ideas.

The applicability of the origin tracking techniques to animation and error location is studied in detail by Dinesh and Tip [Din93, Tip93]. Dinesh presents a specification style for the definition of static semantics of programming languages based on abstract interpretation. He shows how origins can be used to generate type checkers with good error pinpointing facilities. Tip discusses an algebraic specification of an interpreter for a Pascal-based language, and explains how origin tracking can be applied to obtain an animator for this language.

Field and Tip are studying the applicability of origin tracking in the area of *program slicing* [FT94]. They propose an origin-like relation called the *dynamic-dependence* relation. In fact, although called a *dependence* relation, their technique manages to identify *independencies* as well. Their relation discovers which parts of the initial term were irrelevant for the computation of the normal form. This information is used to extract *program slices* for programming languages, where all parts in a program irrelevant for the computation of a particular value are eliminated. Rather than relating *subterms*, they focus on relating so-called *sub-contexts*, i.e., subterms with holes. Field and Tip report on applications in the area of theorem-proving systems as well.

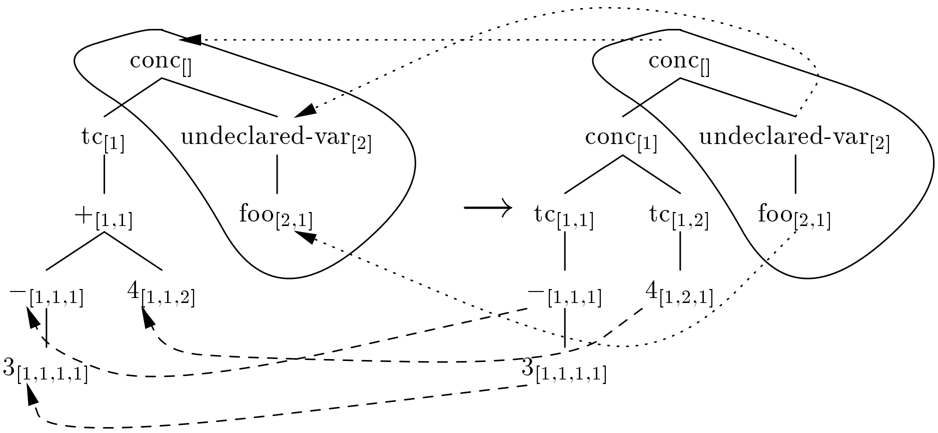
6.3 Origin Tracking in a Nutshell

Before diving into the details of origin tracking, let us try to obtain an understanding of how origin tracking works. Consider a reduction of a term t_0 to t_n, in a sequence of rewrite steps $t_0 \to t_1 \to \cdots \to t_n$. During such a reduction, we annotate every function symbol occurring in some t_i $(0 \leq i \leq n)$ with a set of *occurrences* in the initial term t_0. An occurrence identifies a subterm, and consists of a path from the root to that subterm (an occurrence acts as a sort of *pointer*).

Defining an origin function involves the definition how the origin annotations are affected by each rewrite step $t_i \to t_{i+1}$ $(0 \leq i < n)$. In such a rewrite step, a certain rewrite rule $\alpha \to \beta$ is applied at a particular occurrence in t_i (the *redex* position). It is below this occurrence that the actual changes in t_i take place; above or next to this redex position (i.e., in the *context*), the term does not change. Therefore, origin annotations in the context simply remain the same. These origins are called *Context* Origins.

The redex part of t_i is changed by $\alpha \to \beta$ to a new term, the *contractum*. The answer to the question which origins to establish for this contractum will be given by an analysis of the rewrite rule. Recall that for a rule $\alpha \to \beta$, the variables occurring in β must occur in α as well. In other words, every subterm corresponding to a variable occurrence at the right-hand side must occur in the left-hand side. Therefore, we will establish relationships between all variable occurrences in the right-hand side, and their counterparts in the left-hand side. These origins are called Origins for *Common Variables*.

An example with a picture may help to clarify this. Figure 6.1 shows a single-step reduction. The term



Context:	conc($\square$, undeclared-var(foo))
Rewrite rule:	$\text{tc}(E_1 + E_2) \rightarrow \text{conc}(\text{tc}(E_1), \text{tc}(E_2))$
Substitution:	$\{E_1 \mapsto -3,\ E_2 \mapsto 4\}$
Dashed Lines:	Origins for Common Variables
Dotted Lines:	Context Origins.

The occurrence leading to a function symbol is given as a subscript of that symbol: e.g. taking the first son of the second son of the leftmost term leads to "foo", which is at occurrence [2,1].

Figure 6.1: Origins established for one rewrite step.

conc(tc(-3 + 4), undeclared-var(foo))

is rewritten to

conc(conc(tc(-3), tc(4)), undeclared-var(foo))

In a tree representation, these two terms are given in the figure. The context of this reduction is the term

conc($\square$, undeclared-var(foo))

In the picture, this context is enclosed in a pear-formed shape for both terms.

The origins in this figure are indicated by the arrows. The dotted arrows correspond to origins in the context of this reduction; they simply point to the same occurrence in the left term. The dashed arrows correspond to Common Variables. For instance, the two lines from "$-$" and "3" correspond to the value (instantiation) of the variable E_1 in the rewrite rule.

In the next section, we will make the origin definition more precise by replacing the arrows by sets of occurrences. The origin associated with the "4" symbol at occurrence $[1, 2, 1]$ at the right will then become the set $\{[1, 1, 2]\}$.

In that same section, we will analyze origins in considerable depth. We will particularly study when function symbols will not obtain any origins. For instance, in the picture there are no arrows leaving the "conc" node at occurrence [1]. We will try to understand under which conditions such empty origins appear. Note that for practical purposes, such empty origins are undesirable (see Section 6.7 for a discussion of the applicability of origin tracking in practice).

Context- and variable origins together, to which we will refer as *primary* origins, are rather basic. In order to establish more origins, we will study two other origin rules, which will be called *secondary* origins. We will experiment with linking the top of the redex to the top of the contractum (this is not shown in the Figure, but would result in an arrow from "conc" at [1] to "tc" at [1]). Moreover, we will study the effect of relating common subterms occurring in the uninstantiated sides of the rewrite rule (this case is not applicable to the rule used in the picture).

This latter rule may in some cases build unintended relations, for instance between two constants that just happen to occur at both the left and right hand side. However, it will usually be no coincidence that such a common subterm occurs in both sides. One of the motivating examples is a rule typically used in a description of the operational semantics of the while statement:

ev-stat(while(E, S), ρ) $\rightarrow$ ev-stat(while(E, S), ev-stat(S, ρ))
when ev-exp(E, ρ) $\rightarrow$ true

The Common Subterms rule identifies "while(E,S)" as a subterm common to both rule sides, and therefore establishes a link between the function symbol, i.e., the "while" symbol, occurring in this common subterm.

6.4 Defining the Origin Function

6.4.1 Preliminaries

In order to make the notion of relations between subterms more precise, we need *occurrences* (paths). An occurrence is either equal to [ ] for denoting the entire term or to a sequence of integers $[n_1, \ldots, n_m]$ ($m \geq 1$) representing the access path to the subterm. E.g., occurrence [1, 2] denotes the second son of the first son of the root, i.e., for term $f(g(a,b),c)$ it denotes subterm b. The subterm in t at occurrence u is written t/u. Occurrences are concatenated by the (associative) $\cdot$ operator. If u, v, w are occurrences and $u = v \cdot w$, then v is *above* u, written $v \preceq u$. Also, if $w \neq []$ then we write $v \prec u$. If neither $v \preceq u$ nor $u \preceq v$ then u and v are *disjoint*, written $u \mid v$. The set of all occurrences in a particular term t is denoted by $\mathcal{O}(t)$. Subsets of these are $\mathcal{O}_{var}(t)$ for all occurrences of variables in t, and $\mathcal{O}_{fun}(t)$ for occurrences of a function (or constant) symbols in t. The number of elements in a set O of occurrences is written $|O|$.

When we wish to identify the redex occurrence, substitution, and rule of a reduction explicitly, we write $t \xrightarrow{u,\sigma}_r t'$ for the one-step rewrite relation, indicating that rule r is applied at occurrence u in term t under substitution σ. For multi-step reductions $t_0 \rightarrow t_1 \rightarrow \cdots \rightarrow t_n$ we also write $t_0 \rightarrow^* t_n$ ($n \geq 0$).

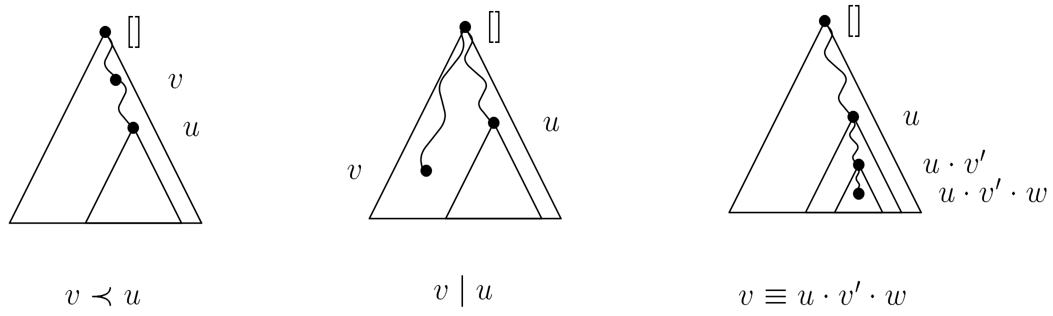


Figure 6.2: Relative positions of v with respect to contractum position u

Finally, we will write $s \subseteq t$ if s is a subterm of t. Moreover, we require that for each rewrite rule $r : \alpha \to \beta$, every variable occurring in β also occurs in α.

6.4.2 Primary Origins

Let $t \xrightarrow{u,\sigma}_r t'$, where r is a rule $\alpha \to \beta$, be a single reduction step. With each step we associate a function *prim-step* : $\mathcal{O}(t') \to \mathcal{P}(\mathcal{O}(t))$ mapping occurrences in t' to sets of occurrences in t. Let $v \in \mathcal{O}(t')$. We define *prim-step*(v) by distinguishing the following cases (see also Figure 6.2):

- (Context)

  If $v \prec u$ or $v \mid u$ then *prim-step*$(v) = \{v\}$;

- (Common Variables)

  If $v \equiv u \cdot v' \cdot w$ with $v' \in \mathcal{O}_{var}(\beta)$ the occurrence of some variable X in the right-hand side β of r, and $w \in \mathcal{O}(X^\sigma)$ an occurrence in the instantiation of that variable, then

  $$\textit{prim-step}(v) = \{u \cdot v'' \cdot w \mid v'' \in \mathcal{O}_{var}(\alpha),\ \alpha/v'' \equiv X\}$$

  (Note that $v'' \in \mathcal{O}_{var}(\alpha)$ is an occurrence of X in the left-hand side α of r).

- (Otherwise)

  For all other cases *prim-step*$(v) = \emptyset$.

This function *prim-step* covers single reduction steps. It is generalized to a function *prim-org** for multi-step reductions $t_0 \to t_1 \to \cdots \to t_n$ $(n \geq 0)$ by considering the origin functions for the individual steps. We will denote the i-th reduction step as A_i : $t_{i-1} \to t_i$ $(0 < i \leq n)$. The associated origin function will be written *prim-step*$_{A_i}$. We can then define the origin relation for the first j steps as *prim-org*^{j}, where *prim-org*^{j} : $\mathcal{O}(t_j) \to \mathcal{P}(\mathcal{O}(t_0))$ for $0 \leq j \leq n$, and $v \in \mathcal{O}(t_j)$:

- $j = 0$: *prim-org*$^j(v) = \{v\}$.

- $1 \leq j \leq n$:

$$\textit{prim-org}^j(v) = \{w \mid \exists w' \in \textit{prim-step}_{A_j}(v) : w \in \textit{prim-org}^{j-1}(w')\}$$

Then *prim-org** is equal to *prim-org*^{n} for multi-step reduction $t_0 \to^* t_n$.

Given a multi-step reduction $A : t_0 \to t_n$, with associated function $\textit{prim-org}^*{}_A$ and occurrence $u \in \mathcal{O}(t_n)$, the set $O = \textit{prim-org}^*{}_A(u)$ is called *the origin set* of t_n/u, and the elements of O are called *the origins* of t_n/u. Often it is natural to relax the difference between sets and elements. If no confusion is possible, we might, for example, use "subterm s has an origin" to indicate that the origin set of s is non-empty, and "subterm s has multiple origins" to state that the origin set of s contains more than one occurrence.

The very same definition of the origin function has been translated into an Asf+Sdf specification. The full specification, including a definition of term rewriting, is discussed in Appendix B. The module defining primary origins is Org-Step (Module B.3.3). This Asf+Sdf specification is executable, and has been used to conduct some experiments in order to evaluate the usefulness of the origins established.

6.4.3 Properties

In the following properties, let $A : t_0 \to^* t_n$ ($n \geq 0$) be a multi-step reduction over TRS R. We will not require that t_n is a normal form, so t_n can represent any term occurring in a reduction. We will closely consider the origin function for A, for which we will write $\textit{prim-org}^*{}_A$. We will use an occurrence $v \in \mathcal{O}(t_n)$ in the last term t_n, and its origin $o \in \textit{prim-org}^*{}_A(v) \subseteq \mathcal{O}(t_0)$.

Primary origins are very similar to the better-known *residuals* or *descendants* [HL91], which are used to study the survival of redexes during reductions over so-called *orthogonal* TRSs (left-linear and non-overlapping [Klo92]). We keep track of *all* terms: redexes as well as irreducible subterms. Therefore, we have:

Property 6.1 *Assume R is* orthogonal*. Let $\backslash A$ be Huet and Lévy's residual mapping for reduction A. Then $v \in o\backslash A \Rightarrow o \in \textit{prim-org}^*{}_A(v)$.*

Proof: Direct from the similarities between the definition of *prim-step* (Section 6.4.2) and residual maps [HL91, Definition 2.1]. □

For left-linear TRSs, we can say something about the size of the origin sets:

Property 6.2 *Assume R is* left-linear*. Then for every $v \in \mathcal{O}(t_n)$ we have $0 \leq |\textit{prim-org}^*{}_A(v)| \leq 1$.*

Proof: The Context case and Otherwise case cannot introduce origin sets containing more than one origin. The Common Variables case can only yield more than one occurrence if there is a rule $\alpha \to \beta$ using a variable X in β for which there are at least two occurrences $v', v'' \in \mathcal{O}(\alpha)$ with $v' \not\equiv v''$ and $X \equiv \alpha/v' \equiv \alpha/v''$. By left-linearity this cannot happen. □

For arbitrary TRSs, we can only say that the number of elements in the sets is smaller than the number of nodes in the initial term:

Property 6.3 *For every $v \in \mathcal{O}(t_n)$ we have $0 \leq |\textit{prim-org}^*{}_A(v)| \leq |\mathcal{O}(t_0)|$.*

Proof: Trivial, as the co-domain of $\textit{prim-org}^*{}_A$ is $\mathcal{P}(\mathcal{O}(t_0))$, the elements of which are sets containing at most $|\mathcal{O}(t_0)|$ occurrences. □

Concerning the subterms that are related, if a subterm $s_n \subseteq t_n$ has a subterm $s_0 \subseteq t_0$ as one of its origins, then s_0 can be rewritten to s_n, in zero (syntactic equivalence) or more steps.

Property 6.4 *For every $o \in \textit{prim-org}^*{}_A(v)$ we have $t_0/o \rightarrow^* t_n/v$.*

Proof: By induction over the length n of reduction A.

- If $n = 0$, then $\textit{prim-org}^*{}_A(v) = \{v\}$ for every $v \in \mathcal{O}(t_n)$. So $v \equiv o$ and $t_0 \equiv t_n$, hence $t_0/o \rightarrow^* t_n/v$.
- Now assume $n > 0$, and consider reduction $A : t \rightarrow^* t_{n-1} \rightarrow t_n$.

  We will prove that for every $v_n \in \mathcal{O}(t_n)$ and $v_{n-1} \in \textit{prim-step}_{A_n}(t_n)$ we have either $t_{n-1}/v_{n-1} \rightarrow t_n/v_n$ or $t_{n-1}/v_{n-1} \equiv t_n/v_n$.

  We will do so by considering the rule $r : \alpha \rightarrow \beta$ applied at step $t_{n-1} \rightarrow t_n$, and the position u of the redex in t_{n-1}.

  We will distinguish three cases, depending on how this redex position u is related to v_n:

  – If $v_n \mid u$ then $\textit{prim-step}(v_n) = \{v_n\}$ (Context case). As $v_n \mid u$, v_n is independent of this reduction step, and $t_{n-1}/v_n \equiv t_n/v$

  – If $v_n \prec u$, then again $\textit{prim-step}(v_n) = \{v_n\}$ (The Context case again). In this case, v_n is right above the reduction position u. But that means $t_{n-1}/v_n \rightarrow t_n/v_n$.

  – If $u \preceq v_n$, then we are within the contractum, for which only the Common Variables case is non-empty. Therefore, assume $v_n \equiv u \cdot v' \cdot w$, where $v' \in \mathcal{O}_{var}(\beta)$ denotes a variable X occurring in the right-hand side. Pick an arbitrary $v'' \in \mathcal{O}_{var}(\alpha)$ such that $\alpha/v'' \equiv X$ (there must be at least one). Then $u \cdot v'' \cdot w \in \textit{prim-step}(v)$, which denotes a syntactically equivalent term in the instantiation of X. Hence, $t_{n-1}/u \cdot v'' \cdot w \equiv t_n/u \cdot v' \cdot w$.

□

A careful inspection of the above proof leads to the following property. Let us use $\textit{top}(t)$ to denote the top symbol f when $t \equiv f(s_1, \cdots s_k)$, $(k \geq 0)$, for terms $s_1, \cdots, s_k$.

Property 6.5 *For every $o \in \textit{prim-org}^*{}_A(v)$ we have $\textit{top}(t_n/v) = \textit{top}(t_0/o)$.*

Proof: Similar to the proof of Property 6.4. For the induction step, the cases $v \mid u$ and $u \preceq v$ are trivial, as the terms related by the single origin step, t_{n-1}/v and t_n/v, and $t_{n-1}/u \cdot v'' \cdot w$ and $t_n/u \cdot v \cdot w$ respectively, are syntactically equal and therefore have the same top-symbol. For the $v \prec u$ case, the terms reduce to each other, $t_{n-1}/v \rightarrow t_n/v$, but the reduction takes place strictly below v, so the top-symbols of t_{n-1}/v and t_n/v remain the same. □

We can rephrase this to identify origins that are always empty:

Property 6.6 *$prim\text{-}org^*{}_A(v) = \emptyset$ if $top(t_n/v)$ is not used in t_0.*

Proof: Follows directly from Property 6.5. □

In other words, so-called *created* function symbols, i.e., those introduced during rewriting and not yet part of the initial term always obtain the empty origin (that is, for the primary origins).

Moreover, in practice Property 6.4 will be slightly stronger: Often we will have $t_0/o \equiv t_n/v$ rather than the weaker $t_0/o \rightarrow^* t_n/v$. In particular, this is the case for subterms which are themselves in normal form already, and which are simply passed as variable instantiations from the initial term to the result term.

6.4.4 Secondary Origins

We now extend the definition of a primary origin to that of a secondary origin, which also includes the redex-contractum and common subterms case.

Proceeding in the same spirit, with reduction step $t \xrightarrow{u,\sigma}_r t'$, where r is a rule $\alpha \rightarrow \beta$, we will associate three more functions with each step, *comm-sub-step*, *redex-contr-step*, and *sec-step*, all mapping occurrences from $\mathcal{O}(t')$ to sets of occurrences from $\mathcal{P}(\mathcal{O}(t))$.

For $v \in \mathcal{O}(t')$, we define *comm-sub-step*(v) as follows:

- (Common Subterms)

  If $v \equiv u \cdot v'$ with $v' \in \mathcal{O}_{fun}(\beta)$ the occurrence of a function symbol (or constant) in the right-hand side β of r, then

  $$comm\text{-}sub\text{-}step(v) = \{u \cdot v'' \mid \alpha/v'' \equiv \beta/v'\}$$

  Note that common subterms are extracted from the *uninstantiated* sides α and β. In all other cases *comm-sub-step*$(v) = \emptyset$.

- (Redex-Contractum)

  $$redex\text{-}contr\text{-}step(v) = \begin{cases} \{v\} & \text{if } v = u \\ \emptyset & \text{otherwise} \end{cases}$$

- (Secondary Origins)

  $$sec\text{-}step(v) = prim\text{-}step(v) \cup comm\text{-}sub\text{-}step(v) \cup redex\text{-}contr\text{-}step(v)$$

The function *sec-step* can be simply extended to *sec-org**, similar to the generalization of *prim-step* to *prim-org**.

Again, Appendix B.3.3 gives an ASF+SDF definition of this *sec-step*.

6.4.5 Properties

Again, let $A : t_0 \rightarrow^* t_n$, $(n \geq 0)$, be a multi-step reduction in a TRS R. Moreover, let $o \in \mathcal{O}(t_0)$ and $v \in \mathcal{O}(t_n)$, and let $\mathit{sec\text{-}org}^*{}_A$ be the secondary origin function associated with A.

Because of the Redex-Contractum case, we need to consider the right-hand side β of a rule $r : \alpha \rightarrow \beta$ a little bit closer. Rule r is said to be a *variable-collapse* rule if β consists of a single variable. Rule r is said to be a *term-collapse* rule if $\beta \subseteq \alpha$, i.e., if the entire right-hand side is contained in the left-hand side. Note that a variable-collapse rule is a special case of a term collapse rule. A TRS without term-collapse rules is called *non-term-collapsing*.

Moreover, because of the Common Subterms case, we need to classify rules according to their left-hand sides. Recall that a term is *linear* if it does not contain multiple occurrences of the same variable. Let us refer to a term as being *term-linear* if it does not contain multiple occurrences of the same *subterm*. A TRS with only rules whose left-hand side is term-linear is called *left-term-linear*.

Using these definitions, we can characterize unique origins:

Property 6.7 *Assume R is non-term-collapsing and left-term-linear. Then for every $v \in \mathcal{O}(t_n)$ we have $0 \leq |\mathit{sec\text{-}org}^*{}_A(v)| \leq 1$.*

Proof: First, as R is non-term-collapsing, the functions *prim-step*, *comm-sub-step*, and *redex-contr-step* are exclusive, i.e., at most one of them is non-empty. Second, each of these three yields at most one origin: *prim-step* because of Property 6.2, *comm-sub-step* since R is left-term-linear, and *redex-contr-step* as it by definition yields either $\emptyset$ or a set of one element. $\square$

Property 6.4 remains valid for secondary origins:

Property 6.8 *For every $o \in \mathit{sec\text{-}org}^*{}_A(v)$ we have $t_0/o \rightarrow^* t_n/v$.*

Proof: Along the same lines as the proof of Property 6.4. For the $u \preceq v$ case, we have to make a further distinction for the Redex-Contractum case (for which we obviously have $t_{n-1}/v \rightarrow t_n/v$) and the Common Subterms case (for which we have $t_{n-1}/u \cdot v'' \equiv t_n/u \cdot v'$). $\square$

Finally, we will characterize when origins will be empty. To that end, let us study when a subterm s with top-symbol f is *not* empty:

- The Context or Common Variables case (see also Property 6.6) is applicable when a copy of s is passed from the initial term to the final term; or
- The Redex-Contractum case is used to give f an origin when it occurs as the top-symbol of the right-hand side of some equation; or
- The Redex-Contractum case is used to give f an origin when f is part of the instantiation of a variable, and the right-hand side of the rule consists of just that variable (a collapse-rule); or

- The Common Subterms rule is used for a subterm containing s.

To formalize this we need the following notion:

Definition 6.9 *A TRS R is* top-preserving *in f if every rule $r : \alpha \to \beta$ in R meets the following requirements:*

1. $top(\beta) \neq f$;
2. *r is not a variable-collapse rule;*
3. *If there is an $s \subseteq \beta$ such that $top(s) = f$, then $s \not\subseteq \alpha$.*

Lemma 6.10 *Let $B : t \to t'$ be a single reduction step in a TRS that is top-preserving in f. Let $w' \in \mathcal{O}(t')$ such that $top(t'/w') = f$. Then for every $w \in$ sec-step$_B(w')$ we have $top(t/w) = f$.*

Proof: Consider the three constituents of *sec-step*:

- If w' obtains an origin from *prim-step*, then w must point to the same top-symbol f by Property 6.6.
- w' cannot get an origin from *comm-sub-step*, as requirement 3 guarantees that the Common Subterms case is not applicable.
- Likewise, w' cannot get an origin from *redex-contr-step*, which would only be non-empty for the contractum occurrence. But at that occurrence, the function symbol cannot be f; not as part of an instantiation because of requirement 2; and not as part of the right-hand side because of requirement 1.

□

Property 6.11 *Let f be a function symbol occurring in t_n at occurrence v. sec-org$^*{}_A(v) = \emptyset$ if R is top-preserving in f, and f does not occur in t_0.*

Proof: By induction over the length n of reduction A.

- Basis case, $n = 0$: As f cannot be both part of term t_n and not part of t_0 if $t_n \equiv t_0$ the condition is always false and the property holds.
- Assume $n > 0$, let $A_{n-1} : t_0 \to^* t_{n-1}$, and let $A_n : t_{n-1} \to t_n$. The induction hypothesis is *sec-org*$^*{}_{A_{n-1}}(v_{n-1}) = \emptyset$ for every v_{n-1} with $top(t_{n-1}/v_{n-1}) = f$.

  We have to prove that for every $v_{n-1} \in$ *sec-step*$_{A_n}(v)$ we have $top(t_{n-1}/v_{n-1}) = f$. But this follows directly from Lemma 6.10.

□

When switching to a *sorted* rewrite system, we can weaken requirement 2 of Definition 6.9, no collapse rules, to the restriction that there are no collapse rules for variables with the same output-sort as function symbol f.

Note that this last property again states a fact for *created* symbols, i.e., symbols not occurring in the initial term. It states that a created symbol f will be empty if the TRS is top-preserving in f. In Section 6.7 we will discuss the importance of this property for specifications occurring in practice.

6.5 Conditional Rewrite Rules

Conditional rewrite rules [BK86, Klo92] are used to execute conditional equations. As conditional equations occur very frequently in realistic specifications, and as origin tracking aims at supporting the generation of better tools for realistic specifications, an extension to conditional rewriting is mandatory.

6.5.1 Preliminaries

A conditional rewrite rule takes the form

$$s_1 \square t_1 \wedge \cdots \wedge s_n \square t_n \Rightarrow s_0 \rightarrow t_0$$

with $n \geq 0$, and s_i, t_i $(0 \leq i \leq n)$ term. We will consider two possibilities for $s \square t$, namely $s \downarrow_! t$ denoting that s and t reduce to the same normal form, and $s := t$ denoting that the normal form of t matches the pattern s. Conditions using $\downarrow_!$ are called *join*-conditions; those using $:=$ *match*-conditions.

We will impose the following restrictions on variables (see also [Wal91, p.16], [Hen91, p.36]). Let $vars(t)$ be the set of variables used in term t. For a conditional rewrite rule, inductively define V_i $(0 \leq i \leq n)$:

- $V_0 = vars(s_0)$;
- $V_{i+1} = vars(s_{i+1}) \cup V_i$

Now for each condition $s_j \square t_j$ $(1 \leq j \leq n)$ the following restrictions apply:

- If $s_j \square t_j$ is a *join*-condition, then $vars(s_j) \subseteq V_{j-1}$ and $vars(t_j) \subseteq V_{j-1}$. In other words, a join condition should not introduce new variables.
- If $s_j \square t_j$ is an *match*-condition, then $vars(t_j) \subseteq V_{j-1}$, i.e., t_j should not introduce new variables. Moreover, $vars(s_j) \cap V_{j-1} = \emptyset$, i.e., the left-hand side should only contain unused variables.

Finally, we require $vars(t_0) \subseteq V_n$, i.e., the right-hand side should only use variables introduced earlier in conditions or in the left-hand side.

A small example of two conditional rewrite rules is given in Figure 6.3. The first two equations are match-conditions, used to unpack the elements P_1 and P_2, which are pairs consisting of a key and an information field. The third condition uses two variables assigned in the first and the second condition in order to decide whether to insert element P_1 before or after P_2. Although not shown, the right-hand sides of both conclusions are allowed to use variables $Key_{\{1,2\}}$ and $Pair_{\{1,2\}}$ as well.

6.5.2 Conditional Origins

As for the unconditional case, origins are defined by focusing on a single reduction step, which now can include sub-reductions needed to check the conditions. If we

$$\begin{array}{rcll}
\text{pair}(Key_1, Info_1) & := & P_1 & \wedge \\
\text{pair}(Key_2, Info_2) & := & P_2 & \wedge \\
\text{greater}(Key_1, Key_2) & \downarrow_! & \text{true} & \\
\Rightarrow & & & \\
\multicolumn{4}{l}{\text{insert}(P_1, \text{cons}(P_2, L)) \rightarrow \text{cons}(P_2, \text{insert}(P_1, L))}
\end{array}$$

$$\begin{array}{rcll}
\text{pair}(Key_1, Info_1) & := & P_1 & \wedge \\
\text{pair}(Key_2, Info_2) & := & P_2 & \wedge \\
\text{less-eq}(Key_1, Key_2) & \downarrow_! & \text{true} & \\
\Rightarrow & & & \\
\multicolumn{4}{l}{\text{insert}(P_1, \text{cons}(P_2, L)) \rightarrow \text{cons}(P_1, \text{cons}(P_2, L))}
\end{array}$$

Figure 6.3: Conditional rules inserting elements in increasing order in a list.

are only interested in origins between the normal form and the initial term, and if we only have join conditions, these sub-reductions are irrelevant for the origins to be computed; the definition of the previous section directly applies.

If we admit match-conditions as well, we have to take sub-reductions into account. For example, assume the right-hand side uses a variable X that does not occur in the left-hand side, but which is assigned a value by one of the match-conditions $s_i := t_i$. Then we need to compute origins involved in the normalization of t_i.

The origins for the sub-reduction $t_i \rightarrow^* t_i'$ itself can be computed just as if it were a normal reduction. This gives an origin relation between t_i' and t_i. In order to relate it to the left-hand side s_0, Common Subterms and Common Variables relations between s_0 and the initial term t_i of the sub-reduction can be established.

The last step consists of linking the right-hand side t_0 to the normalized condition side t_i'. This link is made via the introducing condition side s_i. The link between t_0 and s_i can again follow the Common Subterms and Common Variables relations. As s_i and t_i' are syntactically equal under the matching substitution needed to check condition $s_i := t_i$, these can be related by an identity map.

The full sequence of relations is illustrated in Figure 6.4. Starting in the right-hand side t_0, the dotted line can be followed to condition side s_i (finding Common Variable X). Continuing there, we reach the identical position for the instantiated X in the normalized side t_i'. From here we follow normal origins to the term t_i starting the sub-reduction. From that initial term we finally reach left-hand side s_0 (finding Common Variable Y).

This picture can be extended to an arbitrary number of match-conditions, which mainly involves extra administration indicating where each variable is introduced (either in the left-hand side or in a match-condition). Moreover, we can use the same techniques to compute origins for sub-reductions occurring in join conditions. This makes it possible to compute origins for arbitrary terms occurring in (sub-reductions of) the rewrite process, which, e.g., is necessary for applications such as animation.

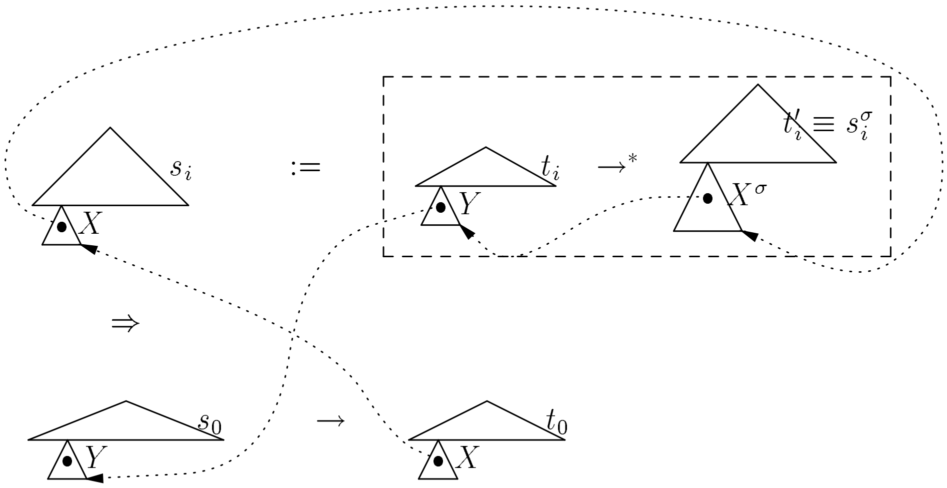


Figure 6.4: Origin relations for a matching condition

This definition of origin tracking for conditional rewrite rules is formalized in [DKT93]. A large set of so-called *relate*-clauses is constructed. Elementary functions generate the proper *relate*-clauses between the left-hand side and reduction-starting conditions, between conditions themselves, and between the right-hand side and the variable-introducing conditions. The transitive closure of these *relate*-clauses defines the full origin relation.

6.6 Implementation

A prototype implementation of origin tracking for conditional term rewriting has been incorporated in the ASF+SDF Meta-environment. If origin tracking is enabled, origin information is maintained during rewriting. This information can either be accessed during rewriting (for animation or source-level debuggers) or after the reduction has finished (error pin-pointing applications).

Several measures have been taken to minimize time and space overhead. Origins are represented by sets of pointers. During rewriting, these sets are maintained as annotations of the trees representing terms. Reducing one term to another involves both a replacement of the redex by the contractum, and a computation of the new annotations for the contractum. Note that the annotations outside the contractum, the context, can be left unaltered.

Computation of annotations for common variables is eased by the use of *sharing*, adopted by the underlying reduction machine of ASF+SDF. For example, if a variable X occurs twice in the right-hand side of a rule, the instantiated contractum will not be built from two *copies* of X, but will contain — as long as this is safe to do — two pointers to the old tree instead. For left-linear rules, this sharing behavior is exactly consistent with the origins to be established. This implies that no annotations in the instantiations of variables need to be updated. In other words: the implementation for the Common Variables case comes — for most rewrite rules — for free.

In order to minimize the time lost in origin computations during rewriting, the exact common variables and common subterm relations are computed in advance ("compile time") for each rule. Although storing this information involves some space overhead, it significantly reduces the origin computations, in particular for rules that are often applied, and contain complicated patterns in conditions and rule sides.

These measurements are sufficient to bring the loss of efficiency to an acceptable level. So far, it seems that using origin tracking involves an increase in reduction time with at most a factor 2.

We expect to be able to reduce this overhead further in the near future. For instance, if one is only interested in origins for the normal form (error handling applications), there is no need to maintain origins for join conditions. The current implementation does not allow to disable such computations. Moreover, it can be the case that we are only interested in origins for particular sorts, for instance only for sorts used in error messages. This may significantly reduce the total amount of administration needed. Again, the current implementation does not yet facilitate such sort selections.

6.7 Applications of Origin Tracking

In this section we describe how origin tracking can be used to obtain a prototype version for a type checker, animator, or source-level debugger, given a specification of the language's static or dynamic semantics or of its translation to target code respectively.

6.7.1 Improving Error Messages

Error handling is the most straight-forward application of origin tracking. A language developer describes the static semantics as a mapping from the language's abstract syntax to some domain of error values. Term rewriting is used to execute this specification, which gives a basic tool that can produce a list of error messages indicating *what* is wrong in a program. If term rewriting with origin tracking is used, the error messages are annotated with sets of occurrences in the source program, indicating *where* the errors were made. An *error reporter* can use these sets of occurrences to high-light or color the relevant program parts for each error message.

As an example, consider Figure 6.5, showing an error reporting application as generated for the language Clax (a Pascal derivative), taken from [DT92]. In the large window, a Clax program is shown, for which the programmer has invoked a type check. The four error messages for this program are shown in the small window. For each message, the Clax programmer can ask "Show Origin", in order to detect the source of his error. In the figure, he has done so for the error `multiply-defined-label step`. This caused the two relevant occurrences of `step` in the source program to be high-lighted. Note that not *all* occurrences of the identifier `step` are marked, only those related to this particular error message.

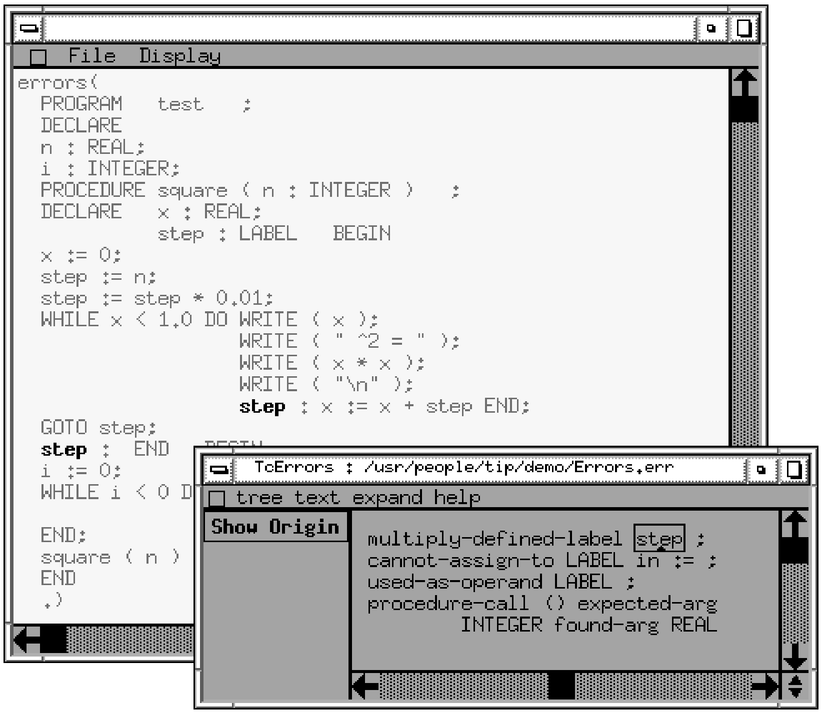


Figure 6.5: Example of a generated environment using origin tracking.

Using origins this way naturally raises the questions whether the origin annotations obtained from a typical static semantic definition are (a) non-empty and (b) if non-empty, are they of good quality. Unfortunately, when using secondary origin tracking, resulting messages far too often have the empty set as their origin. When they are non-empty, however, the occurrences are sufficiently detailed.

To see why origins for error messages can easily be empty, consider Property 6.11. Consider an arbitrary error function symbol e of output-sort Error as used in the Pascal specification. First of all note that an error symbol e will obviously not yet occur in the initial Pascal program. Secondly, it is easy to see that the Pascal specification is indeed *top-preserving in* e (Definition 6.9):

- In general, variable collapse rules are unlikely to occur (requirement 2); and indeed, in the Pascal specification they do not occur for sort Error.

- Similarly, equations having a common subterm concerning an error message are unlikely (requirement 1), and they do not occur in Pascal.

- The first requirement, no right-hand side should have e as top-symbol, could be easily met by a type checker.

  For the Pascal specification of Chapter 4, however, it does not hold. Every error message e is introduced by a special function "add-error", which takes e and a State argument, and produces a new State. Thus, e is never introduced as the top-symbol of a right-hand side.

Thus, the origins for all error message symbols occurring in Pascal are empty. In practice, this situation is not as bad as it seems, as about 40% of the error symbols has an argument to which meaningful origins can be given (as for the identifier `step` in Figure 6.5).

This leaves us with empty origins for the remaining 60%. In the next chapter, we will propose an extension of primary origin tracking which establishes significantly more non-empty origins, which solves this problem. A second solution is given by Dinesh [Din93], who proposes a different view (called *tokenization*) on abstract-syntax trees, which has the effect that the Common Subterms and Variables cases are far more often applicable. His techniques were used for the Clax example of Figure 6.5.

6.7.2 Visualizing Program Execution

The application of origin tracking to program *animation*, i.e., to the visualization of an execution of a program, is discussed by Tip [Tip93]. The point of departure is a specification of the operational semantics of a programming language that can be executed using (conditional) term rewriting. In this case, one is not interested in origins from the normal form to the initial term (the program to be executed), but in origins from terms occurring *during* rewriting back to the initial term. Particularly interesting are intermediate terms representing certain events taking place during execution. For instance, every time the redex matches a pattern like "eval-stat(S, Env)", a statement is executed. If such a term is contracted, the origin of the first argument exactly indicates *which* statement is currently being executed. Likewise, redex-patterns can be given which correspond to expression evaluation, procedure entry, and so on.

An animator tool asks the language designer to give a set of patterns of interest, and to indicate for each pattern which argument position contains the relevant program construct. Next, the animator tool executes the specification using rewriting. Whenever the redex matches one of the patterns, the origins of the indicated subterm are used to obtain a set of occurrences in the program. These occurrences are visualized using, e.g., high-lighting or coloring, and program execution resumes, leading to the next program piece to be visualized, etc.

First of all, note that the origins to be maintained for animation performed this way are mainly origins for terms that represent program pieces, e.g., statements, expressions, and so on. These program pieces literally recur in the initial term, so the Common Variables and Context case can easily provide satisfactory origins.

Secondly, observe that this form of animation does not show any *values* of, e.g., expressions, variables, parameters, etc. Assume that certain values are computed, and that the terms representing these values have non-empty origins. Then this value could be shown in the original program at the position indicated by the origin. So far, no experience exists with the use of origin tracking to obtain such animation facilities. Further research should indicate whether this approach is feasible.

6.7.3 Source-Level Debugging

Finally, we will discuss how origin tracking can be used by a generated debugger. We will focus on *breakpoints*, which can be used to stop execution in order to inspect, e.g., the values of variables. First, we will explain how a debugger can be derived from the specification of the operational semantics of the programming language; next we will discuss the compiled case. As for animation, the debugger tool needs to know which patterns correspond to interesting events, such as statement execution, procedure entry, expression evaluation and so on.

A first possibility are so-called *position-dependent* breakpoints, which stop the execution whenever a certain position in the program (e.g., a particular statement) is reached. The debugging tool asks the programmer to indicate the position in the program where the breakpoint is to be set. The debugging tool computes the occurrence O of that position with respect to the initial term of the program execution. Now term rewriting starts, but stops as soon as the redex matches one of the specified patterns, and — most importantly — the origin of the relevant subterm of the pattern contains the breakpoint occurrence O.

A second possibility are position *independent* breakpoints, stopping execution whenever, for instance, a particular variable is assigned or referenced. Again the programmer indicates which variable he is interested in, for which the debugger tool computes the corresponding occurrence O in the initial term. Execution is stopped whenever the look-up or update function operating on the symbol table is invoked with a symbol argument whose origin set contains the occurrence O.

As for animation, most origins correspond to terms representing parts of the initial program. Therefore, the Common Variables and Context cases are sufficient, when deriving debuggers on basis of a language interpreter.

More challenging is debugging in a compiler context, where the program is translated to some target language. A *source-level* debugger allows the programmer to set breakpoints and inspect the state in terms of the source language, rather than in terms of the target language. Using origin tracking to obtain source-level debuggers proceeds basically in a manner similar to the uncompiled, interpreted case. A first important difference is that the patterns of interest have to be indicated for the evaluation function over the *target* language. Secondly, it is not obvious anymore that the arguments of the target evaluation function, which will be, e.g., machine instructions, will have the proper origins to source-level statements (or to the patterns of interest). As for the problems discussed for error handling (Section 6.7.1), these machine instructions will often have empty origins, even when secondary origin tracking is used. Again the need for an origin function establishing more relations shows up, the subject which will be addressed in the next chapter.

6.8 Concluding Remarks

Origin tracking is a technique to maintain relations between the result of a computation and the initial data. In the framework of (conditional) term rewriting, origin tracking indicates how subterms of terms occurring during rewriting *originate* from subterms of the initial term.

This chapter discusses and analyzes a particular origin relation based on residual maps. Origins are established for the Contexts, Common Variables, Common Subterms, and the Redex Contractum cases. Several properties are formulated, from which it can be concluded that (i) origins for subterms that literally occur in the initial term are dealt with satisfactorily, and (ii) that origins for function symbols not occurring in the initial term can easily have an empty origin.

Origin tracking can be and has been applied to the generation of tools from language specifications. In particular, it proved a valuable technique when using (i) a static semantic definition to obtain a type checker with error pinpointing facilities; (ii) an operational semantics to obtain an execution animator; and (iii) a translation specification and operational semantics to obtain a source-level debugger.

In particular for the generation of type checkers and source-level debuggers, it can be problematic that new function symbols, representing, e.g., error messages or assembly instructions, easily have empty origins. If this problem can be solved, origin tracking can be a very successful technique for the three application areas mentioned. One solution to this problem will be discussed in full detail in the next chapter.

Chapter 7

Origin Tracking in Primitive Recursive Schemes

Many specifications occurring in practice are defined using some form of primitive recursion over a given structure. Such specifications are formalized by so-called *Primitive Recursive Schemes* (PRSs). Typical examples are type checkers or translators, which are defined inductively over the abstract syntax of a programming language. The strong use of PRSs in practice makes good origin tracking facilities for such specifications desirable.

The strict format of a PRS is used to come up with a good proposal for *PRS origins*, and to analyze these origins in full detail. As not all specifications meet these requirement exactly, a second proposal, *syntax-directed origins*, is presented. Syntax-directed origins are defined for arbitrary specifications.

7.1 Introduction

Many functions occurring in first-order algebraic specifications are defined by primitive recursion over some structure. This is, in particular, the case for language definitions, where functions characterizing static semantics or compilation are defined by performing a single pass over the syntax, i.e., they are defined by primitive recursion over the abstract syntax. A typical example is the definition of the type check functions for Pascal as discussed in Chapter 4.

Algebraic specifications containing such primitive-recursive functions are formalized in the class of *Primitive Recursive Schemes* (PRSs) [CF82]. A PRS is a plain algebraic specification, of which we have extra knowledge concerning the function symbols used. The symbols are partitioned into a set G of functions which are used to construct, e.g., the abstract syntax of a language, and a set Φ of functions which are defined inductively over these constructor functions. The equations defining the Φ-functions have to meet certain criteria.

As PRSs play such a central role in language definitions, and particularly in language definitions to which origin tracking is applicable (static semantics with error handling, compilation with source-level debugging), it seems worthwhile to investigate whether it is possible to come up with good origins for specifications in the PRS format.

This chapter starts with mentioning related work (Section 7.2), and arguing the need for special origins for PRSs (Section 7.3). Next, the precise framework of a PRS is introduced (Section 7.4). Within this framework, it is easy to understand "what is going on in a specification", and therefore good origins, the *PRS origins*, can be defined for it (Section 7.5). Moreover, these origins can be easily analyzed, and some desirable properties can be proven for them (Section 7.5.3). However, (larger) specifications as occurring in practice seldom meet *all* requirements of a PRS. Therefore, it is discussed what happens if the requirements are dropped, and how the quality of the origins can be "saved" under those circumstances, giving rise to so-called *syntax-directed* origins (Section 7.6). Finally, it is discussed how syntax-directed origins are of use in existing type checking and animation applications (Section 7.7).

7.2 Related Work

The notion of a *program scheme* [Cou90] is a general device to understand control structures: loops, iterations, goto's and so on are translated to functions defined recursively by equations. Primitive Recursive Schemes were introduced by Courcelle and Franchi-Zannettacci in order to understand attribute grammars (AGs). They gave a one-to-one correspondence between PRSs and AGs [CF82]. As an example of an application of this mapping, Van der Meulen has used techniques for incremental attribute evaluation in order to obtain the effect of incremental rewriting [Meu92, Meu94].

One of the aims of our origin tracking technique is to produce, in an algebraic framework, error messages with good location information associated with it. In AGs, error messages typically correspond to attributes of type string. The error position is identified by printing the message close to the text position of the grammar node the attribute belongs to. In the Synthesizer Generator [RT89b] the language describer can give unparsing (pretty printing) rules to indicate where the error message should be printed. If there is no error, the message attribute contains the invisible empty string.

To compare this with our approach, consider an identifier `x` of type string in an arithmetic expression like `-x`, causing a message like `integer instead of string expected`. In an AG approach, this message will be associated with the `-` node, which gives useful information. In our approach, the message can have origins to (1) the position where the type inconsistency was detected which is the `-` node as in AGs, (2) the place where `x` was declared of type `string`, and (3) the actual position where `x` was used erroneously. Typically, the signature for error messages will include a "`_ instead of _ expected`" symbol, and origins are associated with both the function and the two arguments.

Although not intended for this purpose, our technique could be used to enhance error location in AGs as well, provided the attribute evaluation mechanism is in one way or another based on term rewriting.

The effect of the origins proposed here is very similar to the effect of *subject* tracking

sorts:	EXP STAT ...		
functions:	**if** _ **then** _ **else** _ **fi**:	EXP × STAT × STAT	→ STAT
	const:	INT	→ EXP
	_ + _ :	EXP × EXP	→ EXP
	...		

Figure 7.1: Abstract syntax of simple statements and expressions.

sorts:	ASSEMBLY COMMAND LABEL		
functions:	null:		→ ASSEMBLY
	_ @ _ :	COMMAND × ASSEMBLY	→ ASSEMBLY
	cjump:	LABEL	→ COMMAND
	jump:	LABEL	→ COMMAND
	lab:	LABEL	→ COMMAND
	push:	INT	→ COMMAND
	add:		→ COMMAND
	...		
	0,1,2,3:		→ LABEL
	_ ^ _ :	LABEL × LABEL	→ LABEL

Figure 7.2: Part of the abstract syntax for a simple assembly language. Individual COMMANDS are sequenced by the cons-like function "_ @ _". The empty sequence is written "null"

in the implementation of the specification language TYPOL [Kah87, Des88]. TYPOL is a formalism supporting *natural semantics* to define programming languages. A TYPOL description is typically defined inductively over the syntax of the language. During execution, the Typol implementation maintains one global variable, *subject*, which contains the construct currently begin processed. This variable can be used by external tools for such applications as animation and high-lighting of error locations. If the TYPOL specification is not purely inductively defined, the specifier can use the special *with-subject* construct to reset the subject.

7.3 Primary Origins Reconsidered

7.3.1 An Example

The definition for primary origins discussed in the previous chapter only establishes very basic relations. That these relations can be too basic, is illustrated by the example given in Figures 7.1, 7.2 and 7.3. A small specification of a compilation between a mini-programming language and a tiny assembly language is shown (the equations are

functions:

functions:	tr-stat:	STAT × LABEL	→ ASSEMBLY
	tr-exp:	EXP	→ ASSEMBLY
	...		
	_ ; _ :	ASSEMBLY × ASSEMBLY	→ ASSEMBLY
	...		

variables:	E_1, E_2: EXP		N:	INT
	S_1, S_2: STAT		*Alist*:	ASSEMBLY
	L: LABEL		C:	COMMAND

equations:

[1] tr-stat(**if** E **then** S_1 **else** S_2 **fi**, L) =
 tr-exp(E) ; %% condition
 cjump(L) @
 tr-stat(S_2, 1^L) ; %% else part
 jump(3^L) @
 lab(L) @ %% then part
 tr-stat(S_1, 2^L) ;
 lab(3^L) @
 null

[2] tr-exp(**const**(N)) = push(N) @ null

[3] tr-exp(E_1+E_2) = tr-exp(E_1) ; tr-exp(E_2) ; add @ null

[4] null ; *Alist* = *Alist*
[5] (C @ *Alist*) ; *Alist'* = C @ (*Alist* ; *Alist'*)
...

Figure 7.3: Example specification of a simple translation. Equation [1] defines the translation of an if-statement. Equations [2] and [3] specify the compilation of expressions. Equations [4] and [5] deal with the "_ ; _" operator used to append of two lists of assembly commands.

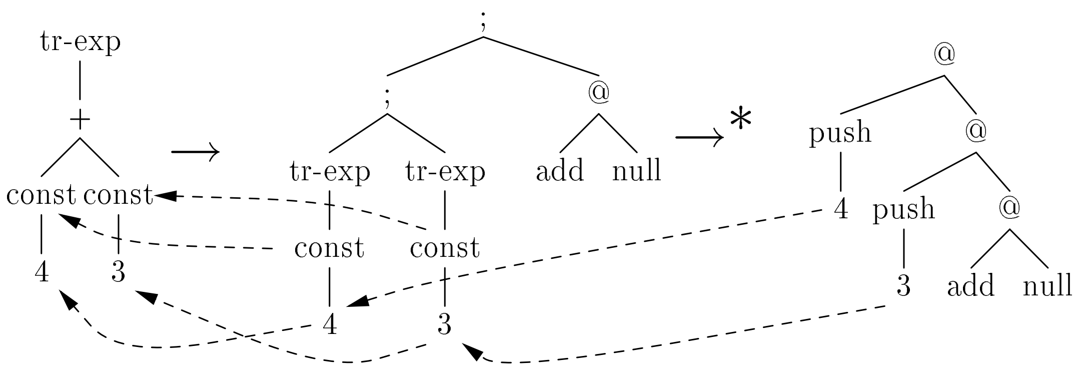


Figure 7.4: Part of a reduction performing the translation of an expression. The dashed lines indicate primary origin relations.

based on specifications discussed by Broy [Bro92] and Van der Meulen [Meu88]).

Signatures for the abstract syntax of the source and target language are given in Figures 7.1 and 7.2. The equations describing the actual translation are shown in Figure 7.3. For the time being, we can ignore the underlining and bold face fonts used for the symbols (see Section 7.4).

An example reduction over this specification is shown in Figure 7.4. The expression "const(4) + const(3)" is translated to "push(4) @ push(3) @ add @ null" using the function "tr-exp". The dashed lines in the figure indicate the primary origin relations established for this reduction. In the first rewrite step, equation [3] is applied. Since variables E_1 and E_2 occur both in its left- and right-hand side, origin relations are established between their instantiations, i.e., between the occurrences of "const(4)" and "const(3)" respectively. In the remaining rewrite steps, in particular those where equation [2] is applied, only the origins for the constants "3" and "4" survive, as indicated by the dashed lines in the figure.

7.3.2 Limitations

The primary origins established for this small compilation example contain surprisingly little information: Only the straightforward copying of the "3" and "4" symbols could be tracked! No origins were established for the "add" or "push" symbols, even though the "+" and "const" nodes occurring in the initial term seem to be good candidates.

This problem is present even stronger in equation [1]. Origins are established for all variables E, S_1, S_2, and, L, but none of the function symbols in the right hand side, e.g. "jump", "cjump", "lab", ..., are given an origin. This is undesirable since these are the symbols that will occur in the resulting normal form.

In general, the problem is that function symbols introduced in the right-hand side of a rule have an origin consisting of the empty set of occurrences. This is unattractive, since it provides very little information on why such a function symbol has been created. We will propose an extension which solves this problem. It will establish good origins for function symbols that are created during rewriting in the setting of a primitive recursive scheme.

7.3.3 Extending Origins

Having noticed the limitations of the existing scheme, one may wonder why it is so difficult to present a suitable extension. Ideally, origins should meet the following requirements:

(A) Origin sets should be small. One would like the origin sets to be as specific as possible. Thus, rather than having an origin which, e.g., states that this assembly instruction originated from the set of all statements in the source program, one would like to know exactly which statement was responsible.

(B) However, origin sets should not be too small. Having the empty set as origin provides little information. Moreover, some applications require that multiple origins be established; e.g., for error handling purposes one would like to have origins both to a declaration of an identifier and its conflicting usage.

(C) Origins should point to deep subtrees. The higher a path points in the initial term, the smaller the information content. For instance, having an origin to the top-node of the initial term will only point out that the normal form somehow has resulted from the initial term. This does not provide very much information.

(D) Origins that point too deep may be misleading. If, again in an error-handling example, an expression "plus(E_1, E_2)" has incompatible argument types, the origin for a message indicating this should point to either the plus or both the top nodes of E_1 and E_2, but not to a very deep subexpression occurring within E_1.

A proposal to extend origin tracking should look for a compromise between these conflicting requirements.

7.4 Primitive Recursive Schemes

A Primitive Recursive Scheme (PRS) is a program scheme formalizing the notion of functions defined inductively (using primitive recursion) over some structure. A typical example of a PRS is a type checker defined inductively over the syntax of a programming language. Definitions of PRSs can be found in [CF82, Meu92]. We follow [Meu92]:

(i) A PRS is a five-tuple $\langle G, \Phi, S, E_\Phi, E_S \rangle$, with G, S signatures, Φ a set of functions, and E_Φ, E_S sets of equations. A PRS corresponds to an algebraic specification $\langle \Sigma, E \rangle$ with signature $\Sigma = G \cup S \cup \Phi$ and equations $E = E_\Phi \cup E_S$;

(ii) All functions in G are free constructors (i.e., there are no equations defined between terms in G);

(iii) The first argument of each ϕ in Φ is of a sort from G, and all other arguments (the *parameters* of ϕ) as well as the output sort come from S;

(iv) E_Φ consists of the Φ-*defining equations*: For each constructor $p : X_1 \times \cdots X_n \to X_0$ in G and each function ϕ in Φ, E_Φ contains exactly one defining equation:

$$\phi(p(x_1, \ldots, x_n), y_1, \ldots, y_m) = \tau \tag{7.1}$$

(v) All equations in E_Φ are strictly decreasing in G: i.e., in equation (7.1), the only G-terms allowed in τ are $x_1, ..., x_n$;

(vi) All equations are left-linear (all variables in the left-hand side of each equation are distinct).

A typical example of a PRS is the specification shown in Figures 7.1, 7.2 and 7.3. The G-signature of this PRS consists of the **boldface** function symbols (introduced in Figure 7.1), defining the grammar of a simple programming language. The Φ-functions of this PRS are <u>underlined</u> functions <u>tr-stat</u> and <u>tr-exp</u> (introduced in Figure 7.3). They are defined using primitive recursion in equations [1], [2], and [3]. These equations satisfy the requirements (iv), (v), (vi) for Φ-defining equations. The S-signature consists of the functions of Figure 7.2 as well as function _ ; _ from Figure 7.3, defining result values as well as auxiliary functions.

In summary, a PRS contains a set Φ of functions defined inductively over the abstract syntax trees of G. Context information is passed downward using the parameters of the Φ-functions. The effect of the Φ-functions is a mapping of G-terms to S-terms. Equations over S may be used to define further computations or simplifications of resulting terms.

7.5 Origins in PRSs

In a PRS, a large G-term (typically the abstract syntax tree of some program) is processed by several Φ-functions; different Φ-functions operate on different G-constructors p, e.g., there will be one Φ-function to translate an if-statement, another one to translate an assignment, and so on. Now consider a Φ-defining equation $\phi(p(x_1, ..., x_n), y_1, ..., y_m) = \tau$. The right-hand side τ is a formula to compute a particular value for some grammar node p. It consists of:

(1) Variables (the x_i occurring in τ) representing sub-constructs of the current node p.

(2) Variables (the y_j occurring in τ) representing global information;

(3) Function symbols initiating computations over sub-constructs of the current node (Φ-functions in τ, with some x_i as first argument)

(4) Function symbols from S indicating how to "synthesize" the result value from the ingredients mentioned above, or how to construct context information to be passed as parameters to the Φ-functions occurring in τ.

This division is reflected in the origins we will define. The Common Variables case (Section 6.4.2) is used to take care of (1) and (2). For case (3) we have a Φ-function ϕ' operating on a sub-construct, and as origin for ϕ' we will use the origin of the first argument of ϕ', that is, of the relevant x_i. Finally, for case (4) the new function symbols are created when working on the $p(x_1, ..., x_n)$ grammar node in the left-hand side; therefore, these new function symbols will obtain as origin the p node in the left-hand side.

These origins caused by Φ-functions traversing the abstract syntax tree are the kernel of the PRS-origins. The remaining origins, concerning equations over S, simply propagate these "Φ-origins", which is achieved by giving all new function symbols in the right-hand side of a rewrite rule from E_S an origin to the top-symbol of the left-hand side. A precise definition is given in Section 7.5.2.

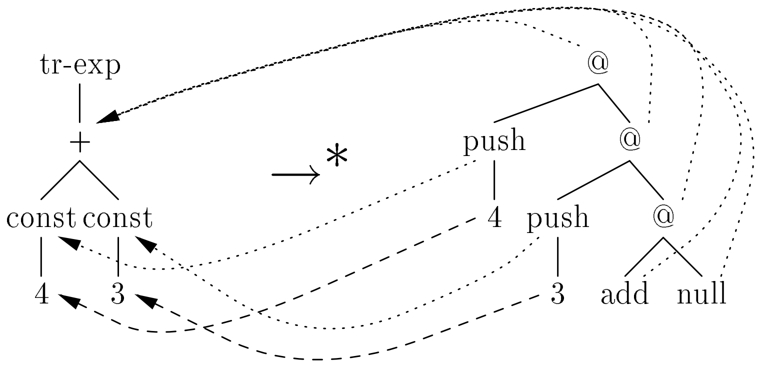


Figure 7.5: Syntax-directed origins for a simple reduction

7.5.1 Example

As an example, reconsider the reduction of "tr-exp(const(4) + const(3))" according to the equations in Figure 7.3. The PRS origins for this reduction are shown in Figure 7.5. The relations between the constants "3" and "4" in the normal form and initial term are established because of Common Variable N when applying equation [2] of Figure 7.3.

The relations between "push" and "const" result from reductions according to Φ-defining equation [2]: the S-function symbol "push" gets the G-argument "const(N)" of Φ-function "tr-exp" as origin. Likewise, S-function symbols "_ ; _", "_ @ _", "add" and "null" introduced in Φ-defining equation [3] are given an origin to the "_ + _", which is the G-argument of Φ-function "tr-exp".

Finally, equations [4] and [5] are used to eliminate the concatenation of assembly code operator "_ ; _". New functions introduced in these S-equations receive the top-function symbol of the left-hand side as origin. Since in this case the "_ ; _" operators were introduced by equation [3], these origins point to the "+" function symbol.

7.5.2 Definition

For a PRS $\langle G, \Phi, S, E_\Phi, E_S \rangle$ and term t, we introduce $\mathcal{O}_\Phi(t)$, $\mathcal{O}_G(t)$, and $\mathcal{O}_S(t)$ as the sets of occurrences for function symbols from Φ, G, and S respectively. To define origins for PRSs, we again consider a single reduction step $t \equiv C[\alpha^\sigma] \to C[\beta^\sigma] \equiv t'$. Let u be the occurrence in C of the redex position. The function *prs-org-step* : $\mathcal{O}(t') \to \mathcal{P}(\mathcal{O}(t))$ maps occurrences in t' to sets of occurrences in t. Define *prs-org-step*(v) by taking the "Common Variables" and "Context" cases of *prim-step* (see Section 6.4.2), together with the following cases, where v is the occurrence of a function (or constant) symbol in β:

- (Φ-Functions)

  If $v \equiv u \cdot v'$ with $v' \in \mathcal{O}_\Phi(\beta)$ the occurrence of a Φ-function symbol in the right-hand side β then

$$\textit{prs-org-step}(v) = \textit{prs-org-step}(\ v \cdot [1]\ )$$

In other words, the origin of a Φ-function is equal to the origin of its G-argument.

- (Synthesizers)

  If $v \equiv u \cdot v'$ with $v' \in \mathcal{O}_S(\beta)$ the occurrence of a function symbol from S in the right-hand side β, and $r \in E_\Phi$ is a Φ-defining equation with left-hand side $\alpha \equiv \phi(p(x_1, ..., x_n), y_1, ..., y_m)$, then

  $$\textit{prs-org-step}(v) = \{u \cdot [1]\}$$

  In other words, the origin is the G-term $p(x_1, ..., x_n)$ as it occurs in the left-hand side.

- (Auxiliary Symbols)

  Finally, if the rule $\alpha \to \beta$ is from E_S, and $v \equiv u \cdot v'$ with $v' \in \mathcal{O}_G(\beta) \cup \mathcal{O}_S(\beta)$ the occurrence of a function symbol from G or S in the right-hand side β, then

  $$\textit{prs-org-step}(v) = \{u\}$$

  In other words, the origin is the top-symbol of the left-hand side.

At first sight the definition of *prs-org-step* for the Φ-Functions case may seem a little dangerous since *prs-org-step* appears at both sides of the equality sign. However, the first argument of a Φ-function must — by definition of a PRS — be a G-term, for which the *prs-org-step* function is directly defined in the remaining cases. When we know that rule r actually is a Φ-defining equation $\phi(p(x_1, ..., x_n), y_1, ..., y_m) = \tau$ we can even make a stronger statement: A Φ-function ϕ' occurring at position v in right-hand side τ must — again by definition of a PRS — have one of the $x_i (1 \leq i \leq n)$ as its first argument. The occurrence of that x_i in the left-hand side is $[1, i]$, so for this case we can define *prs-org-step* alternatively as $\textit{prs-org-step}(v) = \{u \cdot [1, i]\}$.

The function *prs-org-step* can be extended to a function $\textit{prs-org}^*$ covering multi-step reduction similar to the extension of *org-step* to $\textit{org}^*$ (see Section 6.4.2).

7.5.3 Properties

In Section 7.3.3 we mentioned four desirable characteristics (labeled (A), (B), (C), and (D)) for extensions of origins. Concerning the number of elements in the origin sets (requirements (A) and (B)), we observe that origin sets in pure PRSs always contain exactly one element:

Property 7.1 *Let t, t' be terms, $A : t \to^* t'$ a reduction in a PRS, and let $\textit{prs-org}^*_A$ be the origin function for this reduction. For all $v \in \mathcal{O}(t')$ we have $|\textit{prs-org}^*_A(v)| = 1$.*

Proof: This follows from the facts that (1) PRSs are left-linear, (2) none of the various cases for function symbol origins in PRSs overlap, and (3) because every individual case yields exactly one origin. $\square$

This means that requirement (A) to keep the origin sets small is met, as they always consist of exactly one element. However, its counterpart (B), not to make them too small, is only partly met. On the positive side, no empty origins will occur (which contrasts with the primary or secondary origins, see Properties 6.6 and 6.11). On the negative side, situations were multiple origins are desirable (error handling) are treated in an unsatisfactory manner.

PRS origins try to achieve the proper depth (requirements (C) and (D)) by focusing on the G-terms. Created function symbols get an origin to the G-argument of the *closest* surrounding Φ-function. More precisely, assume we are reducing t, by applying a rule $\phi(p(x_1, \ldots, x_n), y_1, \ldots, y_m) = \tau$. Function symbols that will be created when further reducing τ will have an origin to a subterm of one of $p(x_1, \ldots, x_n)$. Existing functions that are passed around in variables $y_{1,\ldots,m}, x_{1,\ldots,n}$ simply maintain their origins.

Property 7.2 *Let t, t' be terms, $t \neq t'$, let $A : t \rightarrow^* t'$ be a reduction in a PRS, and let prs-org$^*_A$ be the origin function for this reduction. Assuming that the top operator of t is a Φ-function, we have: For all $v' \in \mathcal{O}(t')$, let prs-org$^*_A(v') = \{v\}$. Then either:*

- *v is is an occurrence in one of the y_j, i.e., $[j] \preceq v$, with $j \geq 2$, and $t'/v = t/u$; or*
- *v is an occurrence in the G-argument of ϕ, i.e., $[1] \preceq u$.*

Proof: Direct from the definition of *prs-org-step* and the fact that t has a Φ-function as its top node. □

As an example, assume we are type checking or translating an expression using a Φ-function ϕ, expecting an expression as its G-argument, and a single parameter y_1 representing a symbol table S. All function symbols created while reducing $\phi(E, SymT)$ will have an origin to E, and not to a subterm occurring in the symbol table.

7.6 Syntax-Directed Origin Tracking

PRSs are a good format to define origins for, but in practice few specifications will exactly meet all requirements. Here we study how origins still can be established if some of the requirements are relaxed.

7.6.1 Relaxing the PRS Requirements

Some of the five requirements (ii) to (vi) of Section 7.4 are non-restrictive, and others can be relaxed easily:

(ii) Equations over G-terms can be useful, as in

$$\text{repeat}(S, E) = \text{seq}(S, \text{while}(\text{not}(E), S))$$

We can handle such equations by relating each function symbol in the right-hand side to the top of the left-hand side. Hence equations over G-terms are treated as equations over S-terms (the Auxiliary Symbols case).

(iii) The restriction to recursion over the *first* argument is not essential, provided there is some way in which the specifier can indicate over which the recursion runs.

(iv) The fixed left-hand side $\phi(p(x_1, ..., x_n), y_1, ..., y_m)$ of Φ-defining equations can be annoying when writing large specifications. Deeper patterns at the x_i or y_j positions can be allowed without problems (allowing non-trivial matching at these positions).

(v) Right-hand sides of Φ-defining equations are not allowed to contain any function symbols from G. This can be a problem when writing, e.g., an operational semantics of a while loop, where the while constructor will appear in the right-hand side again. For origin tracking purposes, it is possible to link every new G-term to the $p(x_1, ..., x_n)$ node at the left-hand side. Thus, G-symbols introduced in right-hand sides of Φ-defining equations can be treated as S-symbols introduced in such right-hand side (the Synthesizer case).

(vi) Linearity of left-hand sides is not essential. Allowing non-linear patterns causes origins to contain multiple paths, which (Section 7.3.3, (B)) can be useful under certain circumstances.

Relaxing requirements (ii) and (v) can make some origins less precise. For instance, the "not(E)" in the right-hand side of the equation mentioned under (ii) will have an origin to the entire repeat statement. Note however, that the origins still will be non-empty, i.e., they consist of at least one path.

7.6.2 Further Extensions

In the PRS context, Common Subterms as discussed in Section 6.4.4 can be useful in the Auxiliary Symbols case (which also applies to G-terms if (ii) is relaxed). Moreover, if the fixed patterns of left-hand sides of Φ-defining equations are allowed to contain arbitrary patterns (iv), the common subterms case could be useful to find origins for S-symbols (or even G-symbols, if (v) is relaxed) occurring in the right-hand side.

A detailed account of the use of PRSs is given by Van der Meulen [Meu94]. She proposes extensions of PRSs to deal with *conditional* equations as well as with associative *lists*. Syntax-directed origins can easily be extended to deal with these mechanisms as well.

A particularly interesting topic Van der Meulen discusses is the nested or *layered* PRS. A typical example of a layered PRS is a compilation defined by a translation to an intermediate language, followed by a translation to the target language. Both translations will be defined as PRSs. Thus, the S-functions of the first PRS act as G-functions of the second PRS. Syntax-directed origins easily apply to both PRSs.

As an example of a very simple nested PRS, the append function called "_ ; _" for lists of assembly commands occurring in Figure 7.3 is defined by primitive recursion over the two constructors "null" and "_ @ _". Therefore, "_ ; _" can be considered as a Φ-function by the syntax-directed origin scheme. Doing so has an effect on the origins for the "_ @ _" symbol created in the right-hand side of equation [5]. If not regarded as a Φ-defining equation, its origin will be — according to the Auxiliary Symbols case — the top symbol of the left-hand side. If "_ ; _" is considered a Φ-function, then its origin will be — following the Synthesizer Case — the "_ @ _" symbol at the left. Thus, one can influence the origins actually established by indicating which functions are to be considered Φ-functions. In general, the larger the set of Φ-functions is, the more specific the origins will be.

7.7 Concluding Remarks

We have presented an origin function for specifications having a syntax-directed nature, using the notion of a Primitive Recursive Scheme as starting point. The origins have the desirable property that each node in the normal form is linked to exactly one (PRS case) or at least one (general case) subterm in the initial term.

In order to compare the syntax-directed origins with primary origins, consider the specification of the static semantics of Pascal as discussed in Chapter 4. Recall from Section 6.7.1 that the origins for *all* error message symbols were empty, and that for only 40% of the error messages a satisfactory origin could be established for the arguments of the message. By contrast, syntax-directed origin tracking creates non-empty origins for all error messages. Moreover it establishes good origins for all arguments as well, in particular for created symbols representing Pascal types.

Also, we can take a look at the specification of the static and dynamic semantics of Clax (a subset of Pascal) given by Dinesh and Tip [Tip93, Din93]. In order to get the proper origins for his type checker, Dinesh comes up with a clever solution (called *tokenization*) to circumvent the limitations of the primary origin tracking mechanism. Although his idea could be systematized and automated (i.e., made invisible to the specifier), it remains undesirable to adapt specifications to the origin tracking mechanism. His specification is contains a translation which is by nature int he primitive recursive format, although he needs some equations over the G-terms (i.e., he violates requirement (ii)). With the extensions of Section 7.6.1, syntax-directed origins will automatically establish the desired origins.

Tip studies the generation of an animator for Clax from the specification of an evaluation function. He uses *pattern specifications* to determine which statement is currently being executed (see also Section 6.7.2). Although his specification is not a pure PRS (in essence it is defined inductively over the abstract syntax, though), syntax-directed origin tracking could be used to simplify his patterns, and to overcome the problems he mentions concerning expressing evaluation of the repeat statement in terms of the while statement (see Section 7.6.1 and [Tip93, Section 7]).

In general, animation seems to be a natural application of syntax-directed origin tracking; a simple animator is obtained by high-lighting the G-argument whenever a Φ-function is rewritten. However, the evaluation function for programming languages need not be strictly decreasing, since the evaluation may involve loops, goto's or procedure calls. Nevertheless, as the "continuations" used to specify evaluation of these constructs will typically have an origin obtained by the Common Variables case, syntax-directed origins, including the extensions of will often be able to deal with such constructs satisfactory.

One could argue against the idea of defining origins for PRSs because PRSs are a too limited class of specifications. This argument, however, is easily refuted by studying arbitrary language definitions of e.g., static semantics, evaluation, translation, and so on. In a first-order formalism, it is virtually impossible to specify these without some use of primitive recursion.

We intend to extend the implementation as discussed in Section 6.6 as soon as possible with syntax-directed origins. The extension seems to be trivial, especially since much of the PRS machinery has already been implemented as part of the incremental term rewriting machinery of the ASF+SDF Meta-environment [Meu94].

We were, however, able to gain some experience with the use of syntax-directed origins, by extending the ASF+SDF specification of origin tracking given in Appendix B to cover them. In particular, module Org-Step-PRS, Appendix B.3.5, contains an executable definition of origin tracking. Given this specification, we were able to conduct some initial experiments and observed that syntax-directed origins behaved as expected.

Chapter 8

Origin Tracking for Higher-Order Term Rewriting Systems

Algebraic specifications extended with higher-order facilities, have the advantage of being more concise and less deterministic. Recent experiments with the use of higher-order algebraic specifications for the definition of programming languages revealed a need to extend origin tracking to higher-order term rewriting systems. In this chapter we discuss how origin information can be maintained for β reductions and η-expansions, during higher-order rewriting. We give a definition of higher-order origin tracking. The suitability of this definition is illustrated with a small, existing specification.

8.1 Introduction

Many first-order specifications of programming languages are more deterministic than desirable. Typically, a considerable amount of specification effort is invested in encoding explicit program traversals in order to extract information from programs (concerning, e.g., declaration and use of identifiers). Such encodings are uninteresting, and distract readers from the more essential parts of the specification.

Recently, Heering studied the use of *higher-order* specifications as a solution to this over-specification problem [Hee92]. He came up with an example specification of the static semantics of a small language, for which he did not need any program traversal, nor the explicit construction of a program environment (symbol table).

One of his observations was that when turning this higher-order specification into an executable type checker, "a suitable extended origin tracking facility" was mandatory [Hee92, Section 2.2]. This raises the question whether the notion of origin is easily extensible to the higher-order case, the topic we will be studying in this chapter. It turned out that origin tracking in higher-order systems is considerably more complicated. This extra complexity mainly arises because equality in the higher-order case is modulo $\overline{\beta eta}$ conversion, rather than simple syntactic equivalence.

We start this chapter with a summary of the definitions of higher-order rewriting along with a small example. Next, we present primary origins for the higher-order case in Section 8.3, and extensions to secondary origins in Section 8.4. In Section 8.5 we mention related work and draw some conclusions.

Note that this chapter is complementary to the previous chapter, dealing with origins for Primitive Recursive Schemes (PRSs). A PRS is a formalization of a set of functions performing an explicit program traversal. In a first-order setting, it is difficult to avoid such traversals. Therefore, PRSs will occur frequently in first-order specifications, and establishing good origins for them is important. In a higher-order setting, by contrast, explicit traversals and PRSs can be avoided. Their role is taken over by rewrite rules over higher-order terms. Finding good origins for these higher-order rules is the theme of this chapter.

8.2 Higher-Order Term Rewriting

For the definition of Higher-Order Term Rewriting Systems (HRSs), we follow [Wol93, Nip91, OR94]. The main difference with the first-order case is that terms in HRSs are constructed according to the simply-typed λ-calculus [Chu40].

8.2.1 The Simply-Typed λ-Calculus

The set of *type symbols* T consists of *elementary* type symbols from T_0 and of *functional* type symbols $(\alpha \rightarrow \beta)$, where $\alpha, \beta \in T$. We may abbreviate a type $(\alpha_1 \rightarrow (\alpha_2 \rightarrow (\cdots \rightarrow (\alpha_n \rightarrow \beta)\cdots)))$ to $(\alpha_1, \ldots, \alpha_n \rightarrow \beta)$. *Terms* are built using *constants* and *variables*, each of which has an associated type symbol. The type of t is written $\tau(t)$. If x is a variable with $\tau(x) = \alpha$, and t a term with $\tau(t) = \beta$, then the *abstraction* $(\lambda x.t)$ is a term of type $(\alpha \rightarrow \beta)$. If t, t' are terms with $\tau(t) = (\alpha \rightarrow \beta)$ and $\tau(t') = \alpha$, then the *application*[1] $(t\ t')$ is a term of type β. Application is left-associative.

Occurrences in λ-terms are defined as for first-order terms, by representing abstraction as a node with one son and application as a node with two sons. As an example, Figure 8.1 shows all occurrences in the term (add $((\lambda N.N)$ zero) zero). Again, $\mathcal{O}(t)$ is the set of all occurrences in t.

All occurrences of x in $(\lambda x.t)$ are said to be *bound*. Non-bound occurrences are *free*. A term is *closed* if it does not contain free variables, *open* otherwise. Bound variables can be renamed according to the rule of α-conversion. A *replacement* of a term t at occurrence u by subterm s is denoted by $t[u \leftarrow s]$. A *substitution* σ is a mapping from variables to terms. Application of a substitution σ to a term t, written t^σ, has the effect that all free occurrences of variables in the domain of σ are replaced by their associated term. Following the *variable convention* [Bar84], bound variables are renamed if necessary.

Let x be a variable, t_1, t_2 terms, and let substitution $\sigma = \{x \mapsto t_2\}$. Then the term $((\lambda x.t_1)\ t_2)$ is a *β-redex* and can be transformed to t_1^σ by β-reduction. A term without β-redex occurrences is said to be in *β-normal form*. All typed λ-terms have a β-normal form, which is unique up to α-conversion. A β-normal form always has the form

$$(\lambda x_1.(\lambda x_2.\cdots(\lambda x_n.\{(\cdots((H\ t_1)\ t_2)\cdots\ t_m)\})\cdots))$$

[1] We use $@(t, t')$ alternatively, when there is a need to make the application operator explicit, as in Figure 8.1. We also use $t(t')$ in the context of algebraic specification, as in Figure 8.2.

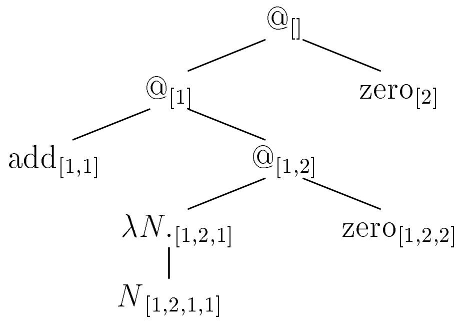


Figure 8.1: Occurrences in the term "(add $((\lambda N.N)$ zero)) zero".

where $x_1, \ldots, x_n$ are variables, $t_1, \ldots, t_m$ terms in β-normal form, H a constant or a variable, $m, n \geq 0$. We will sometimes write this as $\lambda x_1 \cdots x_n.H(t_1, \ldots, t_m)$. In such a term, H is called the *head*, $H(t_1, \ldots, t_m)$ is called the *matrix*, and $\lambda x_1 \cdots x_n$ is called the *binder*.

The rule of η-reduction states that terms of the form $\lambda x.(t\ x)$ can be transformed to just t, provided that x does not occur freely in t. Its counterpart is $\overline{\eta}$-expansion: if a head H of a β-normal form $\lambda x_1 \cdots x_n.H(t_1, \ldots, t_m)$ is of type $(\alpha_1, \ldots, \alpha_{m+k} \to \beta)$ ($k > 0$), then clearly as H expects more arguments, we can add these as extra abstractions. The term above can be $\overline{\eta}$-expanded to $\lambda x_1 \cdots x_n y.H(t_1, \ldots, t_m, y)$, where y is a fresh variable of type α_{m+1}. Every term has a $\overline{\eta}$-normal form.

Let χ be any of $\{\alpha, \beta, \eta, \overline{\eta}\}$. If t can be transformed to t' by performing a χ-reduction at occurrence u, we write this as $t \triangleright_{\chi,u} t'$, or alternatively as $t' \triangleleft_{\chi,u} t$, where we may omit occurrence u. Repeated χ-reduction is written $t \triangleright_{\chi}^{*} t'$. Since $\triangleright_{\alpha}^{*}$ is a symmetric relation, we will sometimes write it as $=_{\alpha}$. The $\beta\overline{\eta}$-normal form of t is indicated by $t\!\downarrow_{\beta\overline{\eta}}$. The relation $t =_{\beta\overline{\eta}} t'$ holds if and only if $t\!\downarrow_{\beta\overline{\eta}} =_{\alpha} t'\!\downarrow_{\beta\overline{\eta}}$.

8.2.2 Higher-Order Rewrite Steps

If p, q are (open) simply-typed λ-terms of the same type and in $\beta\overline{\eta}$-normal form, and if every free variable in q also occurs in p, then $p \to q$ is a (higher-order) rewrite rule. A reduction $t \xrightarrow{u,\sigma}_{r} t'$, where t, t' are closed λ-terms in $\beta\overline{\eta}$-normal form, σ is a substitution, $r : p \to q$ is a rewrite rule, and u is an occurrence in $\mathcal{O}(t)$ denoting the redex position, is possible if:

- The types of the redex and the left-hand side of the rule are the same:

  $\tau(t/u) = \tau(p)$

- The instantiated left-hand side is $\beta\overline{\eta}$-equal to the redex:

  $\{p^{\sigma}\}\!\downarrow_{\beta\overline{\eta}}\ =_{\alpha}\ \{t/u\}\!\downarrow_{\overline{\eta}}$

- Replacement of the redex by the instantiated right-hand side followed by $\beta\overline{\eta}$-normalization yields the result t':

  $\{t[u \leftarrow q^{\sigma}]\}\!\downarrow_{\beta\overline{\eta}}\ =_{\alpha}\ t'$

Notice the variety of $\{\alpha, \beta, \overline{\eta}\}$-conversions involved in the application of one rule. This turns out to have consequences for the definition of origins. Also note that matching the redex against a left-hand side may yield more than one substitution. For origin tracking purposes, however, we are not concerned with finding matches; we assume that in some way it has been decided to apply a rewrite rule under a given substitution (see also Section 8.4.3).

8.2.3 Example

Consider the second-order algebraic specification of a simple type checker shown in Figure 8.2, which was taken from [Hee92]. The objective of this specification is to replace all simple expressions (identifiers, string or natural constants) by a term "tp(τ)", where τ is the type of that simple expression (see equations [1], [2], and [3]). Next, type correct expressions are reduced to their type (equation [4]). Finally, type correct statements are eliminated (equation [5]). The resulting normal form only contains the incorrect statements.

Take the initial term P_1:

```
program( decls( decl(n,natural),  decls( decl(s,string), emptydecls) ),
        stats( assign(s, plus(id(n),id(n))), emptystats )   )
```

It can be reduced according to equation [1] with, e.g., the substitution[2] σ_1:

$$
\begin{array}{lrcl}
\{ & \mathcal{D} & \mapsto & \lambda Decl.\ \text{decls}(Decl, \text{decls}(\text{decl}(\text{s,string}), \text{emptydecls})), \\
 & \mathcal{S} & \mapsto & \lambda Id.\ \text{stats}(\text{assign}(\text{s,plus}(\text{id}(Id),\text{id}(Id))), \text{emptystats}), \\
 & X & \mapsto & \text{n}, \\
 & \tau & \mapsto & \text{natural} \quad \}
\end{array}
$$

Applying this rule replaces occurrences of "n" by "tp(natural)", which results in a term P_2:

```
program( decls( decl(n,natural),  decls( decl(s,string), emptydecls) ),
        stats( assign(s, plus(id( tp(natural) ),
                               id( tp(natural) ))), emptystats )  )
```

Next, equation [1] can be applied again, this time replacing "s" by "tp(string)", yielding a P_3. Finally, equation [4] can be used to replace the "plus" expression by a representation of its type (natural) resulting in P_4, which is the normal form of P_1.

Initially, we are allowed to apply equation [1] on P_1, since under substitution σ_1, the left-hand side of equation [1] produces a new term P_1'', which after two β-reductions (one for $\mathcal{D}$ and one for $\mathcal{S}$) is exactly equal to term P_1.

To construct the result P_2 of this one-step reduction, we first apply σ_1 to the right-hand side of equation [1], producing some term P_2''. Then two more β-reductions

[2] It is necessary to avoid vacuous abstraction of *Decl* in the assignments of $\mathcal{D}$ [Hee92].

transform P_2'' to its β-normal form, which results in the desired P_2. We can summarize this first single-step rewrite as follows:

$$P_1 \lhd_\beta P_1' \lhd_\beta P_1'' \equiv l_1^{\sigma_1} \rightsquigarrow r_1^{\sigma_1} \equiv P_2'' \rhd_\beta P_2' \rhd_\beta P_2$$

where $\rightsquigarrow$ denotes the replacement of the instantiated left-hand side by the instantiated right-hand side, and l_1 and r_1 are the left and right-hand side of equation [1]. Our definition of origins also follows this "flow" where origins between P_2 and P_1 are defined using elementary origin definitions between the pairs P_2 – P_2', P_2' – P_2'', etc.

8.3 Higher-Order Origins

We define origins for higher-order rewriting by (i) indicating how origins are to be established for $\rhd_\alpha$, $\rhd_\beta$, $\rhd_\eta$, and $\rhd_{\overline{\eta}}$ conversion; then (ii) describing how the inverses $\lhd_\beta$ and $\lhd_{\overline{\eta}}$ can be dealt with; and (iii) explaining how origin relations can be set up between the left- and right-hand side of a rewrite rule. In this section we give a very basic definition, which we refer to as *primary, higher-order, origins*. In the next section we discuss various proposals and heuristics to extend these origins.

We use the following notational conventions. For a term t and variable x, we write $\mathcal{O}_{fvars}(t)$ for all free variable occurrences in t, $\mathcal{O}_{fvars(x)}(t)$ for the occurrences of x in t that are free, and $\mathcal{O}_{bfun}(t)$ for the application, abstraction, or constants as well as the bound variable occurrences in t. Moreover, we abbreviate occurrences of a series of n b-branches as $[b^n]$. For example, for a β-normal form $\lambda x_1 \cdots x_n.H(t_1, \ldots, t_m)$, the path to λx_j is $[1^{j-1}]$ $(1 \leq j \leq n)$ and the path to t_i is $[1^n] \cdot [1^{m-i}] \cdot [2]$. The left side of Figure 8.3 shows a term in β normal form, and some path abbreviations.

8.3.1 Conversions

Let t, t' be terms, $u \in \mathcal{O}(t)$, and let χ be any of $\{\alpha, \beta, \eta, \overline{\eta}\}$. Given $t \rhd_{\chi,u} t'$, we define $org(v)$ for $v \in \mathcal{O}(t')$. First, if $v \mid u$ or $v \prec u$ then $org(v) = \{v\}$. Otherwise,

- $\chi = \alpha$:

  α-Conversion does not change the term structure, so we simply have $org(v) = \{v\}$.

- $\chi = \beta$:

  Since t/u is a β-redex, we have $t/u \equiv ((\lambda x.t_1)\ t_2)$. Note that in t/u, the path to t_1 is $[1, 1]$, and to t_2 is $[2]$. Now let $w_1 \in \mathcal{O}(t_1), w_2 \in \mathcal{O}(t_2)$. We distinguish two cases:

  1. $v \equiv u \cdot w_1$: hen $org(v) = \{u \cdot [1, 1] \cdot w_1\}$.
  2. $v \equiv u \cdot w_1 \cdot w_2$ with $t_1/w_1 = x$, and $w_2 \succ [\,]$: then $org(v) = \{u \cdot [2] \cdot w_2\}$. The condition $w_2 \succ [\,]$ avoids overlap with the former case.

sorts: PROG DECLS DECL STAT STATS ID TYPE EXP ...

functions:

program	: DECLS, STATS	$\rightarrow$ PROG
decls	: DECL, DECLS	$\rightarrow$ DECLS
emptydecls	:	$\rightarrow$ DECLS
decl	: ID, TYPE	$\rightarrow$ DECL
natural	:	$\rightarrow$ TYPE
string	:	$\rightarrow$ TYPE
stats	: STAT, STATS	$\rightarrow$ STATS
emptystats	:	$\rightarrow$ STATS
assign	: ID, EXP	$\rightarrow$ STAT
plus	: EXP, EXP	$\rightarrow$ EXP
id	: ID	$\rightarrow$ EXP
nat	: NAT	$\rightarrow$ EXP
str	: STRING	$\rightarrow$ EXP
...		
tp	: TYPE	$\rightarrow$ ID

variables:

$\mathcal{D}$	: DECL $\rightarrow$ DECLS	X	: ID
τ	: TYPE	$\mathcal{S}$	: ID $\rightarrow$ STATS
S	: STATS	N	: NAT
R	: STRING		

equations:

[1] program($\mathcal{D}$(decl(X,τ)), $\mathcal{S}$(X)) = program($\mathcal{D}$(decl(X,τ)), $\mathcal{S}$(tp(τ)))
[2] nat(N) = id(tp(natural))
[3] str(R) = id(tp(string))
[4] plus(id(tp(natural)), id(tp(natural))) = id(tp(natural))
[5] stats(assign(tp(τ), id(tp(τ))), S) = S

Figure 8.2: Part of a static semantics specification

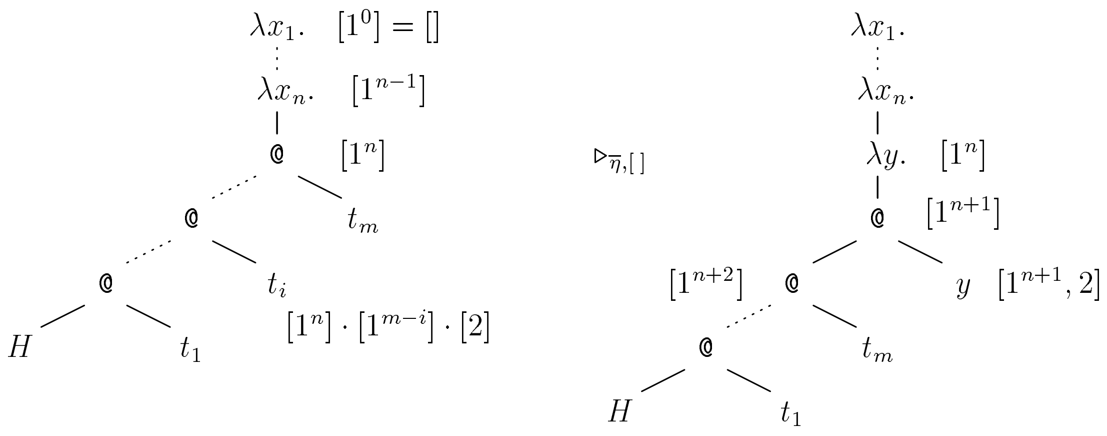


Figure 8.3: $\overline{\eta}$-Expansion.

Thus, origins in the body t_1 "remain the same"; origins for the top node of an instantiated variable have an origin to their corresponding variable position in the body t_1, which is indicated by the dashed lines in Figure 8.4; and origins to non-top nodes of an instantiated variable have an origin to their position in the actual parameter t_2, which is indicated by the dotted lines.

- $\chi = \eta$:

  In η-reduction one λ is eliminated. Since t/u is an η-redex, we can assume $t/u = \lambda x.(t_1\ x)$. Realizing that the path to t_1 is $[1, 1]$, we simply have: $org(u \cdot v') = \{u \cdot [1, 1] \cdot v'\}$.

- $\chi = \overline{\eta}$:

  In η-expansion, an extra λ is added. The origins of the old parts point to the same old parts, while the origin of the new λ is the empty set:

  Since t/u is an $\overline{\eta}$-redex, we have $t/u = \lambda x_1 \cdots x_n.H(t_1, \ldots, t_m)$. We distinguish three cases for $v = u \cdot v'$:

  1. For $v' \preceq [1^{n-1}]$, $org(u \cdot v') = \{u \cdot v'\}$.
  2. For $v' \in \{[1^n], [1^{n+1}], [1^{n+1}, 2]\}$, $org(u \cdot v') = \emptyset$.
     Figure 8.3 shows, using tree representations, the occurrences $[1^n]$, $[1^{n+1}]$ and $[1^{n+1}, 2]$ introduced by $\overline{\eta}$-expansion.
  3. For $v' \succeq [1^{n+2}]$, $org(u \cdot [1^{n+2}] \cdot v'') = \{u \cdot [1^{n+1}] \cdot v''\}$ where $v' \equiv [1^{n+2}] \cdot v''$.

Assume that we have an origin function O mapping occurrences of t' to sets of occurrences in t. Then O is said to be *unitary* if its result values are always sets containing exactly one element, and *unique* if they contain at most one element. If an occurrence can have the empty set as origin, we say O is *forgetful*. If several occurrences in t' have an origin to the same node in t, we may refer to O as *many-to-one*, while its counterpart, where an origin set can contain more than one path, is

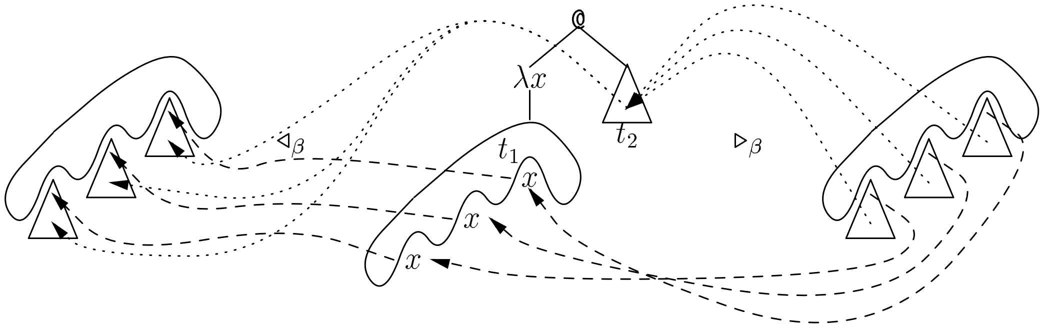


Figure 8.4: β-reduction in both directions.

called *one-to-many*. Finally, if for every $v \in \mathcal{O}(t')$ we have $O(v) = \{v\}$, then we say O is *identical*.

Thus, the origin function is identical for α, is unitary for η, is forgetful for $\overline{\eta}$ and finally, is unitary and many-to-one for β. None of these is one-to-many, which is fortunate, since in Section 7.3.3 we concluded that it was advisable to keep the origin sets small.

8.3.2 Equality Modulo $\beta\overline{\eta}$-conversions

In Section 8.2.3 we discussed reversed β and $\overline{\eta}$-reductions that need to take place during higher-order rewriting. In order to obtain origins for these reversed reductions, the origin functions for $\triangleright_{\{\alpha,\beta,\eta,\overline{\eta}\}}$ defined in the previous section can easily be inverted, thus yielding origin functions for $\triangleleft_{\{\alpha,\beta,\eta,\overline{\eta}\}}$. Note that, from an origin tracking point of view, the inverse of η-reduction is $\overline{\eta}$-expansion.

Since the origin function for α-conversion is identical, performing several α-conversions in one direction or another does not affect the origins. This is not the case for $\overline{\eta}$ or β reduction. Since β-reduction is many-to-one, its inverse must be one-to-many. As can be seen from Figure 8.4, this may lead to a growth of the origin sets. Consider a conversion $t \triangleleft_\beta t' \triangleright_\beta t''$, where $t' = ((\lambda x.t_1)\ t_2)$, and $t, t'' = t_1^{\{x \mapsto t_2\}}$, then the origins from t'' to t' will cause all instantiated occurrences of x to be related to the same t_2 in t'; the origins of t' to t in turn will link this t_2 to all instantiated occurrences of x in t. Thus, transitively, one occurrence of t_2 in t'' has origins to *all* occurrences of t_2 in t. This is illustrated by the dotted lines in Figure 8.4. Note that the definition of the origin function for the β reduction (case 1), relates the *top node* of t_2, via the xs occurring in t_1 to its position in t (dashed lines of Figure 8.4).

Since the origins for $\overline{\eta}$ conversions are unique this problem does not arise for $\overline{\eta}$ conversions. However, the $\triangleright_{\overline{\eta}}$ are forgetful, so checking for $\overline{\eta}$-equality may result in loss of some origin information (in particular in the binders).

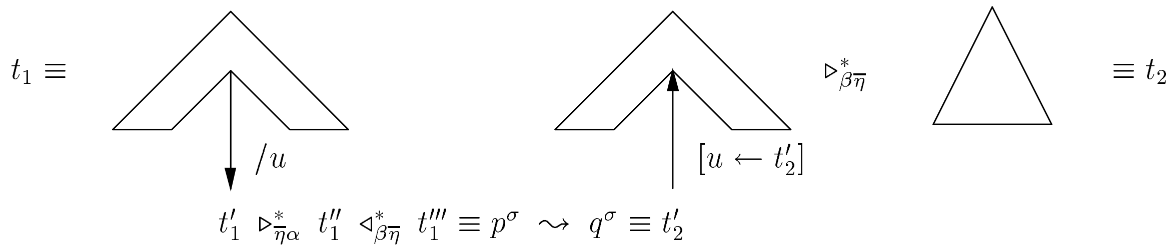


Figure 8.5: All conversions for one reduction step $t_1 \rightarrow t_2$, applying rule $p \rightarrow q$ at occurrence u in t_1 under substitution σ.

8.3.3 Left- and Right-Hand Sides

We define the relations between the instantiated left and right-hand side of a rewrite rule, where we assume that these are instantiated but not yet $\beta\overline{\eta}$-normalized. We closely follow the first-order case defined in Section 6.4.2.

Let $p \rightarrow q$ be a rewrite rule, and σ a substitution. The function $org : \mathcal{O}(q^\sigma) \rightarrow \mathcal{P}(\mathcal{O}(p^\sigma))$, for a path $v \in \mathcal{O}(q^\sigma)$, is defined as follows:

- (Common Free Variables)

  If $v \equiv v' \cdot w$ with $v' \in \mathcal{O}_{fvars}(q)$ denoting some variable X in the right-hand side, and $w \in \mathcal{O}(X^\sigma)$ an occurrence in the instantiation of that variable. Then:

  $$org(v) = \{v'' \cdot w \mid q/v' \equiv p/v'',\ v'' \in \mathcal{O}_{fvars(X)}(p)\}$$

  Thus, v'' denotes an occurrence of X in left-hand side p.

- (Function Symbols)

  If $v \in \mathcal{O}_{bfun}(q)$, then $org(v) = \emptyset$.

This is obviously a forgetful definition, but this situation is improved in Section 8.4. As in the first-order case, it is also possibly one-to-many (in the case of non-left-linearity).

Note that the common free variables case results in the same origins as in the common variables case of Section 6.4.2, when the specification does not use the higher-order features. The Context case will be dealt with in the next section.

8.3.4 Rewrite Steps

Knowing both how to establish origins for α-, β-, and $\overline{\eta}$-conversions in either direction and to set up origins between the instantiated left- and right-hand side, we can obtain the origins for one complete reduction step $t_1 \rightarrow t_2$. Figure 8.5 summarizes the work to be done for one reduction, following the description of Section 8.2.2.

Note that in general the situation is slightly more complicated than in the example of Section 8.2.3

$$P_1 \ \triangleleft_\beta \ P'_1 \ \triangleleft_\beta \ P''_1 \equiv l_1^{\sigma_1} \ \rightsquigarrow \ r_1^{\sigma_1} \equiv P''_2 \ \triangleright_\beta \ P'_2 \ \triangleright_\beta \ P_2$$

where the rewrite rule is applied at the root of P_1 which has the effect that Figure 8.5 can be reduced to just "one level": The context is empty ($u = []$), and consequently the term t/u is already a $\overline{\eta}$-normal form, hence the result need not be put back into the context (in the figure, $[\,[] \leftarrow t'_2]$ is just equal to t'_2).

8.3.5 Example

Consider reduction $P_1 \rightarrow P_2$ as presented in Section 8.2.3. Most occurrences in P_2 have their intuitive origin; mainly because they also occur in bodies of the instantiations of $\mathcal{D}$ and $\mathcal{S}$ in substitution σ_1. However, some origins are lost; in particular for nodes occurring in the right-hand side of rule [1]. Thus, symbols "program", "decl" (for the declaration of n), and "tp" do not have an origin. Moreover, rule [1] is non-linear in X, and therefore the X-occurrence in the declaration at the right-hand side has an origin to the occurrence in the statement as well as in the declaration. Thus, the single n in P_2 has origins to all n occurrences in P_1 (this does not seem intuitive). All occurrences of "natural" in P_2 have their origin to the declaration it came from (seems reasonable).

Now consider the entire reduction $P_1 \rightarrow^* P_4$, where normal form P_4 is:

```
program( decls( decl(n,natural),  decls( decl(s,string), emptydecls) ),
         stats( assign( tp(string),
                  plus(id( tp(natural) ),id( tp(natural) ))),
                emptystats )  )
```

In this case, more origins are lost. In particular, the two "decl" nodes have empty origins, and the reduction according to rule [4] did not establish any origins, so "tp(natural)" does not have any origins.

8.4 Extensions

The origins in the previous example were nice, but still not sufficient for using them in practice. In this section we present some extensions of the origin function. Some of these extensions are of a heuristic nature, based on frequently occurring forms of (higher-order) rewrite rules.

8.4.1 Extended Contexts

Taking a close look at equation [1] of Figure 8.2, we see that its intention is to identify some context "program(...)" in which a certain term (the identifier denoted by X) is to be replaced by another term (in this case tp(τ)). This context is exactly the same in the left- and right-hand side of the rewrite rule.

It seems reasonable to extend the notion of a context to cover such similarities within rewrite rules as well. Considering a rewrite rule $p \rightarrow q$, we can look for a (possibly empty) *common context* C and *holes* (terms) $h_1, \ldots, h_m$ and $h'_1 \ldots h'_m$ ($m \geq$

0) such that $p =_\alpha C[h_1, \ldots, h_m]$ and $q =_\alpha C[h'_1, \ldots, h'_m]$, where $h_j \neq_\alpha h'_j$ for all $1 \leq j \leq m$. We are actually looking for the biggest of such contexts which contain the smallest possible number of holes where none of the holes h_j, h'_j $(1 \leq j \leq m)$ start with a non-empty context $\overline{C}$ such that $h_j =_\alpha \overline{C}[\overline{h}_1, \ldots \overline{h}_n]$ and $h'_j =_\alpha \overline{C}[\overline{h'}_1, \ldots \overline{h'}_n]$. As an example, equation [1] of Figure 8.2 has a common context $C =$ "program($\mathcal{D}$(decl(X,τ)), $\mathcal{S}(\Box)$)", where the hole h_1 at the left is equal to "X", and h'_1 at the right to "tp(τ)".

For every node in this extended context, the origin should point only to its corresponding occurrence in that same context at the left-hand side. Note that, as a consequence, the common variables case should *not* apply to variables occurring in the common context. For example, in equation [1], the origin of X at the right will only point to its counterpart under the "decl" at the left and not to the X in the statements. Moreover, when trying to find origins for a node in a hole h'_j, it seems reasonable to focus on origins that can be found within the corresponding hole h_j. Only if it is impossible to find origins there, an origin can be looked for in the rest of the left-hand side.

There is, however, a minor catch in this. If two consecutive holes h_j and h_{j+1} are only separated by an application in the context C, i.e. they actually occur as $@(h_j, h_{j+1})$ at the left and as $@(h'_j, h'_{j+1})$ at the right, then it is more natural to regard these two as one hole ($H = @(h_j, h_{j+1})$ instead of h_j and h_{j+1}). As an example, equation [2] in applicative form reads as @(nat,N) = @(id, @(tp, natural)). It would be counter-intuitive to regard the top-application as a common context $@(\Box, \Box)$ with two holes: $h_1 =$ nat, $h'_1 =$ id, and $h_2 = N$, $h'_2 =$ @(tp, natural).

Note that this new extended context case would be useful in the first-order case as well.

8.4.2 Origins for Constants

Let $p \equiv C[h_1, \ldots, h_m] \rightarrow C[h'_1, \ldots, h'_m] \equiv q$ be a rewrite rule with the common context C and m $(m \geq 0)$ holes. We define origins for constants occurring in the h'_j $(1 \leq j \leq m)$ according to the following three cases:

1. Head-to-Head

   The origin for the occurrence of the head symbol of a hole h'_j at the right is the occurrence of the head symbol of that same hole h_j at the left. For example, the "tp" symbol in equation [1] is linked to the occurrence of X in the statements at the left. This head-to-head rule corresponds to the "redex-contractum" rule of the first-order origins as described in Section 6.4.4. Note that if the head symbol at the right is a free variable, the common variables case is applicable as well. This can, in general, have the effect that the origin set for the head symbols consist of more than one path.

2. Common Subterms.

   If a term s is a subterm of both h'_j and h_j, then these occurrences of s are related. For example, the subterm "tp(natural)" at the right of equation [4] (Figure 8.2) is related to both occurrences of "tp(natural)" at the left. Note that these common subterms are identified in the *un-instantiated* left- and right-hand side. This rule can in some cases lead to seemingly wild connections, but has already proven its usefulness for the first-order case, as discussed in Sections 6.4.4 and 7.6. The common subterms behave slightly different in the higher-order case, due to the applicative form of the λ-terms. In the first-order case, function symbols were only related if all arguments were identical at the left and right. In the higher-order case, function symbols are constants. Each constant F in h'_j is related to all occurrences of F in h_j. This effect is similar to the *tokenization* discussed in [Din93].

If for a subterm s of h'_j no occurrences of s can be found in h_j, then the entire left-hand side p can be used to find a common subterm occurrence of s.

3. Any to All.

   If after application of the head-to-head and common subterms case there are still constants in h'_j with empty origins, these obtain the set of all constant occurrences from the left-hole h_j as their origin. For example, in equation [2], the subterms "tp(natural)" and "natural" relate to both "nat(N)" and "N".

8.4.3 Abstraction and Concretization Degree

Let us end our discussion with an interesting observation. Recall from Section 8.3.2 that $\triangleleft_\beta$ conversions are one-to-many. Assume that $t' \triangleleft_\beta t$ with $t \equiv ((\lambda x.t_1)\ t_2)$. It would be useful to call the number of free occurrences of x in t_1 the *abstraction degree* of $\lambda x.t_1$, and the number of occurrences of term t_2 in t_1 the *concretization degree*. When trying to find a matching substitution σ in order to apply a rewrite rule, freedom exists concerning the abstraction and concretization degree. For example, if σ assigns F a value T with abstraction degree $N > 0$ and concretization degree $M \geq 0$, then an alternative match σ' can also be possible which assigns F a term T' with abstraction degree $N-1$ and concretization degree $M+1$. The problems with $\triangleleft_\beta$ are minimized if matches with abstraction degree 1 are preferred over those with a higher abstraction degree.

In practice, however, such a preference may be somewhat problematic. Firstly, a substitution with a lower degree of abstraction may not even exist. Secondly, the repeated application of a substitution with abstraction degree 1 need not yield the same result as a single application with a higher abstraction degree. Finally, repeated applications may be more expensive in terms of run time behavior, than a single application with a high abstraction degree.

8.4.4 Example

With these extensions, suitable origins for the example in Section 8.2.3 are obtained. We assume that equation [1] is applied with substitutions of abstraction degree 1 only. The extended contexts assure that "program" and "decl" are linked. Moreover, the effect of linking variables in contexts only to the same occurrence in the context, guarantees that the n and s in the declaration have the proper unitary origin. Furthermore, relating heads of holes guarantees that the "tp" nodes get the right origin to the variable they were substituted for. Likewise, the application of equation [4] results in "plus" as the origin of "tp". Finally, common subterms results in "tp(natural)" to be linked to both occurrences of "tp(natural)" in the "plus" expression (equation [4]).

The example given here is only part of the specification discussed in [Hee92]. The origins with extensions create the proper relations for the full specification as well.

8.5 Concluding Remarks

In this chapter we have studied how the notion of origin tracking can be extended to term rewriting systems containing bound variables. It turned out that the extra β and η conversions and expansions cause a considerable extra complication. Nevertheless, it has been possible to come up with a scheme that works satisfactory for the full type checking example given by Heering [Hee92] (of which we have shown only a small part).

We have discussed origins in the liberal setting for higher-order term rewriting proposed by Wolfram [Wol93]. Nipkow discusses a more restrictive definition where the rewrite rules are to satisfy several syntactic constraints [Nip91]. Obviously, the same origin function can be applied in Nipkow's HRSs. The nicer matching behavior of Nipkow's HRSs will probably have a favorable effect on the origins.

Origin tracking is a technique grown out of practical need. Full assessment of higher-order origin tracking is only possible when more higher-order specifications of programming languages are available. We expect higher-order origin tracking to be a useful device when executing these language descriptions using rewriting.

An interesting area for further research might be a closer study of the various labeling schemes for λ-calculus discussed, e.g., by Lévy [Lév75]. His labels contain much more structure than our occurrences, which might be used to encode more information in structures representing origins. Moreover, a better understanding of the exact relationship with the various descendance or residual relations would be beneficial [Oos94]. An interesting comparison between residuals and labels for the λ-calculus and term rewriting systems is discussed by [Ber92].

Chapter 9

Conclusions

The basic idea of this thesis is that algebraic specifications can be fruitfully used for the formal definition of computer languages. Not only do these specifications yield a description of the most import features of the language concerned, they can also be executed (by means of term rewriting) and in this way they can be used to obtain a prototype environment for said language automatically.

In this thesis, we have tried to obtain a better understanding of what the opportunities and problems are when using algebraic specifications for the purpose of defining languages. We did this by working out a number of case studies. For one of the problems encountered in the case studies we discussed a solution (origin tracking) in considerable detail.

In this final chapter we will summarize the various chapters, and indicate what, in our opinion, are the main contributions of this thesis. We will conclude with a brief glance at the future, mentioning two possibilities for further research.

9.1 Summary

Chapter 2: The λ-calculus The various case studies use the ASF+SDF formalism and the ASF+SDF Meta-environment. The λ-calculus specification simply served to introduce ASF+SDF by example. The full specification, as well as the resulting λ-calculus environment were discussed.

Chapter 3: Financial Products The formalization of the language RISLA for defining interest rate products was commissioned by an industrial partner, CAP Volmac in cooperation with Bank MeesPierson. Writing a grammar for RISLA, and specifying some of the underlying data types, had a very clarifying effect during the design of RISLA.

The executability of the formalism was valuable while specifying the fundamental data types of RISLA, such as cash flows, balances, date intervals, and so on. The bank decided, however, not to use ASF+SDF to build prototype tools for RISLA, as it was unclear how difficult it would be to express, using ASF+SDF, the necessary interfacing with the bank's running systems. We believe that it can indeed be beneficial to use algebraic specifications for these purposes. The specified compiler from RISLA to COBOL described by Res [Res94] is a promising step in this direction.

Chapter 4: Static Semantics of Pascal In order to assess the usefulness of ASF+SDF for the definition of realistic languages, we studied the definition of the static semantics of Pascal. The specification shows how a disciplined and systematic use of the user-definable syntax of ASF+SDF can help to make a language definition formal, yet readable and close to the (non-formalized) ISO definition.

The Pascal specification was one of the reasons we started studying origin tracking. Only with origin tracking, this specification can be used to obtain a type checker that indicates *where* in a Pascal program the errors are made.

Chapter 5: Tools for Action Semantics The most challenging case study was the construction of the ASD environment, which gives support for writing Action Semantic Descriptions. A major complicating factor was the use of mixfix syntax in the language, the "Meta Notation" (MN), to be described. Moreover, MN was characterized by specifying a translation from MN modules to ASF+SDF modules. In order to *use* these modules, they had to be loaded interactively into a running Meta-environment; something not easily expressible within the ASF+SDF formalism.

Using ASF+SDF to build the ASD Tools forced us to visit almost all (remote) corners of the ASF+SDF formalism and system. This resulted in a list of suggestions for improvement, mostly dealing with ambiguities.

Although we have been able to come up with a solution for most of the problems we came across, we have not yet found time to generalize these solutions. Several recent applications, such as the work on pretty-printing by Van den Brand [Bra93] or on natural language processing by Groenink also translate to ASF+SDF. All this work might benefit from a more abstract treatment of the problems we encountered for ASD.

Chapter 6: Origin Tracking The wish to obtain a better type checker from the Pascal specification, and the ongoing research on "generic debugging", were the two direct causes to start exploring *origin tracking* in the style of Bertot [Ber91b, Ber92, Ber93]. We came up with an extension to non-orthogonal, conditional TRSs. Moreover, we considered *secondary origins*, which maintain connections between Common Subterms and the Redex-Contractum. We formulated several properties to identify the conditions under which origin sets are empty.

Given a specification of the static or dynamic semantics of a language, or a translation from a programming language to some machine language, origin tracking can be used, without the need to adapt the specification, to generate error handlers for type checkers, animators or source-level debuggers. For animation, the origins proposed are satisfactory; for error handling or source-level debugging the risk is too high that the origin sets are empty.

Chapter 7: Primitive Recursive Schemes Because primary and secondary origins are frequently empty, these origin tracking mechanisms are unsuccessful when applied to the Pascal specification. Inspired by Van der Meulen's work on incremental term rewriting [Meu92, Meu94], we studied a subclass of algebraic specifications, Primitive Recursive Schemes, a subclass which is ubiquitous in language definitions.

In the strict format of a PRS it is easy to come up with a useful origin tracking mechanism that never produces empty origin sets. Moreover, we showed how it remains possible to maintain good origins in a more liberal setting, giving rise to *syntax-directed* origins. We expect that these syntax-directed origins will become a frequently used feature of the ASF+SDF Meta-environment.

Chapter 8: Higher-Order Specifications Recent experiments with the use of higher-order algebraic specifications for writing executable language definitions [Hee92] made an extension of origin tracking to higher-order TRSs necessary. Here the complicating factor is that matching is no longer based on syntactic equivalence, but on equality modulo $\beta\overline{\eta}$ conversions.

By giving origin rules for α, β, and η reduction and expansion, it is possible to obtain an origin tracking mechanism that provides useful origins for error-handling and type-checking purposes.

9.2 Contributions

The contributions of this thesis are:

- Further emperical evidence for the conjecture that algebraic specifications are a suitable vehicle for the construction of executable language definitions:
  - An example where algebraic specification methods helped to improve the design and implementation of an application language used in the financial engineering domain;
  - A prospect of how an ASF+SDF definition for a realistic language can remain readable and close to an official language standard;
  - The ASD Tools, used to check existing Action Semantic Descriptions, to write new ones, and to teach action semantics;
  - A detailed list containing several suggestions for improvements in the ASF+SDF formalism and system (Section 5.7);
- An extensive study of origin tracking, covering
  - a definition for non-linear TRSs, and conditional TRSs;
  - a discussion of secondary origins, including the Redex-Contractum and Common Subterms cases;
  - a series of properties, identifying when origin sets will be empty;

- an account of the applicability to error-handling, animation and source-level debugging.
- An extension of origin tracking yielding non-empty and useful origins for specifications written in a syntax-directed style;
- A proposal for establishing origins for higher-order term rewriting systems.

9.3 Further Research

Type Checking Writing a specification of the static semantics for a realistic language is still a complicated task.

The Pascal definition discussed in Chapter 4 is satisfactory in that it is readable and stays reasonably close to the ISO definition. As mentioned in Chapter 4, however, the use of explicit variables to pass states between type check functions is undesirable. An alternative to these states could be the use of *combinators*, for instance close to the combinators used in Action Notation (see Chapter 5).

Doing so might have a second advantage. A desirable property of a static semantics is that — in some sense — it is consistent with the dynamic semantics. For instance, if the dynamic semantics concludes that a variable N should be assigned the value 3, then the static semantics should not claim that N has type `char`. Using a notation (action combinators) in a static semantics that is close to a notation that can be used for the dynamic semantics (an action-semantic definition) will help to reason about the correspondence between the static and the dynamic semantics.

Taking this point to the extreme brings us to the area of *abstract interpretation*, where static analysis is performed by interpreting the meaning of a program abstractly [AH87]. Some recent experiments [Hee92, Din93, Res94] have shown that it is possible and can be beneficial to characterize type checking by evaluating over an abstract type domain. Lacking in their approach so far, however, is a well-defined correspondence between the type domain and the dynamic semantics. Combining this approach with the use of action combinators when defining static semantics might provide a solution to this problem.

Origin Tracking The research on origin tracking could be continued in more than one way. A large number of applications exists for which the automatic maintenance of an origin-like relation will be beneficial — one can think of program slicing, mappings between programs and their visual representation, theorem proving systems, and so on. Moreover, the technique could be defined for other frameworks than term rewriting: Functional languages seem a natural candidate, but it should even be possible to write a C-library helping in the maintenance of origin information.

Alternatively, one could see how origin tracking behaves for some real hard case studies. A good candidate is specifying a type checker of ML. It is well-known that the extensive type inference of ML makes it difficult to yield error messages that still have a clear relation to the source program. Origin tracking has been designed to maintain such information automatically.

Appendices

Appendix A

ASD Meta Notation in SDF

The following six ASF+SDF modules define a grammar for the Meta Notation formalism used to write unified-algebraic specifications and action-semantic descriptions.

Particularly noteworthy is module Symbols (Module A.2), defining a syntax for the symbols that can be introduced in Meta Notation modules. Some examples of legal symbols are `_[_]`, `the sum of (_, _)`, `Expressions`, or `[[<_,_>]]`. Not allowed are strings or characters, i.e., the characters `"`, `'` or `` ` `` are not to occur in symbols. Moreover, only well-balanced bracketing is allowed for bracket pairs brackets `<...>`, `{...}`, `[...]`, and `(...)` are only allowed in pairs. As a consequence a symbol like `_<_` is not allowed.

Module Terms (Module A.3) is basically an extension of module Symbols. The place holder `_` is replaced by strings `"..."`, characters `` `...' ``, as well as in-term variable declarations. These variable declarations are of the form $v : s$ or $v \leq s$, with v a variable and s a sort, specified by the SUB-RELATOR line. Moreover, the built-in infix operators `_|_` and `_&_` are supported. Originally, a slightly simpler grammar (containing one sort OUTER rather than {U,R,V,L,G}-OUTER was used containing priority declarations to deal with the priorities between infixes `_|_` and `_&_`, postfixes `_*`, `_?`, and `_+`, and special sort-denoters `_^_` and `_[_]`. Parsing time for that very ambiguous grammar was unacceptable, however, since the SDF parser had to deal with too many parsing possibilities before it could reject some of them using priority declaration information. Therefore, a more classical grammar was chosen allowing significantly less parses for each sentence.

The remaining modules Functionalities (Module A.4), Clauses (Module A.5), and MetaNotation (Module A.6) are almost direct copies of the abstract syntax description given in [Mos92, Appendix F.3]. Following [Mos92, Appendix F.1.1.4], bracketing is allowed for clauses and formulae; but for the (more extensive) module and grammar it was decided to use keywords instead of ordinary brackets. In practice, section numbering is used in [Mos92] to determine module nesting — unattractive in an interactive environment.

A.1 Module Lexicals

exports

sorts UPPER-ALPHA LOWER-ALPHA ALPHA-QUOTE ALPHA
NUMBER CHARACTER STRING NON-ALPHA

lexical syntax

[\t\n] → LAYOUT
“%%”~[\n]*[\n] → LAYOUT

[0-9]+ → NUMBER

[A-Z][a-zA-Z0-9\−]*[a-zA-Z0-9] → UPPER-ALPHA
[A-Z] → UPPER-ALPHA

[a-z][a-zA-Z0-9\−]*[a-zA-Z0-9] → LOWER-ALPHA
[a-z] → LOWER-ALPHA

[a-zA-Z][a-zA-Z0-9\−]*[a-zA-Z0-9][′]+ → ALPHA-QUOTE
[a-zA-Z][′]+ → ALPHA-QUOTE

~[_() <>{}\[\]′“”\ \t\nA-Za-z0-9, . :;&|? * +] → NON-ALPHA

“\””STRING-CHAR*“\”” → STRING
“\” “\”” → STRING-CHAR
“\” “\” → STRING-CHAR
~[”\n\] → STRING-CHAR

“‘” ~[\n\t]“′” → CHARACTER

context-free syntax

UPPER-ALPHA → ALPHA
LOWER-ALPHA → ALPHA
ALPHA-QUOTE → ALPHA

A.2 Module Symbols

imports Lexicals(A.1)
exports

sorts SYMBOLS SYMBOL PLACE S-OUTER S-INNER T-INNER

context-free syntax

“_” → PLACE
{SYMBOL “,”}+ → SYMBOLS

S-OUTER+ → SYMBOL
PLACE “?” → SYMBOL
PLACE “*” → SYMBOL
PLACE “+” → SYMBOL

“(” T-INNER “)” → S-OUTER
“<” T-INNER “>” → S-OUTER
“{” T-INNER “}” → S-OUTER
“[” T-INNER “]” → S-OUTER

ALPHA	→ S-OUTER
NON-ALPHA	→ S-OUTER
NUMBER	→ S-OUTER
PLACE	→ S-OUTER
“.”	→ S-INNER
S-OUTER	→ S-INNER
S-INNER+ “,” T-INNER	→ T-INNER
S-INNER∗	→ T-INNER

A.3 Module Terms

imports Lexicals(A.1)
exports
sorts TERM TERMS U-OUTER R-OUTER V-OUTER L-OUTER G-OUTER U-INNER G-INNER SUB-RELATOR
context-free syntax

{TERM “,”}+	→ TERMS
“:”	→ SUB-RELATOR
“≤”	→ SUB-RELATOR
U-OUTER	→ TERM
U-OUTER “|” R-OUTER	→ U-OUTER
U-OUTER “&” R-OUTER	→ U-OUTER
R-OUTER	→ U-OUTER
V-OUTER+	→ R-OUTER
G-OUTER SUB-RELATOR L-OUTER	→ V-OUTER
L-OUTER	→ V-OUTER
L-OUTER “∗”	→ L-OUTER
L-OUTER “?”	→ L-OUTER
L-OUTER “+”	→ L-OUTER
L-OUTER “^” G-OUTER	→ L-OUTER
L-OUTER “[” U-OUTER “]”	→ L-OUTER
G-OUTER	→ L-OUTER
“(” U-INNER “)”	→ G-OUTER
“<” U-INNER “>”	→ G-OUTER
“{” U-INNER “}”	→ G-OUTER
“[” U-INNER “]”	→ G-OUTER
ALPHA	→ G-OUTER
NON-ALPHA	→ G-OUTER

NUMBER → G-OUTER
STRING → G-OUTER
CHARACTER → G-OUTER

"." → G-INNER
U-OUTER → G-INNER

G-INNER+ "," U-INNER → U-INNER
G-INNER∗ → U-INNER

A.4 Module Functionalities

imports Terms(A.3)
exports
sorts ATTRIBUTE FUNCTIONALITY
context-free syntax
TERMS "→" TERM → FUNCTIONALITY
TERMS "→" TERM "(" {ATTRIBUTE ","}+ ")" → FUNCTIONALITY

"*total*" → ATTRIBUTE
"*partial*" → ATTRIBUTE
"*restricted*" → ATTRIBUTE
"*strict*" → ATTRIBUTE
"*linear*" → ATTRIBUTE
"*injective*" → ATTRIBUTE
"*associative*" → ATTRIBUTE
"*commutative*" → ATTRIBUTE
"*idempotent*" → ATTRIBUTE
"*unit*" "*is*" TERM → ATTRIBUTE

A.5 Module Clauses

imports Functionalities(A.4)
exports
sorts TAG RELATOR CLAUSE CLAUSES FORMULA
FORMULAE RELATION DISJOINER
context-free syntax
{CLAUSE ";"}+ → CLAUSES
"(" CLAUSES ")" → CLAUSES {**bracket**}
FORMULA → CLAUSE
FORMULAE "⇒" CLAUSES → CLAUSE
SYMBOLS "::" {FUNCTIONALITY ","}+ → CLAUSE
TAG CLAUSE → CLAUSE
"(" CLAUSE ")" → CLAUSE {**bracket**}

{FORMULA “;”}+	→ FORMULAE
“(” FORMULAE “)”	→ FORMULAE {**bracket**}
TERM “!”	→ FORMULA
RELATION “(” DISJOINER “)”	→ FORMULA
RELATION	→ FORMULA
TAG FORMULA	→ FORMULA
“(” FORMULA “)”	→ FORMULA {**bracket**}
“(” “*” “)”	→ TAG
“(” NUMBER “:” “)”	→ TAG
“[” NUMBER “:” “]”	→ TAG
TERMS RELATOR TERM	→ RELATION
RELATION RELATOR TERM	→ RELATION
“=”	→ RELATOR
“: −”	→ RELATOR
“≥”	→ RELATOR
SUB-RELATOR	→ RELATOR
“*disjoint*”	→ DISJOINER
“*individual*”	→ DISJOINER

priorities

{ TAG FORMULA → FORMULA } < { TAG CLAUSE → CLAUSE }

A.6 Module MetaNotation

imports Clauses(A.5) Symbols(A.2)
exports
sorts SPECIFICATION MODULE BASIC VARIABLES-TERM TITLE PATH REFERENCE REFERENCES TRANSLATION
context-free syntax

MODULE+	→ SPECIFICATION
BASIC+	→ SPECIFICATION
BASIC+ MODULE+	→ SPECIFICATION
“*module*” “:” PATH “.” SPECIFICATION “*endmodule*” “:” PATH “.”	→ MODULE
“*needs*:” REFERENCES “.”	→ BASIC
“*includes*:” REFERENCES “.”	→ BASIC
“*closed*” “*except*:” REFERENCES “.”	→ BASIC
“*closed*” “.”	→ BASIC
“*open*” “.”	→ BASIC
“*introduces*:” SYMBOLS “.”	→ BASIC
“*privately*” “*introduces*:” SYMBOLS “.”	→ BASIC

"*grammar*:" BASIC+ "*endgrammar*" "."	→ BASIC
"*grammar*:" MODULE+ "*endgrammar*" "."	→ BASIC
"*variables*:" {VARIABLES-TERM ";"}+ "."	→ BASIC
CLAUSES "."	→ BASIC
"."	→ BASIC
"(" BASIC ")"	→ BASIC {**bracket**}
TERMS SUB-RELATOR TERM	→ VARIABLES-TERM
UPPER-ALPHA+	→ TITLE
{TITLE "/"}+	→ PATH
PATH	→ REFERENCE
PATH "(" {TRANSLATION ","}+ ")"	→ REFERENCE
{REFERENCE ","}+	→ REFERENCES
SYMBOL	→ TRANSLATION
SYMBOL "*for*" SYMBOL	→ TRANSLATION

Appendix B

The Origin Function in ASF+SDF

In order to be able to conduct experiments with origin definitions, we have constructed a small ASF+SDF specification characterizing several variants of the origin function. The specification stays as close as possible to the definitions given in Sections 6.4 and 7.5.2. Note that this is an interesting example of the use of ASF+SDF outside the scope of language definitions.

The ASF+SDF specification described in this appendix defines term rewriting and a small extension of rewriting, namely *origin tracking*. This specification can be used to experiment with various definitions of origin tracking schemes. Moreover, as the specification includes a definition of rewriting, it can be used to play with other extensions of term rewriting as well.

Section B.1 just introduces some basic concepts of term rewriting, such as *terms*, *function symbols*, *substitutions*, *occurrences*, *rewrite rule*, *rewrite system*, and so on. The notation used directly comes from [Klo92], and corresponds to the notation used in Section 6.4.1. Section B.1.7 introduces a representation for one reduction-step, i.e., a tuple containing a term, path to a redex, a rewrite rule to be applied, and the substitution to be used.

Section B.2 defines a function "normalize" taking an initial term and a rewrite system, and producing a list of "reduction-step" structures, representing the full reduction. The reduction strategy used here is outermost. It is straightforward to experiment with other rewrite strategies. Moreover, one could extend the definition easily to cover *conditional* rewriting as well; in that case each step-tuple could include the sequences of steps needed to evaluate the conditions.

Section B.3, finally, deals with primary and secondary origin tracking as described in Chapter 6. When defining origins, several alternative definitions can be given. This specification enables one to perform some small scale experiments, or to introduce a new origin definition. The definition of origins is a function taking a sequence of steps, and producing a map from occurrences in the normal form to sets of occurrences in the initial term.

The actual definition given aims at staying as close as possible to the definitions and notation used in Chapters 6 and 7.

The specification was written using ASF+SDF's literate specification tools, which allow a straightforward mapping between ASCII words and LaTeX symbols. Moreover, certain patterns are treated in a special way. If a variable declaration is of the form $V[0-9]*$, then a a variable written as, e.g., $V1$ is printed as V_1. Moreover, a function name containing ^ or _ is used to write arguments in sub or superscript format.

The full sources are available by anonymous *ftp* from ftp.cwi.nl, directory pub/gipe/spec, file rewriting+origins.tar.Z.

B.1 Term Rewriting Systems

B.1.1 Module Rewrite-Term

Terms are built from a (lexically defined) set of *function symbols*, and a set of *variable symbols*. Following Prolog, variables start in uppercase, functions start in lowercase. Every variable V is a term, and if F is a function symbol and $t_1, ...t_n (n \geq 0)$ terms, then $F(t_1, ..., t_n)$ is a term. If $n = 0$, then $F()$ is also called a constant.

imports Layout$^{(2.3)}$
exports
  sorts FUN-SYMBOL VAR-SYMBOL TRS-TERM
  lexical syntax
    $[a\text{-}z][A\text{-}Za\text{-}z0\text{-}9\backslash-]*$ → FUN-SYMBOL
    $[A\text{-}Z][A\text{-}Za\text{-}z0\text{-}9\backslash-]*$ → VAR-SYMBOL

  context-free syntax
    VAR-SYMBOL → TRS-TERM
    FUN-SYMBOL "(" {TRS-TERM ","}* ")" → TRS-TERM
    "(" TRS-TERM ")" → TRS-TERM {**bracket**}

Export variables denoting a variable symbol, function symbol or term. To avoid confusion, these are prefixed with a "‘" character (not strictly necessary, though).
  variables
    "‘"*F* [*0-9*]* → FUN-SYMBOL
    "‘"*V* [*0-9*]* → VAR-SYMBOL
    "‘"*T* [*0-9*]* → TRS-TERM
    "‘"*T* [*0-9*]*"*" → {TRS-TERM ","}*
    "‘"*T* [*0-9*]*"+" → {TRS-TERM ","}+

B.1.2 Module Rewrite-Rule

A rewrite rule r is a pair of two open terms t_1, t_2 written $t_1 \rightarrow t_2$.

imports Rewrite-Term$^{(B.1.1)}$
exports
  sorts RULE
  context-free syntax
    TRS-TERM "→" TRS-TERM → RULE
    RULE "." *lhs* → TRS-TERM
    RULE "." *rhs* → TRS-TERM

variables
R [*0-9*]$*\rightarrow$ RULE

equations

$$`T_1 \rightarrow `T_2 \,.\, lhs = `T_1 \qquad \text{[l]}$$
$$`T_1 \rightarrow `T_2 \,.\, rhs = `T_2 \qquad \text{[r]}$$

B.1.3 Module Rewrite-System

A term rewriting system consists of a set of rewrite rules.

imports Rewrite-Rule[(B.1.2)]
exports
sorts TRS
context-free syntax
"{" {RULE ","}$*$ "}" $\rightarrow$ TRS
variables
TRS [*0-9*]$*$ $\rightarrow$ TRS
R [*0-9*]$*$"$*$" $\rightarrow$ {RULE ","}$*$
R [*0-9*]$*$"+"$\rightarrow$ {RULE ","}+

B.1.4 Module Substitution

A substition is a mapping from variables to terms, indicating which variables are to be given which values. Given a term t and a substitution σ, the result of applying σ to t is written t^{σ}.

imports Rewrite-Term[(B.1.1)] Booleans[(2.1)]
exports
sorts SUBSTITUTION ONE-MAP
context-free syntax

VAR-SYMBOL "$\mapsto$" TRS-TERM	$\rightarrow$ ONE-MAP
"{" {ONE-MAP ","}$*$ "}"	$\rightarrow$ SUBSTITUTION
SUBSTITUTION "$\circ$" SUBSTITUTION	$\rightarrow$ SUBSTITUTION
TRS-TERM "^" SUBSTITUTION	$\rightarrow$ TRS-TERM
VAR-SYMBOL "$\in$" SUBSTITUTION	$\rightarrow$ BOOL
SUBSTITUTION "(" VAR-SYMBOL ")"	$\rightarrow$ TRS-TERM

variables
"σ"[*0-9*]$*$ $\rightarrow$ SUBSTITUTION
M [*0-9*]$*$"$*$"$\rightarrow$ {ONE-MAP ","}$*$

equations

Substitution Composition

$$\{M_1^*\} \circ \{M_2^*\} = \{M_1^*, M_2^*\} \qquad \text{[c1]}$$

Search Operations in σ mappings

$$`V \in \{M_1^*, `V \mapsto `T, M_2^*\} = \mathit{true} \qquad \text{[s1]}$$
$$`V \in \sigma = \mathit{false} \qquad \textbf{otherwise} \qquad \text{[s2]}$$

$$\{M_1^*, `V \mapsto `T, M_2^*\}(`V) = `T \qquad \text{[s3]}$$

Apply a Substitution

$`V$ has an entry in $\sigma$. Replace $`V$.

$$\frac{`V \in \sigma = \mathit{true}}{`V^{\,\sigma} = \sigma(`V)} \qquad \text{[a1]}$$

$`V$ does not occur in $\sigma$; leave $`V$ unchanged.

$$\frac{`V \in \sigma = \mathit{false}}{`V^{\,\sigma} = `V} \qquad \text{[a2]}$$

A constant need not be replaced.

$$`F()^{\,\sigma} = `F() \qquad \text{[a3]}$$

A function with one or more arguments.

$$\frac{`F(`T_1^*)^{\,\sigma} = `F(`T_2^*)}{`F(`T_1, `T_1^*)^{\,\sigma} = `F(`T_1^{\,\sigma}, `T_2^*)} \qquad \text{[a4]}$$

B.1.5 Integers

The standard module defining integers is not shown.

B.1.6 Module Occurrences

An occurrence of a subterm in a term is denoted by a sequence of integers indicating the access path from the root. For example, occurrence $[1, 2]$ denotes the second son of the first son of the root, so for term $f(g(a, b), c))$ it denotes subterm b. Finding the substerm in t identified by occurrence p is written t/p. Replacing the subterm at position p in term t_1 by a new subterm t_2 is written $t_1[p \leftarrow t_2]$.

imports Integers$^{(B.1.5)}$ Rewrite-Term$^{(B.1.1)}$
exports
sorts OCCURRENCE
context-free syntax

"[" {INT ","}* "]"	→ OCCURRENCE	
OCCURRENCE ";" OCCURRENCE	→ OCCURRENCE	{**left**}
"(" OCCURRENCE ")"	→ OCCURRENCE	{**bracket**}
TRS-TERM "/" OCCURRENCE	→ TRS-TERM	
TRS-TERM "[" OCCURRENCE "←" TRS-TERM "]"	→ TRS-TERM	
OCCURRENCE "≺" OCCURRENCE	→ BOOL	
OCCURRENCE "⪯" OCCURRENCE	→ BOOL	
OCCURRENCE "|" OCCURRENCE	→ BOOL	

Variables u, v, and w range over zero or more integers of an occurrence. Associative matching over lists is used to write, e.g., $[u, v]$, where both u and v can be empty.

variables
*[uvw][']**→ {INT ","}*
*N [0-9]** → INT
*O [0-9]** → OCCURRENCE

equations

$$[u]; [v] = [u, v] \tag{1}$$

Finding a Subterm

$$`T \;/\; [] = `T \tag{2}$$

$$`F(`T, `T^*) \;/\; [1, u] = `T \;/\; [u] \tag{3}$$

$$`F_1(`T_1, `T_1^*) \;/\; [N, u] = `F_1(`T_1^*) \;/\; [N-1, u] \Leftarrow N > 1 = true \tag{4}$$

Replacing a Subterm

$$`T_1[[] \leftarrow `T_2] = `T_2 \tag{r1}$$

$$`F(`T, `T^*)[[1, u] \leftarrow `T_2] = `F(`T[[u] \leftarrow `T_2], `T^*) \tag{r2}$$

$$\frac{\begin{array}{c} N > 1 = true, \\ `F_1(`T_1^*)[[N-1, u] \leftarrow `T_2] = `F_2(`T_2^*) \end{array}}{`F_1(`T_1, `T_1^*)[[N, u] \leftarrow `T_2] = `F_1(`T_1, `T_2^*)} \tag{r3}$$

Orderings on Paths

$$
\begin{array}{lll}
[v] \preceq [v, w] = \mathit{true} & & \text{[o1]} \\
[v] \preceq [u] \quad = \mathit{false} & \textbf{otherwise} & \text{[o2]} \\
[v] \prec [u] \quad = \mathit{true} \Leftarrow [v] \preceq [u] = \mathit{true},\ [v] \neq [u] & & \text{[o3]} \\
[v] \prec [u] \quad = \mathit{false} & \textbf{otherwise} & \text{[o4]} \\
[v] \mid [u] \quad = \mathit{true} \Leftarrow [v] \preceq [u] = \mathit{false},\ [u] \preceq [v] = \mathit{false} & & \text{[o5]} \\
[v] \mid [u] \quad = \mathit{false} & \textbf{otherwise} & \text{[o6]}
\end{array}
$$

B.1.7 Module Reduction-Step

A reduction step is a 4-tuple $\langle t, p, \sigma, r\rangle$ indicating that in term t at occurrence p under substitution σ rewrite rule r is applicable. Define some straightforward access functions on such steps, using field selectors as in Pascal. Steps can be sequenced to a series of steps, concatenated by ";". In order to deal with failing searches for reductions, a unit-step is added which does not perform an actual step.

Note that the outcome of a step, the *contractum*, is not part of the step; it can be computed from these four, as is done by function "*.contractum*".

imports Rewrite-Rule$^{(B.1.2)}$ Occurrences$^{(B.1.6)}$ Substitution$^{(B.1.4)}$
exports
  sorts STEP STEPS TRUE-STEP
  context-free syntax

"⟨" "*term*:" TRS-TERM ","	
"*occu*:" OCCURRENCE ","	
"*subs*:" SUBSTITUTION ","	
"*rule*:" RULE "⟩"	→ TRUE-STEP
TRUE-STEP	→ STEP
empty-step(TRS-TERM)	→ STEP
STEP+	→ STEPS
STEPS ";" STEPS	→ STEPS {**left**}
"(" STEPS ")"	→ STEPS {**bracket**}
TRUE-STEP "." *term*	→ TRS-TERM
TRUE-STEP "." *occu*	→ OCCURRENCE
TRUE-STEP "." *subs*	→ SUBSTITUTION
TRUE-STEP "." *rule*	→ RULE
TRUE-STEP "." *term* ":=" TRS-TERM	→ TRUE-STEP
TRUE-STEP "." *occu* ":=" OCCURRENCE	→ TRUE-STEP
TRUE-STEP "." *subs* ":=" SUBSTITUTION	→ TRUE-STEP
TRUE-STEP "." *rule* ":=" RULE	→ TRUE-STEP
TRUE-STEP "." *contractum*	→ TRS-TERM

variables

Step [*0-9*]$*$ $\rightarrow$ STEP
TrueStep [*0-9*]$*$ $\rightarrow$ TRUE-STEP
Step [*0-9*]$*$"$*$" $\rightarrow$ STEP$*$
Step [*0-9*]$*$"+" $\rightarrow$ STEP+
Steps [*0-9*]$*$ $\rightarrow$ STEPS

equations

$$Step_1^+ ;\, Step_2^+ = Step_1^+ \; Step_2^+ \qquad [1]$$

$$\langle term\!: `T,\ occu\!: O,\ subs\!: \sigma,\ rule\!: R\rangle \ .\ term = `T \qquad [o1]$$
$$\langle term\!: `T,\ occu\!: O,\ subs\!: \sigma,\ rule\!: R\rangle \ .\ occu = O \qquad [o2]$$
$$\langle term\!: `T,\ occu\!: O,\ subs\!: \sigma,\ rule\!: R\rangle \ .\ subs = \sigma \qquad [o3]$$
$$\langle term\!: `T,\ occu\!: O,\ subs\!: \sigma,\ rule\!: R\rangle \ .\ rule = R \qquad [o4]$$

The contractum is obtained by replacing the term at redex occurrence O with the subsituted right-hand side of the rule R.

$$\langle term\!: `T,\ occu\!: O,\ subs\!: \sigma,\ rule\!: R\rangle \ .\ contractum = `T[O \leftarrow R\ .\ rhs^{\,\sigma}\ ] \qquad [o5]$$

$$\langle term\!: `T,\ occu\!: O,\ subs\!: \sigma,\ rule\!: R\rangle \ .\ term := `T_1 \ = \qquad [a1]$$
$$\langle term\!: `T_1,\ occu\!: O,\ subs\!: \sigma,\ rule\!: R\rangle$$
$$\langle term\!: `T,\ occu\!: O,\ subs\!: \sigma,\ rule\!: R\rangle \ .\ occu := O_1 \ = \qquad [a2]$$
$$\langle term\!: `T,\ occu\!: O_1,\ subs\!: \sigma,\ rule\!: R\rangle$$
$$\langle term\!: `T,\ occu\!: O,\ subs\!: \sigma,\ rule\!: R\rangle \ .\ subs := \sigma_1 \ = \qquad [a3]$$
$$\langle term\!: `T,\ occu\!: O,\ subs\!: \sigma_1,\ rule\!: R\rangle$$
$$\langle term\!: `T,\ occu\!: O,\ subs\!: \sigma,\ rule\!: R\rangle \ .\ rule := R_1 \ = \qquad [a4]$$
$$\langle term\!: `T,\ occu\!: O,\ subs\!: \sigma,\ rule\!: R_1\rangle$$

B.2 Normalizations

B.2.1 Module Matching

Given an (open) pattern t_1 and a term t_2, find a substitution σ such that $t_1^\sigma \equiv t_2$.

imports Substitution(B.1.4)
exports
sorts MATCH
context-free syntax
success(SUBSTITUTION) $\rightarrow$ MATCH
fail $\rightarrow$ MATCH

match pattern TRS-TERM *against* TRS-TERM → MATCH
variables
"μ" [0-9]∗→ MATCH
hiddens
context-free syntax
MATCH "∧" MATCH → MATCH {**left**}

equations

Elementary Matching

A pattern consisting of just a variable matches everything.

$$\textit{match pattern } `V \textit{ against } `T = \textit{success}(\{`V \mapsto `T\}) \quad [\text{m1}]$$

Two equal constants.

$$\textit{match pattern } `F() \textit{ against } `F() = \textit{success}(\{\}) \quad [\text{m2}]$$

Two equal function symbols; match the subterms.

$$\begin{aligned}&\textit{match pattern } `F(`T_1, `T_1^*) \textit{ against } `F(`T_2, `T_2^*) \;= \\ &\;\textit{match pattern } `T_1 \textit{ against } `T_2 \\ &\;\wedge \textit{match pattern } `F(`T_1^*) \textit{ against } `F(`T_2^*)\end{aligned} \quad [\text{m3}]$$

There are no other possibilities for success.

$$\textit{match pattern } `T_1 \textit{ against } `T_2 = \textit{fail} \qquad \textbf{otherwise} \quad [\text{m4}]$$

Composition of Matches

Note that it is important that the two substitutions resulting from the matches do not contain conflicting assignments (in order to be able to deal with non-linearity).

$$\textit{fail} \wedge \mu = \textit{fail} \quad [\text{a1}]$$

$$\mu \wedge \textit{fail} = \textit{fail} \quad [\text{a2}]$$

$$\frac{`V \in \sigma = \textit{true}, \quad \sigma(`V) = `T}{\textit{success}(\{`V \mapsto `T, M^*\}) \wedge \textit{success}(\sigma) = \textit{success}(\{M^*\}) \wedge \textit{success}(\sigma)} \quad [\text{a3}]$$

$$\frac{`V \in \sigma = \textit{true}, \quad \sigma(`V) \neq `T}{\textit{success}(\{`V \mapsto `T, M^*\}) \wedge \textit{success}(\sigma) = \textit{fail}} \quad [\text{a4}]$$

$$\frac{`V \in \sigma_1 = \textit{false}, \quad \textit{success}(\{M^*\}) \wedge \textit{success}(\sigma_1) = \textit{success}(\sigma_2)}{\textit{success}(\{`V \mapsto `T, M^*\}) \wedge \textit{success}(\sigma_1) = \textit{success}(\{`V \mapsto `T\} \circ \sigma_2)} \quad [\text{a5}]$$

$$\textit{success}(\{\}) \wedge \textit{success}(\sigma) = \textit{success}(\sigma) \quad [\text{a6}]$$

B.2.2 Module Find-Redex

Given a term t and a set of rewrite rules, yield a STEP structure with t as its term if one of the rewrite rules applies, or else return the unit-step. The steps are looked for in an outermost fashion.

This specification uses the associative lists of ASF+SDF, see [Hen91, Chapter 3], [Kli93a, Section "The Treatment of List Variables], and Section 2.2.4 of this thesis. *List matching*, used when rewriting over associative lists, may find more than one match, out of which it chooses one non-deterministically. In the precence of conditional rewrite rules, the system will backtrack until it finds a choice staisfying the condition. This mechanism is used to choose an appicable rewrite rule from the rewrite-system (equation [r1]), and to choose a subterm in which a redex occurs (equation [r3]).

imports Reduction-Step$^{(B.1.7)}$ Matching$^{(B.2.1)}$ Rewrite-System$^{(B.1.3)}$
exports
context-free syntax
find-step(TRS-TERM, TRS) → STEP
hiddens
context-free syntax
nr-of-sons(TRS-TERM) → INT
find-step-top(TRS-TERM, TRS) → STEP
find-step-sons(TRS-TERM, TRS) → STEP

equations

Find a Redex at the Top of the Term

$$\frac{\textit{match pattern } R \textit{ . lhs against } {}^{\backprime}T = \textit{success}(\sigma)}{\textit{find-step-top}({}^{\backprime}T, \{R_1^*, R, R_2^*\}) = \langle \textit{term}\text{: } {}^{\backprime}T, \textit{occu}\text{: } [], \textit{subs}\text{: } \sigma, \textit{rule}\text{: } R\rangle} \quad \text{[r1]}$$

$$\textit{find-step-top}({}^{\backprime}T, \text{TRS}) = \textit{empty-step}({}^{\backprime}T) \qquad \textbf{otherwise} \quad \text{[r2]}$$

Find a Redex in one of the Subterms

$$\frac{\begin{array}{c}{}^{\backprime}T_0 = {}^{\backprime}F({}^{\backprime}T_1^*, {}^{\backprime}T, {}^{\backprime}T_2^*),\\ \textit{find-step}({}^{\backprime}T, \text{TRS}) = \langle \textit{term}\text{: } {}^{\backprime}T, \textit{occu}\text{: } O, \textit{subs}\text{: } \sigma, \textit{rule}\text{: } R\rangle\end{array}}{\begin{array}{l}\textit{find-step-sons}({}^{\backprime}T_0, \text{TRS}) \ = \\ \langle \textit{term}\text{: } {}^{\backprime}T_0, \textit{occu}\text{: } [\textit{nr-of-sons}({}^{\backprime}F({}^{\backprime}T_1^*)) + 1]; O, \textit{subs}\text{: } \sigma, \textit{rule}\text{: } R\rangle\end{array}} \quad \text{[r3]}$$

$$\textit{find-step-sons}({}^{\backprime}T, \text{TRS}) = \textit{empty-step}({}^{\backprime}T) \qquad \textbf{otherwise} \quad \text{[r4]}$$

Return a Redex

$$\frac{\textit{find-step-top}(`T, \mathrm{TRS}) = \textit{Step},\ \textit{Step} \neq \textit{empty-step}(`T)}{\textit{find-step}(`T, \mathrm{TRS}) = \textit{Step}} \quad [\mathrm{r5}]$$

$$\textit{find-step}(`T, \mathrm{TRS}) = \textit{find-step-sons}(`T, \mathrm{TRS}) \qquad \textbf{otherwise} \quad [\mathrm{r6}]$$

Number of subterms in a term

$$\textit{nr-of-sons}(`V) = 0 \quad [\mathrm{s1}]$$
$$\textit{nr-of-sons}(`F()) = 0 \quad [\mathrm{s2}]$$
$$\textit{nr-of-sons}(`F(`T, `T^*)) = \textit{nr-of-sons}(`F(`T^*)) + 1 \quad [\mathrm{s3}]$$

B.2.3 Module TRS-Reduce

Repeatedly perform single reduction steps until no further steps are possible.

imports Find-Redex$^{(\mathrm{B.2.2})}$
exports
  context-free syntax
    normalize(TRS-TERM, TRS) → STEPS
    STEPS "." *result* → TRS-TERM

equations

$$\frac{\textit{find-step}(`T_1, \mathrm{TRS}) = \textit{TrueStep}}{\textit{normalize}(`T_1, \mathrm{TRS}) = \textit{TrueStep}; \textit{normalize}(\textit{TrueStep}\ .\ \textit{contractum}, \mathrm{TRS})} \quad [1]$$

$$\textit{normalize}(`T_1, \mathrm{TRS}) = \textit{empty-step}(`T_1) \qquad \textbf{otherwise} \quad [2]$$

$$\textit{Step}^*\ \textit{empty-step}(`T_1)\ .\ \textit{result} = `T_1 \quad [3]$$

B.3 Origins

B.3.1 Module Occ-Sets

Origins are represented as sets of Occurrences.
imports Occurrences$^{(\mathrm{B.1.6})}$
exports
  sorts OCC-SET
  context-free syntax
    "{" {OCCURRENCE ","}* "}" → OCC-SET

OCC-SET "∪" OCC-SET	→ OCC-SET	**{left}**
OCCURRENCE ";" OCC-SET	→ OCC-SET	
OCC-SET ";" OCCURRENCE	→ OCC-SET	
"(" OCC-SET ")"	→ OCC-SET	**{bracket}**
OCCURRENCE "∈" OCC-SET	→ BOOL	
"$\mathcal{O}_{var}$"(TRS-TERM)	→ OCC-SET	
"$\mathcal{O}_{fun}$"(TRS-TERM)	→ OCC-SET	
"$\mathcal{O}_{all}$"(TRS-TERM)	→ OCC-SET	
find every TRS-TERM *in* TRS-TERM	→ OCC-SET	
shift-right(OCC-SET)	→ OCC-SET	

variables

Os [*0-9*]∗"∗" → {OCCURRENCE ","}∗
O-Set [*0-9*]∗ → OCC-SET

priorities

{ OCCURRENCE ";"OCC-SET → OCC-SET } >
{ OCC-SET ";"OCCURRENCE → OCC-SET, OCC-SET "∪"OCC-SET → OCC-SET }

equations

Set Union

$$\{Os_1^*\} \cup \{Os_2^*\} = \{Os_1^*, Os_2^*\} \qquad \text{[un1]}$$

$$O \in \{Os_1^*, O, Os_2^*\} = \mathit{true} \qquad \text{[el1]}$$

$$O \in \{Os^*\} = \mathit{false} \qquad \textbf{otherwise} \qquad \text{[el2]}$$

Set Prepending and Appending

Prepend a path to all elements of an occurrence set.

$$O; \{\} = \{\} \qquad \text{[pr1]}$$

$$O_1; \{O_2, Os^*\} = \{O_1; O_2\} \cup O_1; \{Os^*\} \qquad \text{[pr2]}$$

Append a path to all elements of an occurrence set.

$$\{\}; O = \{\} \qquad \text{[ap1]}$$

$$\{O_1, Os^*\}; O_2 = \{Os^*\}; O_2 \cup \{O_1; O_2\} \qquad \text{[ap2]}$$

Moving rightwards

$$\mathit{shift\text{-}right}(\{\}) = \{\} \qquad \text{[sh1]}$$

$$\mathit{shift\text{-}right}(\{[N, u], Os^*\}) = \{[N+1, u]\} \cup \mathit{shift\text{-}right}(\{Os^*\}) \qquad \text{[sh2]}$$

$$\mathit{shift\text{-}right}(\{[], Os^*\}) = \{[]\} \cup \mathit{shift\text{-}right}(\{Os^*\}) \qquad \text{[sh3]}$$

Various sets of occurrences.

$\mathcal{O}_{all}(`V) = \{[]\}$ [a1]

$\mathcal{O}_{all}(`F()) = \{[]\}$ [a2]

$\mathcal{O}_{all}(`F(`T, `T^*)) = [1]; \mathcal{O}_{all}(`T) \cup \textit{shift-right}(\mathcal{O}_{all}(`F(`T^*)))$ [a3]

$\mathcal{O}_{var}(`V) = \{[]\}$ [a4]

$\mathcal{O}_{var}(`F()) = \{\}$ [a5]

$\mathcal{O}_{var}(`F(`T, `T^*)) = [1]; \mathcal{O}_{var}(`T) \cup \textit{shift-right}(\mathcal{O}_{var}(`F(`T^*)))$ [a6]

$\mathcal{O}_{fun}(`V) = \{\}$ [a7]

$\mathcal{O}_{fun}(`F()) = \{[]\}$ [a8]

$\mathcal{O}_{fun}(`F(`T, `T^*)) = [1]; \mathcal{O}_{fun}(`T) \cup \textit{shift-right}(\mathcal{O}_{fun}(`F(`T^*)))$ [a9]

All equal subterms

$\textit{find every } `T \textit{ in } `T = \{[]\}$ [e1]

$$\frac{`T_1 \neq `T_2,\quad `T_2 = `F(`T_{11}, `T_{11}^*)}{\textit{find every } `T_1 \textit{ in } `T_2 = [1]; \textit{find every } `T_1 \textit{ in } `T_{11} \cup \textit{shift-right}(\textit{find every } `T_1 \textit{ in } `F(`T_{11}^*))}$$ [e2]

$\textit{find every } `T_1 \textit{ in } `T_2 = \{\}$ **otherwise** [e3]

B.3.2 Module Occ-Maps

An origin will be represented by a mapping between occurrences and sets of occurrences.

imports Occ-Sets(B.3.1)
exports
sorts OCC-MAP OCC-MAP-ENTRY
context-free syntax

OCCURRENCE "$\mapsto$" OCC-SET	$\rightarrow$ OCC-MAP-ENTRY	
"{" {OCC-MAP-ENTRY ","}* "}"	$\rightarrow$ OCC-MAP	
OCC-MAP "$\cup$" OCC-MAP	$\rightarrow$ OCC-MAP	{**left**}
"(" OCC-MAP ")"	$\rightarrow$ OCC-MAP	{**bracket**}
identity-map(OCC-SET)	$\rightarrow$ OCC-MAP	
value of OCCURRENCE *in* OCC-MAP	$\rightarrow$ OCC-SET	
values of OCC-SET *in* OCC-MAP	$\rightarrow$ OCC-SET	
OCC-MAP "@" OCC-MAP	$\rightarrow$ OCC-MAP	

variables

Ms [0-9]∗“∗” → {OCC-MAP-ENTRY “,”}∗
$O\text{-}Map$ [0-9]∗→ OCC-MAP

equations

$$\{Ms_1^*\} \cup \{Ms_2^*\} = \{Ms_1^*, Ms_2^*\} \quad \text{[ma1]}$$

$$identity\text{-}map(\{\}) = \{\} \quad \text{[ma2]}$$
$$identity\text{-}map(\{O, Os^*\}) = \{O \mapsto \{O\}\} \cup identity\text{-}map(\{Os^*\}) \quad \text{[ma3]}$$

$$value\ of\ O\ in\ \{Ms_1^*, O \mapsto O\text{-}Set, Ms_2^*\} = O\text{-}Set \quad \text{[co1]}$$

$$values\ of\ \{\}\ in\ O\text{-}Map = \{\} \quad \text{[co2]}$$
$$values\ of\ \{O, Os^*\}\ in\ O\text{-}Map = value\ of\ O\ in\ O\text{-}Map \cup values\ of\ \{Os^*\}\ in\ O\text{-}Map \quad \text{[co3]}$$

$$O\text{-}Map\ @\ \{\} = \{\} \quad \text{[co4]}$$
$$O\text{-}Map\ @\ \{O \mapsto O\text{-}Set, Ms^*\} = \{O \mapsto values\ of\ O\text{-}Set\ in\ O\text{-}Map\} \cup O\text{-}Map\ @\ \{Ms^*\} \quad \text{[co5]}$$

B.3.3 Module Org-Step

With each rewrite step, an origin function is associated mapping an occurrence in the result term to a set of occurrences in the term to be rewritten. In this specification this is modelled by function “primary-step” taking both a step and an occurrence, and yielding the sets of occurrences related to this occurrence.

The function “primary-step” is split into various subfunctions, representing the origins caused by the various cases, such as “Common Variables”, “Common Subterms”, etc. Splitting the function into several subfunctions allows easy experimentation with alternative compositions.

imports Occ-Maps(B.3.2) Reduction-Step(B.1.7)
exports
context-free syntax
primary-step “_” STEP “(” OCCURRENCE “)” → OCC-SET
secondary-step “_” STEP “(” OCCURRENCE “)” → OCC-SET

org-context “_” STEP “(” OCCURRENCE “)” → OCC-SET
org-vars “_” STEP “(” OCCURRENCE “)” → OCC-SET
org-subterms “_” STEP “(” OCCURRENCE “)” → OCC-SET
org-red-contr “_” STEP “(” OCCURRENCE “)” → OCC-SET
variables
S [0-9]∗→ TRUE-STEP

equations

One Primary Step

$$primary\text{-}step_S\ ([v]) \quad = org\text{-}context_S\ ([v]) \cup org\text{-}vars_S\ ([v]) \qquad [o1]$$

$$secondary\text{-}step_S\ ([v]) = primary\text{-}step_S\ ([v]) \cup org\text{-}subterms_S\ ([v]) \cup org\text{-}red\text{-}contr_S\ ([v]) \qquad [o2]$$

Context

$$\frac{\begin{array}{r} S\ .\ occu = [u], \\ [v] \prec [u] \vee [v] \mid [u] = true \end{array}}{org\text{-}context_S\ ([v]) = \{[v]\}} \qquad [o3]$$

$$org\text{-}context_S\ (O) = \{\} \qquad \textbf{otherwise} \qquad [o4]$$

Common Variables

$$\frac{\begin{array}{r} S\ .\ occu = [u], \\ [v'] \in \mathcal{O}_{var}(S\ .\ rule\ .\ rhs) = true, \\ find\ every\ S\ .\ rule\ .\ rhs\ /\ [v']\ in\ S\ .\ rule\ .\ lhs = O\text{-}Set \end{array}}{org\text{-}vars_S\ ([u, v', w]) = [u]; O\text{-}Set; [w]} \qquad [o5]$$

$$org\text{-}vars_S\ (O) = \{\} \qquad \textbf{otherwise} \qquad [o6]$$

Common Subterms

$$\frac{\begin{array}{r} S\ .\ occu = [u], \\ [v'] \in \mathcal{O}_{fun}(S\ .\ rule\ .\ rhs) = true, \\ find\ every\ S\ .\ rule\ .\ rhs\ /\ [v']\ in\ S\ .\ rule\ .\ lhs = O\text{-}Set \end{array}}{org\text{-}subterms_S\ ([u, v']) = [u]; O\text{-}Set} \qquad [o7]$$

$$org\text{-}subterms_S\ (O) = \{\} \qquad \textbf{otherwise} \qquad [o8]$$

Redex - Contractum

$$org\text{-}red\text{-}contr_S\ ([v]) = \{[v]\} \Leftarrow S\ .\ occu = [v] \qquad [o9]$$

$$org\text{-}red\text{-}contr_S\ ([v]) = \{\} \qquad \textbf{otherwise} \qquad [o10]$$

B.3.4 Module Phi-Functions

Define functions which allow the partitioning of a TRS into a Primitive Recursive Scheme (PRS) (see Chapter 7 and [Meu94, CF82]). A PRS formalizes sets of functions that perform an inductive pass over some structure. These funtions are refered to as Φ-functions.

imports Rewrite-Rule$^{(B.1.2)}$ Booleans$^{(2.1)}$ Occ-Maps$^{(B.3.2)}$
exports
sorts PHI-SYMBOL
context-free syntax

"ϕ" "_" FUN-SYMBOL	$\rightarrow$ PHI-SYMBOL
PHI-SYMBOL	$\rightarrow$ FUN-SYMBOL
"*ϕ-defining-equation?*"(RULE)	$\rightarrow$ BOOL
"$\mathcal{O}_\Phi$"(TRS-TERM)	$\rightarrow$ OCC-SET
"$\mathcal{O}_{non\text{-}\Phi}$"(TRS-TERM)	$\rightarrow$ OCC-SET

variables

"ϕ"	$\rightarrow$ PHI-SYMBOL
"g"	$\rightarrow$ FUN-SYMBOL
"x"[*0-9*]$*$"+"	$\rightarrow$ {TRS-TERM ","}+
"y"[*0-9*]$*$"$*$"	$\rightarrow$ {TRS-TERM ","}$*$
"τ"[*0-9*]$*$	$\rightarrow$ TRS-TERM

equations

Characterize ϕ-defining Equations

$$\textit{\(\phi\)-defining-equation?}(\phi(g(x^+), y^*) \rightarrow \tau) = \textit{true} \quad \text{[phi1]}$$

$$\textit{\(\phi\)-defining-equation?}(\tau_1 \rightarrow \tau_2) = \textit{false} \quad \textbf{otherwise} \quad \text{[phi2]}$$

Various Sets of Occurrences

$$\mathcal{O}_\Phi(\phi()) = \{[]\} \quad \text{[phi3]}$$

$$\mathcal{O}_\Phi(`F()) = \{\} \quad \textbf{otherwise} \quad \text{[phi4]}$$

$$\mathcal{O}_\Phi(`V) = \{\} \quad \text{[phi5]}$$

$$\mathcal{O}_\Phi(`F(`T, `T^*)) = [1]; \mathcal{O}_\Phi(`T) \cup \textit{shift-right}(\mathcal{O}_\Phi(`F(`T^*))) \quad \text{[phi6]}$$

$$\mathcal{O}_{non\text{-}\Phi}(\phi()) = \{\} \quad \text{[phi7]}$$

$$\mathcal{O}_{non\text{-}\Phi}(`F()) = \{[]\} \quad \textbf{otherwise} \quad \text{[phi8]}$$

$$\mathcal{O}_{non\text{-}\Phi}(`V) = \{\} \quad \text{[phi9]}$$

$$\mathcal{O}_{non\text{-}\Phi}(`F(`T, `T^*)) = [1]; \mathcal{O}_{non\text{-}\Phi}(`T) \cup \textit{shift-right}(\mathcal{O}_{non\text{-}\Phi}(`F(`T^*))) \quad \text{[phi10]}$$

B.3.5 Module Org-Step-PRS

imports Org-Step(B.3.3) Phi-Functions(B.3.4)
exports
context-free syntax
prs-step "_" STEP "(" OCCURRENCE ")" → OCC-SET
org-prs "_" STEP "(" OCCURRENCE ")" → OCC-SET

equations

Origins in PRSs

$$\textit{prs-step}_S\ ([v]) = \textit{org-prs}_S\ ([v]) \cup \textit{primary-step}_S\ ([v]) \qquad \text{[p1]}$$

Φ-Functions

$$\frac{\begin{array}{c} S\ .\ \textit{occu} = [u], \\ [v'] \in \mathcal{O}_{\Phi}(S\ .\ \textit{rule}\ .\ \textit{rhs}) = \textit{true} \end{array}}{\textit{org-prs}_S\ ([u, v']) = \textit{prs-step}_S\ ([u, v', 1])} \qquad \text{[p2]}$$

Synthesizers

$$\frac{\begin{array}{c} S\ .\ \textit{occu} = [u], \\ [v'] \in \mathcal{O}_{non\text{-}\Phi}(S\ .\ \textit{rule}\ .\ \textit{rhs}) = \textit{true}, \\ \phi\textit{-defining-equation?}(S\ .\ \textit{rule}) = \textit{true} \end{array}}{\textit{org-prs}_S\ ([u, v']) = \{[u, 1]\}} \qquad \text{[p3]}$$

Auxiliary Symbols

$$\frac{\begin{array}{c} S\ .\ \textit{occu} = [u], \\ [v'] \in \mathcal{O}_{non\text{-}\Phi}(S\ .\ \textit{rule}\ .\ \textit{rhs}) = \textit{true}, \\ \phi\textit{-defining-equation?}(S\ .\ \textit{rule}) = \textit{false} \end{array}}{\textit{org-prs}_S\ ([u, v']) = \{[u]\}} \qquad \text{[p4]}$$

Rest

$$\textit{org-prs}_S\ ([v]) = \{\} \qquad \textbf{otherwise} \qquad \text{[p5]}$$

B.3.6 Module Org-Seq

This module merely conatenates the single-step origin functions to a multistep map.

imports Occ-Maps[(B.3.2)] Org-Step-PRS[(B.3.5)]
exports
  sorts SCHEME
  context-free syntax
    "*org-map*" "_" STEP "(" SCHEME ")" → OCC-MAP
    "*org-seq*" "_" STEPS "(" SCHEME ")" → OCC-MAP

    prs-origins → SCHEME
    primary-origins → SCHEME
    secondary-origins → SCHEME
hiddens
  context-free syntax
    org-map(STEP, OCC-SET, SCHEME) → OCC-MAP
    org-seq(STEPS, OCC-SET, SCHEME) → OCC-MAP
    one-step(STEP, OCCURRENCE, SCHEME) → OCC-SET
  variables
    "Σ"→ SCHEME

equations

Full maps for single steps

$$\textit{org-map}_{\textit{empty-step}(`T)}\ (\Sigma) = \textit{identity-map}(\mathcal{O}_{all}(`T)) \quad \text{[a0]}$$

$$\textit{org-map}_S\ (\Sigma) = \textit{org-map}(S, \mathcal{O}_{all}(S\ .\ \textit{contractum}), \Sigma) \quad \text{[a1]}$$

$$\textit{org-map}(S, \{\}, \Sigma) = \{\} \quad \text{[a2]}$$

$$\textit{org-map}(S, \{O, Os^*\}, \Sigma) = \{O \mapsto \textit{one-step}(S, O, \Sigma)\} \cup \textit{org-map}(S, \{Os^*\}, \Sigma) \quad \text{[a3]}$$

$$\textit{one-step}(S, O, \textit{prs-origins}) = \textit{prs-step}_S\ (O) \quad \text{[a4]}$$

$$\textit{one-step}(S, O, \textit{primary-origins}) = \textit{primary-step}_S\ (O) \quad \text{[a5]}$$

$$\textit{one-step}(S, O, \textit{secondary-origins}) = \textit{secondary-step}_S\ (O) \quad \text{[a6]}$$

Full maps for sequences

$$\textit{org-seq}_{Step}\ (\Sigma) = \textit{org-map}_{Step}\ (\Sigma) \quad \text{[a4]}$$

$$\textit{org-seq}_{Step_1^+\ Step}\ (\Sigma) = \textit{org-seq}_{Step_1^+}\ (\Sigma)\ @\ \textit{org-map}_{Step}\ (\Sigma) \quad \text{[a5]}$$

B.4 Examples

B.4.1 Appending Lists

```
org-seq _
normalize(
  rev( cons(one(), cons(two(), nil()) ) ),
  {  rev( nil()     )         -> nil(),
     rev( cons(E,L) )         -> append( rev(L), cons(E,nil()) ),
     append( nil(), L )       -> L,
     append( cons(E,L1), L2 ) -> cons(E, append(L1,L2) )   }
) ( secondary-origins )
```

The map computed by this term is the following:

$$\begin{aligned} &\{[1] \mapsto \{[1,2,1]\},\\ &\ [2,1] \mapsto \{[1,1]\},\\ &\ [2,2] \mapsto \{\},\\ &\ [2] \mapsto \{\},\\ &\ [] \mapsto \{[]\}\} \end{aligned}$$

This mapping reflects the relations between the original term

```
rev(cons(one(), cons(two(), nil())))
```

and the corresponding result

```
cons(two(), cons(one(), nil()))
```

B.4.2 Simple Translation

The second example consists of the translation equations given in Figures 7.1, 7.2 and 7.3:

```
normalize(
  phi_tr-exp(  plus( const(four()), const(three()))),
  {  phi_tr-exp( const(N) )     -> cons( push(N), null()),
     phi_tr-exp( plus(E1,E2) )  ->
      app(     phi_tr-exp(E1),
       app(    phi_tr-exp(E2),
        cons(  add(),
         null() ))) ,
     app( null(), L1 )     -> L1,
     app( cons(E,L1), L2 ) ->  cons(E, app(L1,L2) )  }
) . result
```

If we just execute this `normalize` function, we obtain the result:

```
cons ( push ( four (  ) ),
       cons ( push ( three (  ) ),
              cons ( add (  ),
                     null (  ) ) ) ) )
```

The primary origins that can be computed for this term are (see also Figure 7.4):

$\{[1, 1] \mapsto \{[1, 1, 1]\},$
$[1] \mapsto \{\},$
$[2, 1, 1] \mapsto \{[1, 2, 1]\},$
$[2, 1] \mapsto \{\},$
$[2, 2, 1] \mapsto \{\},$
$[2, 2, 2] \mapsto \{\},$
$[2, 2] \mapsto \{\},$
$[2] \mapsto \{\},$
$[] \mapsto \{\}\}$

The PRS origins computed for this term are (see also Figure 7.5):

$\{[1, 1] \mapsto \{[1, 1, 1]\},$
$[1] \mapsto \{[1, 1]\},$
$[2, 1, 1] \mapsto \{[1, 2, 1]\},$
$[2, 1] \mapsto \{[1, 2]\},$
$[2, 2, 1] \mapsto \{[1]\},$
$[2, 2, 2] \mapsto \{[1]\},$
$[2, 2] \mapsto \{[1]\},$
$[2] \mapsto \{[1]\},$
$[] \mapsto \{[1]\}\}$

Bibliography

[ACCL90] M. Abadi, L. Cardelli, P.-L. Currien, and J.-J. Lévy. Explicit substitutions. In *Proceedings of the 17th conference on Principles of Programming Languages*, pages 31–46, 1990.

[AD92] B.R.T. Arnold and A. van Deursen. Algebraic specification of a language defining interest rate products. CWI, Amsterdam; ORFIS International, Huis ter Heide. Confidential Document., 1992.

[AG92] B.R.T Arnold and H. Gouw. Internal draft papers on RPM. ORFIS International, Huis ter Heide, The Netherlands, 1992.

[AH87] S. Abramsky and C. Hankin, editors. *Abstract Interpretation of Declarative Languages*. Ellis Horwood Limited, 1987.

[Apt81] K.R. Apt. Ten years of Hoare's logic: A survey — Part I. *ACM Transactions on Programming Languages and Systems*, 3(4):431–483, 1981.

[ASU86] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers. Principles, Techniques and Tools*. Addison-Wesley, 1986.

[Bak80] J.W. de Bakker. *Mathematical Theory of Program Correctness*. Prentice-Hall, 1980.

[Bar84] H.P. Barendregt. *The Lambda Calculus; its Syntax and Semantics*, volume 103 of *Studies in Logic and the Foundations of Mathatematics*. North-Holland, 1984.

[BBK87] J.C.M. Baeten, J.A. Bergstra, and J.W. Klop. Term rewriting systems with priorities. In P. Lescanne, editor, *Rewriting Techniques and Applications*, volume 256 of *LNCS*, pages 83–94, 1987.

[BC85] M. Bidoit and C. Choppy. ASSPEGIQUE: an integrated environment for algebraic specifications. In H. Ehrig, C. Floyd, M. Nivat, and J. Thatcher, editors, *Formal Methods and Software Development - Proceedings of the International Joint Conference on Theory and Practice of Software Development 2*, volume 186 of *LNCS*, pages 246–260. Springer-Verlag, 1985.

[BCD+89] P. Borras, D. Clément, Th. Despeyroux, J. Incerpi, B. Lang, and V. Pascual. CENTAUR: the system. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 14–24, 1989. Appeared as *SIGPLAN Notices* 24(2).

[Ber90] Y. Bertot. Implementation of an interpreter for a parallel language in Centaur. In N. Jones, editor, *ESOP '90 - Proceedings of the Third European Symposium on Programming*, volume 432 of *LNCS*, pages 57–69. Springer-Verlag, 1990.

[Ber91a] D. Berry. *Generating Program Animators from Programming Language Semantics*. PhD thesis, University of Edinburgh, 1991.

[Ber91b] Y. Bertot. *Une Automatisation du Calcul des Résidus en Sémantique Naturelle*. PhD thesis, INRIA, Sophia-Antipolis, 1991. In French.

[Ber92] Y. Bertot. Origin functions in lambda-calculus and term rewriting systems. In J.-C. Raoult, editor, *17th Colloquium on Trees in Algebra and Programming (CAAP '92)*, volume 581 of *LNCS*. Springer-Verlag, 1992.

[Ber93] Y. Bertot. A canonical calculus of residuals. In G. Huet and G. Plotkin, editors, *Logical Environments*. Cambridge University Press, 1993.

[BG77] R.M. Burstall and J.A. Goguen. Putting theories together to make specifications. In *Proceedings of the 5th International Join Conference on Artificial Intelligence*, pages 1045–1058. Massachusetts Institute of Technology, 1977.

[BGM89] M. Bidoit, M.-C. Gaudel, and A. Mauboussin. How to make algebraic specifications more understandable: An experiment with the PLUSS specification language. *Science of Computer Programming*, 12:1–38, 1989.

[BGV92] R.A. Ballance, S.L. Graham, and M.L. van de Vanter. The Pan language-based editing system. *ACM Transactions on Software Engineering Methodology*, 1(1):95–127, 1992.

[BHK89] J.A. Bergstra, J. Heering, and P. Klint, editors. *Algebraic Specification*. ACM Press Frontier Series. The ACM Press in co-operation with Addison-Wesley, 1989.

[BJ82] D. Bjørner and C.B. Jones. *Formal Specification and Software Development*. Prentice-Hall, 1982.

[BK86] J.A. Bergstra and J.W. Klop. Conditional rewrite rules: confluence and termination. *Journal of Computer and System Sciences*, 32(3):323–362, 1986.

[BMW92] D.F. Brown, H. Moura, and D.A. Watt. Actress: Action semantics directed compiler generator. In U. Kastens and P. Pfahler, editors, *Proceedings of the 4th International Workshop on Compiler Construction CC'92*, number 641 in LNCS, pages 95–109. Springer-Verlag, 1992.

[BR88] W. Bousdira and J.L. Rémy. REVEUR4: a laboratory for conditional rewriting. In S. Kaplan and J.-P. Jouannaud, editors, *First International Workshop on Conditional Term Rewriting Systems*, volume 308 of *LNCS*, pages 253–257. Springer-Verlag, 1988.

[Bra93] M.G.J. van den Brand. Prettyprinting without losing comments. Report P9315, University of Amsterdam, 1993. Available by *ftp* from ftp.cwi.nl:/pub/gipe as Bra93.ps.Z.

[Bro92] M. Broy. Experiences with software specification and verification using LP, the Larch Proof Assistant. Technical Report 93, DEC Systems Research Center, 1992. Available by *ftp* from gatekeeper.pa.dec.com: /pub/DEC/SRC/research-reports.

[BS86] R. Bahlke and G. Snelting. The PSG system: from formal language definitions to interactive programming environments. *ACM Transactions on Programming Languages and Systems*, 8(4):547–576, 1986.

[BWP87] M. Broy, M. Wirsing, and P. Pepper. On the algebraic definition of programming languages. *ACM Transactions on Programming Languages and Systems*, 9(1):54–99, 1987.

[CF82] B. Courcelle and P. Franchi-Zannettacci. Attribute grammars and recursive program schemes I and II. *Theoretical Computer Science*, 17:163–191 and 235–257, 1982.

[Chu40] A. Church. A formulation of a Simple Theory of Types. *Journal of Symbolic Logic*, 5:56–68, 1940.

[Cou90] B. Courcelle. Recursive applicative program schemes. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, Volume B*, pages 459–492. Elsevier Science Publishers, 1990.

[DD94] A. van Deursen and T.B. Dinesh. Origin tracking for higher-order term rewriting systems. In J. Heering, K. Meinke, B. Möller, and T. Nipkow, editors, *Higher-Order Algebra, Logic and Term Rewriting (HOA '93)*, volume 816 of *LNCS*, pages 76–95. Springer-Verlag, 1994.

[Des88] Th. Despeyroux. Typol: a formalism to implement natural semantics. Technical Report 94, INRIA, 1988.

[Deu91] A. van Deursen. An algebraic specification for the static semantics of Pascal. Technical Report CS-R9129, Centrum voor Wiskunde en Informatica (CWI), Amsterdam, 1991. Full specification available by *ftp* from ftp.cwi.nl: pub/gipe/spec.

[Deu92] A. van Deursen. Specification and generation of a λ-calculus environment. In J.L.G. Dietz, editor, *Conference Proceedings Computing Science in the Netherlands CSN'92*, pages 14–26, 1992.

[Deu94] A. van Deursen. Origin tracking in primitive recursive schemes. Technical Report CS-R9401, Centrum voor Wiskunde en Informatica (CWI), Amsterdam, 1994. Earlier version appeared in H.A. Wijshoff (Ed.), *Conference Proceedings Computing Science in the Netherlands CSN'93*.

[Din93] T.B. Dinesh. Type checking revisited: Modular error handling. In D.J. Andrews, J.F. Groote, and C.A. Middelburg, editors, *Semantics of Specification Languages*, Workshops in Computing, pages 216–231. Springer-Verlag, 1993.

[DKT93] A. van Deursen, P. Klint, and F. Tip. Origin tracking. *Journal of Symbolic Computation*, 15:523–545, 1993. Special Issue on Automatic Programming.

[DM94] A. van Deursen and P. D. Mosses. *ASD: The Action Semantic Description Tools. The Implementation. Version 2.04*. Computer Science Department, Aarhus University, March 1994. Available by *ftp*: ftp.daimi.aau.dk, pub/action/sytstems/ASD-2.04.tar.Z, file ASD/doc/implementation.dvi.

[DS93] K.-G. Doh and D.A. Schmidt. Action semantics-directed prototyping. *Computer Languages*, 19(4):213–233, 1993.

[DT92] T.B. Dinesh and F. Tip. Animators and error reporters for generated environments. Technical Report CS-R9253, Centrum voor Wiskunde en Informatica (CWI), Amsterdam, 1992. Available by *ftp* from ftp.cwi.nl:/pub/gipe.

[Duk87] R. Duke. Predicate rules for Pascal static semantics. Technical Report 86, The University of Queensland, Department of Computer Science, September 1987.

[EG92] Th. Eggenschwiler and E. Gamma. ET++ SwapsManager: Using object technology in the financial engineering domain. In *OOPSLA'92 Seventh Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 166–177. ACM, 1992. SIGPLAN Notices 27(10).

[Eke92] S.M. Eker. A comparison of OBJ3 and ASF+SDF. Report CS-R9223, Centrum voor Wiskunde en Informatica (CWI), Amsterdam, 1992.

[EM85] H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specifications, vol. I, Equations and Initial Semantics*. Springer-Verlag, 1985.

[FGJM85] K. Futatsugi, J.A. Goguen, J.-P. Jouannaud, and J. Meseguer. Principles of OBJ2. In B. Reid, editor, *Conference Record of the Twelfth Annual ACM Symposium on Principles of Programming Languages*, pages 52–66. ACM, 1985.

[Fie92] J. Field. A simple rewriting semantics for realistic imperative programs and its application to program analysis. In *Proc. ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, pages 98–107, San Francisco, June 1992. Published as Yale University Technical Report YALEU/DCS/RR–909.

[FT94] J. H. Field and F. Tip. Dynamic dependence in term rewriting systems and its application to program slicing. Technical report, Centrum voor Wiskunde en Informatica (CWI), 1994. To appear.

[GG89] S.J. Garland and J.V. Guttag. An overview of LP, the Larch Prover. In N. Dershowitz, editor, *Proceedings of the 3rd International Conference on Rewriting Techniques and Applictions RTA 89*, number 355 in LNCS, pages 137–151. Springer-Verlag, 1989.

[GH78] J.V. Guttag and J.J. Horning. The algebraic specification of abstract data types. *Acta Informatica*, 10:27–52, 1978.

[Gor88] M.J.C. Gordon. *Programming Language Theory and its Implementation*. Prentice-Hall, 1988.

[GTW78] J.A. Goguen, J.W. Thatcher, and E.G. Wagner. An initial algebra approach to the specification, correctness and implementation of abstract data types. In R. Yeh, editor, *Current Trends in Programming Methodology*, pages 80–149. Prentice-Hall, 1978.

[GTWW77] J.A. Goguen, J.W. Thatcher, E.G. Wagner, and J.B. Wright. Initial algebra semantics and continuous algebras. *Journal of the ACM*, 24(1):68–95, 1977.

[GWM+92] J.A. Goguen, T. Winkler, J. Meseguer, K. Futatsugi, and J.P. Jouannaud. Introducing OBJ. In J.A. Goguen, D. Coleman, and R. Gallimore, editors, *Applications of Algebraic Specification Using OBJ*. Cambridge University Press, 1992. To Appear.

[Han87] H. Hansen. The ACT-system: Experiences and future enhancements. In D.T. Sannella and A. Tarlecki, editors, *Recent Trends in Data Type Specifications WADT'87*, volume 332 of *LNCS*, pages 111–130. Springer-Verlag, 1987.

[Har93] T. Hardin. How to get confluence for explicit substitutions. In M. R. Sleep, M. J. Plasmeijer, and M. C. J. D. van Eekelen, editors, *Term Graph Rewriting*, pages 31–46. John Wiley & Sons, 1993.

[Hee92] J. Heering. Second-order algebraic specification of static semantics. Technical Report CS-R9254, Centrum voor Wiskunde en Informatica (CWI), 1992. Extented version to appear, 1994.

[Hen88] P.R.H. Hendriks. ASF system user's guide. Report CS-R8823, Centrum voor Wiskunde en Informatica (CWI), Amsterdam, 1988. Extended abstract in: Conference Proceedings of Computing Science in the Netherlands, CSN'88 1, pp. 83-94, SION (1988).

[Hen91] P.R.H. Hendriks. *Implementation of Modular Algebraic Specifications*. PhD thesis, University of Amsterdam, 1991.

[HHKR89] J. Heering, P.R.H. Hendriks, P. Klint, and J. Rekers. The syntax definition formalism SDF - reference manual. *SIGPLAN Notices*, 24(11):43–75, 1989. Most recent version available by *ftp* from ftp.cwi.nl, pub/gipe, SDF-manual.ps.Z.

[HK94] J. Heering and P. Klint. Prehistory of the ASF+SDF system (1980-1984). Available by *ftp*, pub/gipe, 1994.

[HKR90] J. Heering, P. Klint, and J. Rekers. Incremental generation of parsers. *IEEE Transactions on Software Engineering*, 16(12):1344–1351, 1990. Also in: *SIGPLAN Notices*, 24(7):179-191, 1989.

[HKR92] J. Heering, P. Klint, and J. Rekers. Incremental generation of lexical scanners. *ACM Transactions on Programming Languages and Systems*, 14(4):490–520, 1992.

[HKR94] J. Heering, P. Klint, and J. Rekers. Lazy and incremental program generation. *ACM Transactions on Programming Languages and Systems*, 16(3), May 1994.

[HL91] G. Huet and J.-J. Lévy. Computations in orthogonal rewriting systems part I and II. In J.-L. Lassez and G. Plotkin, editors, *Computational Logic; essays in honour of Alan Robinson*, pages 395–443. MIT Press, 1991.

[HN86] A. N. Habermann and D. Notkin. Gandalf: software development environments. *IEEE Transactions on Software Engineering*, 12(12):1117–1127, 1986.

[Hus88] H. Hussmann. The Passau RAP system: rapid prototyping for algebraic specifications. In S. Kaplan and J.-P. Jouannaud, editors, *Proceedings of the First International Workshop on Conditional Term Rewriting Systems*, volume 308 of *LNCS*, pages 264–265. Springer-Verlag, 1988.

[HW73] C.A.R. Hoare and N. Wirth. An axiomatic definition of the programming language Pascal. *Acta Informatica*, 2:335–355, 1973.

[ISO83] International organization for standardization. *International standard ISO 7185, Programming languages - PASCAL, Ref. No. 7185-1983(E)*, first edition, 1983.

[Kah87] G. Kahn. Natural semantics. In F.J. Brandenburg, G. Vidal-Naquet, and M. Wirsing, editors, *Fourth Annual Symposium on Theoretical Aspects of Computer Science*, volume 247 of *LNCS*, pages 22–39. Springer-Verlag, 1987.

[Kah93] S. Kahrs. Mistakes and ambiguities in the definition of Standard ML. Technical Report ECS-LFCS-93-257, University of Edinburgh, 1993.

[Kam94] S. Kamin, editor. Report on a workshop on future directions in programming languages and compilers. Available by World Wide Web, http://www.cs.uiuc.edu/CS_INFO_SERVER/DEPT_INFO/CS_FACULTY/FAC_HTMLS/kamin.html, May 1994.

[Kap88] S. Kaplan. Positive/negative conditional rewriting. In S. Kaplan and J.-P. Jouannaud, editors, *Conditional Term Rewriting Systems*, volume 308 of *LNCS*, pages 129–143. Springer-Verlag, 1988.

[Kas93] U. Kastens. Executable specifications for language implementation. In M. Bruynhooge and J. Penjam, editors, *Pogramming Language Implementation and Logic Progarmming PLILP 93*, volume 714 of *LNCS*, pages 1–11. Springer-Verlag, 1993.

[KB93] J.W.C. Koorn and H.C.N. Bakker. Building an editor from existing components: an exercise in software re-use. Report P9312, Programming Research Group, University of Amsterdam, 1993. Available by *ftp* from ftp.cwi.nl:/pub/gipe as KB93.ps.Z.

[KHZ82] U. Kastens, B. Hutt, and E. Zimmermann. *GAG: A Practical Compiler Generator*, volume 141 of *LNCS*. Springer Verlag, 1982.

[Kli80] P. Klint. An overview of the SUMMER programming language. In *Proceedings of the Seventh International Symposium on the Principles of Programming Languages, POPL'80*, pages 47–55, 1980.

[Kli85] P. Klint. *A Study in String Processing Languages*, volume 205 of *LNCS*. Springer-Verlag, 1985.

[Kli93a] P. Klint, editor. *The ASF+SDF Meta-environment User's Guide,* version 26 February. Centrum voor Wiskunde en Informatica (CWI), 1993. Available by *ftp* from ftp.cwi.nl:/pub/gipe as SysManual.ps.Z.

[Kli93b] P. Klint. Design of a fixed exchange format for ASF+SDF. CWI, Department of Software Technology., 1993. Technical Note.

[Kli93c] P. Klint. A meta-environment for generating programming environments. *ACM Transactions on Software Engineering and Methodology*, 2(2):176–201, 1993.

[KLMM83] G. Kahn, B. Lang, B. Mélèse, and E. Morcos. Metal: a formalism to specify formalisms. *Science of Computer Programming*, 3:151–188, 1983.

[Klo92] J.W. Klop. Term rewriting systems. In S. Abramsky, D. Gabbay, and T. Maibaum, editors, *Handbook of Logic in Computer Science, Volume 2.*, pages 1–116. Oxford University Press, 1992.

[Knu92] D.E. Knuth. *Literate Programming.* Number 27 in CSLI Lecture Notes. Center for the Study of Language and Information, 1992.

[Koo94] J.W.C. Koorn. *Generating Uniform User-Interfaces for Interactive Programming Environments*. PhD thesis, University of Amsterdam, 1994.

[KW93] J. F. Th. Kamperman and H.R. Walters. ARM, abstract rewriting machine. Technical Report CS-9330, Centrum voor Wiskunde en Informatica, 1993. Available by *ftp* from ftp.cwi.nl:/pub/gipe as KW93.ps.Z.

[Lév75] J.-J. Lévy. An algebraic interpretation of the $\lambda\beta$K-calculus and a labelled λ-calculus. In C. Böhm, editor, *λ-Calculus and Computer Science Theory*, number 37 in LNCS. Springer-Verlag, 1975.

[LP87] P. Lee and U. Pleban. A realistic compiler generator based on high-level semantics. In *Conference Record of the Fourteenth Annual ACM Symposium on Principles of Programming Languages*, pages 284–295. ACM, 1987.

[Mar91] L. Maranget. Optimal derivations in weak lambda-calculi and in orthogonal term rewriting systems. In *Proceedings of the Eighteenth conference on Principles of Programming Languages POPL '91*, pages 225–269, 1991.

[Mar93] A. Martin. Encoding W: A logic for Z in 2OBJ. In J.C.P. Woodcock and P.G. Larsen, editors, *FME'93: Industrial Strength Formal Methods*, volume 670 of *LNCS*, pages 462–481, 1993.

[Meu88] E.A. van der Meulen. Algebraic specification of a compiler for a language with pointers. Report CS-R8848, Centrum voor Wiskunde en Informatica (CWI), Amsterdam, 1988.

[Meu92] E.A. van der Meulen. Deriving incremental implementations from algebraic specifications. In *Proceedings of the Second International Conference on Algebraic Methodology and Software Technology*, Workshops in Computing, pages 277–286. Springer-Verlag, 1992. Full version available by *ftp* from ftp.cwi.nl:/pub/gipe as Meu90.ps.Z.

[Meu94] E. A. van der Meulen. *Incremental Rewriting*. PhD thesis, University of Amsterdam, 1994. Available by *ftp* from ftp.cwi.nl:/pub/gipe as Meu94.ps.Z.

[Mey88] B. Meyer. *Object-oriented Software Construction*. Prentice Hall, 1988.

[MG85] J. Meseguer and J.A. Goguen. Initiality, induction, and computability. In M. Nivat and J.C. Reynolds, editors, *Algebraic Methods in Semantics*, pages 459–541. Cambridge University Press, 1985.

[Mic93] G.J. Michaelson. *Interpreter Prototypes from Formal Language Definitions*. PhD thesis, Heriot-Watt University, 1993. Available by *ftp* from brolga.cc.uq.oz.au, as /theses/greg-michaelson.ps.Z.

[Moh89] C.K. Mohan. Priority rewriting: Semantics, confluence, and conditionals. In N. Dershowitz, editor, *Rewriting Techniques and Applications*, volume 355 of *LNCS*, pages 278–292, 1989.

[Mos79] P.D. Mosses. SIS — semantics implementation system: Reference manual and user guide. Technical Report DAIMI MD-30, Computer Science Department, Aarhus University, 1979.

[Mos89] P.D. Mosses. Unified algebras and institutions. In *LICS'89, Proceedings 4th Annual Symposium on Logic in Computer Science*, pages 304–312. IEEE, 1989.

[Mos92] P.D. Mosses. *Action Semantics*, volume 26 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1992.

[MS88] C.K. Mohan and M.K. Srivas. Conditional specifications with inequational assumptions. In S. Kaplan and J.-P. Jouannaud, editors, *Conditional Term Rewriting Systems*, volume 308 of *LNCS*, pages 161–178. Springer-Verlag, 1988.

[MS89] C.K. Mohan and M.K. Srivas. Negation with logical variables in conditional rewriting. In N. Dershowitz, editor, *Rewriting Techniques and Applications*, volume 355 of *LNCS*, pages 292–310, 1989.

[MT92] K. Meinke and J.V. Tucker. Universal algebra. In S. Abramsky, D. Gabbay, and T.S.E. Maibaum, editors, *Handbook of Logic in Computer Science. Volume 1.*, pages 189–411. Oxford University Press, 1992.

[MTH90] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML*. MIT Press, 1990.

[MV90] S. Mauw and G.J. Veltink. A process specification formalism. *Fundamenta Informaticae*, 12:85–139, 1990.

[MW92] U. Martin and J. M. Wing, editors. *First International Workshop on Larch*. Workshops in Computing. Springer-Verlag, 1992.

[MW93] P.D. Mosses and D.A. Watt. Pascal action semantics. Technical report, Aarhus University, 1993. Draft, version 0.6. Available by *ftp* from ftp.daimi.aau.dk: pub/action/pascal.

[Nip91] T. Nipkow. Higher-order critical pairs. In *Proceedings of the Sixth Annual IEEE Symposium on Logic in Computer Science*, pages 342–349. IEEE Computer Society Press, 1991.

[NN92] H.R. Nielson and F. Nielson. *Semantics with Applications; A Formal Introduction*. John Wiley & Sons, 1992.

[O'D85] M.J. O'Donnell. *Equational Logic as a Programming Language*. MIT Press, 1985.

[Oos94] V. van Oostrom. *Confluence for Abstract and Higher-Order Rewriting*. PhD thesis, Vrije Universiteit, Amsterdam, March 1994.

[OR94] V. van Oostrom and F. van Raamsdonk. Comparing combinatory reduction systems and higher-order rewrite systems. In J. Heering, K. Meinke, B. Möller, and T. Nipkow, editors, *Higher-Order Algebra, Logic and Term Rewriting (HOA '93)*, volume 816 of *LNCS*. Springer-Verlag, 1994.

[Ørb94] P. Ørbaek. Oasis: An optimizing action-based compiler generator. In P.A. Fritszon, editor, *CC'94; Compiler Construction*, volume 786 of *LNCS*, pages 1–15. Springer-Verlag, 1994.

[Pal92a] J. Palsberg. An automatically generated and provably correct compiler for a subset of Ada. In *Proceedings of the Fourth IEEE Interncational Conference on Computer Languages, ICCL92*, pages 117–126. IEEE, 1992.

[Pal92b] J. Palsberg. A provably correct compiler generator. In *Proceedings European Symposium on Programming, ESOP92*, volume 592 of *LNCS*, pages 418–434. Springer-Verlag, 1992.

[Pau82] L. Paulson. A semantics-directed compiler generator. In *Ninth Annaul ACM Symposium on the Principles of Programmgin Languages*, pages 224–233, 1982.

[Plo81] G.D. Plotkin. A structural approach to operational semantics. Report DAIMI FN-19, Aarhus University, 1981.

[RC86] J. Rees and W. Clinger (editors). Revised^3 report on the algorithmic language Scheme. *ACM SIGPLAN Notices*, 21(12):37–97, 1986.

[Rek92] J. Rekers. *Parser Generation for Interactive Environments*. PhD thesis, University of Amsterdam, 1992. Available by *ftp* from ftp.cwi.nl:/pub/gipe as Rek92.ps.Z.

[Res94] M. Res. A generated programming environment for RISLA, a specification language defining financial products. Master's thesis, Programming Research Group, University of Amsterdam, 1994.

[RS93] R. Ruei and K. Slonneger. Semantic prototyping: Implementing action semantics in Standard ML. Technical report, The University of Iowa, Department of Computer Science, Iowa City, 1993.

[RT89a] T. Reps and T. Teitelbaum. *The Synthesizer Generator: a System for Constructing Language-Based Editors*. Springer-Verlag, 1989.

[RT89b] T. Reps and T. Teitelbaum. *The Synthesizer Generator Reference Manual - Third edition*. Springer-Verlag, 1989.

[RT92] E.P.B.M. Rutten and S. Thiébaux. Semantics of Manifold: specification in ASF+SDF and extension. Technical Report CS-R9269, Centrum voor Wiskunde en Informatica (CWI), Amsterdam, 1992. Available by *ftp* from ftp.cwi.nl:/pub/CWIreports/IS.

[Sch86] D.A. Schmidt. *Denotational Semantics, A Methodology for Language Development*. Allyn and Bacon, Inc., 1986.

[Sto77] J.E. Stoy. *The Scott-Strachey Approach to Programming Language Theory*. MIT Press, 1977.

[Ten77] R. D. Tennent. A denotational definition of the programming language PASCAL. Technical Report 77-47, Queen's University, Kingston, Ontario, 1977.

[Thi84] J.J. Thiel. Stop losing sleep over incomplete data type specifications. In *Conference Record of the Eleventh ACM Symposium on Principles of Programming Languages*, pages 76–82, 1984.

[Tip93] F. Tip. Animators for generated programming environments. In P. Fritzson, editor, *AADEBUG'93; Automated and Algorithmic Debugging*, volume 749 of *LNCS*, pages 241–254. Springer-Verlag, 1993.

[Tof90] M. Tofte. *Compiler Generators: What They Can Do, What They Might Do, and What They Will Probably Never Do*, volume 19 of *EATCS Monographs on Theoretical Computer Science*. Springer-Verlag, 1990.

[Tom85] M. Tomita. *Efficient Parsing for Natural Languages*. Kluwer Academic Publishers, 1985.

[Vis92] E. Visser. Syntax and static semantics of Eiffel. A case study in algebraic specification techniques. Available by *ftp* from ftp.cwi.nl:/pub/gipe as Vis92.ps.Z, 1992.

[Vis93] E. Visser. Combinatory logic & compilation of list matching. Master's thesis, University of Amsterdam, Programming Research Group, 1993.

[Voi86] F. Voisin. CIGALE: a tool for interactive grammar construction and expression parsing. *Science of Computer Programming*, 7:61–86, 1986.

[Wad80] C.P. Wadsworth. Some unusual λ-calculus numeral systems. In J.P. Seldin and J.R. Hindly, editors, *To H.B. Curry: Essays on Combinatory Logic, λ-calculus, and Formalism*, pages 215–229. Academic Press, 1980.

[Wal91] H.R. Walters. *On Equal Terms, Implementing Algebraic Specifications*. PhD thesis, University of Amsterdam, 1991. Available by *ftp* from ftp.cwi.nl:/pub/gipe as Wal91.ps.Z.

[Wan84] M. Wand. A semantic prototyping system. In *Proceedings of the ACM SIGPLAN Symposium on Compiler Construction*, pages 213–221, 1984. Appeared as SIGPLAN Notices 19(6).

[Wat79] D.A. Watt. An extended attribute grammar for Pascal. *SIGPLAN Notices*, 14(2):60–74, 1979.

[Wat87] D.A. Watt. An action semantics of Standard ML. In M. Main, A. Melton, M. Mislove, and D.A. Schmidt, editors, *Mathematical Foundations of Programming Language Semantics*, volume 298 of *LNCS*, page 5720589. Springer-Verlag, 1987.

[Wat91] D.A. Watt. *Programming Language Syntax and Semantics*. Prentice Hall, 1991.

[Wat94] D.A. Watt. Using ASF+SDF to prototype an action notation transformer and interpreter. In P.D. Mosses, editor, *Proceedings of the First International Workshop on Action Semantics*. Aarhus University, 1994. Published as Technical Report.

[Wie91] F. Wiedijk. *Persistence in Algebraic Specifications*. PhD thesis, University of Amsterdam, 1991.

[Wir90] M. Wirsing. Algebraic specification. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, Volume B*, pages 675–789. Elsevier Science Publishers, 1990.

[Wol93] D.A. Wolfram. *The Clausal Theory of Types*, volume 21 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1993.

[Zha92] H. Zhang. Herky: High performance rewriting in RRL. In D. Kapur, editor, *Automated Deduction CADE-11*, volume 607 of *LNAI*, pages 698–700. Springer-Verlag, 1992.

Index

Samenvatting

Dit proefschrift gaat over wiskundige beschrijvingen van computertalen. Dit roept de vraag op wat computertalen zijn, en waarom deze wiskundig beschreven zouden moeten worden.

Een computertaal geeft mensen de mogelijkheid om met een geautomatiseerd systeem te communiceren. Om een drietal voorbeelden te noemen: De *programmeertaal* Pascal kan gebruikt worden om volledig nieuwe systemen te bouwen; De *query-taal* SQL dient om vragen te stellen aan een gegevensbank; De *commandotaal* van MS-DOS, ten slotte, stelt een PC-gebruiker in staat bestanden te kopiëren of te verwijderen. Voorbeeldzinnen van deze talen zijn te vinden in Figuur S.1.

Het spreekt vanzelf dat de systemen waaraan in zo'n computertaal opdrachten gegeven worden iets van de betreffende taal moeten weten. Dit wordt mogelijk gemaakt door *taal-specifieke hulpmiddelen*, die zinnen in een bepaalde taal kunnen ontleden, intepreteren, analyseren, corrigeren, optimaliseren, vertalen, enz. Een collectie van dergelijke hulpmiddelen wordt een "omgeving" voor die taal genoemd.

Om het inzicht in computertalen te vergroten, is een groot deel van het informatica-onderzoek gewijd aan het beschrijven en analyseren van allerlei soorten talen. Centraal hulpmiddel daarbij is een wiskundige beschrijving van een taal. Zo'n beschrijving karakteriseert de belangrijkste eigenschappen van de taal, waaronder de syntax en semantiek (structuur en betekenis) van de taal.

Het maken van een beschrijving is veel werk, maar heeft ten minste twee belangrijke

```
if munt = kwartje
 then
 som := som + 0.25
```

(A) Pascal

```
SELECT Titel
FROM   Boekenbestand
WHERE  Auteur = 'G. Reve'
AND    Jaar >= 1963
```

(B) De querytaal SQL

```
XCOPY A:\ B:\ /S /E
```

(C) Een regel MS-DOS

Figuur S.1. Fragmenten van enkele computertalen praktische voordelen:

- Allereerst kan zij dienen als *definitie* van een taal.

  Dit is met name van belang wanneer een *nieuwe* taal wordt ontworpen.[1] De wiskundige definitie vormt dan een precieze en ondubbelzinnige beschrijving van de taal.

- Ten tweede, en dit is verrassender, kan zo'n taalbeschrijving onder bepaalde omstandigheden gebruikt worden om een omgeving voor een taal automatisch te *genereren* (zie ook Figuur S.2). Dit is mogelijk als de definitie voldoende gedetailleerd is, en wanneer zij zich laat lezen als een soort "recept" dat op de een of andere manier "uitgevoerd" kan worden.

  Ook dit is van groot belang tijdens het ontwerp van een nieuwe taal. Het eenvoudig beschikbaar zijn van een taal-omgeving maakt experimenten met het gebruik van de taal in een vroeg stadium mogelijk, waarmee de bruikbaarheid van de taal onderzocht kan worden.

Dit soort uitvoerbare taaldefinities staan centraal in dit proefschrift, zoals ook tot uitdrukking komt in de titel: *Executable Language Definitions*.

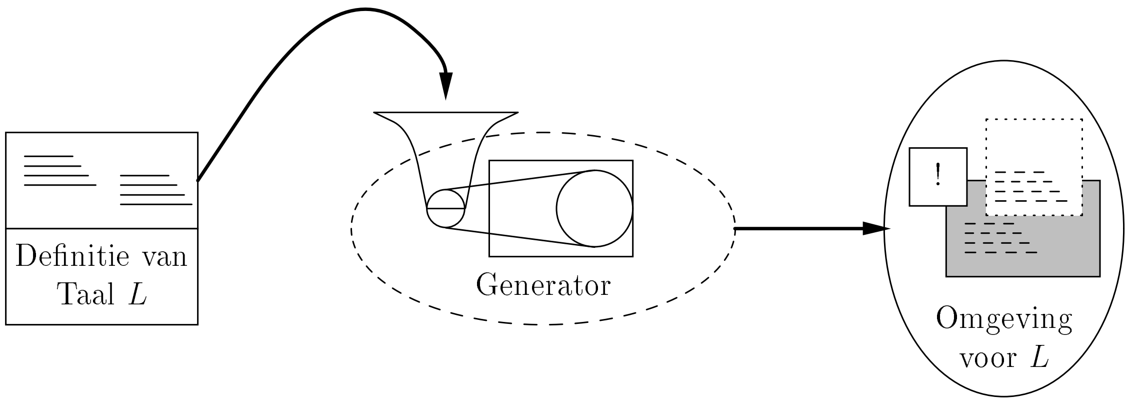
De afgelopen jaren zijn diverse methoden voorgesteld om dergelijke uitvoerbare taaldefinities te schrijven. Dit proefschrift speelt zich af in de context van de *algebraische specificaties*, die eind jaren zeventig werden geïntroduceerd. Vaak kunnen dergelijke specificaties door middel van *termherschrijven* uitgevoerd worden. Eind jaren tachtig is aan het Centrum voor Wiskunde en Informatica en de Universiteit van Amsterdam een algebraisch specificatie formalisme met de naam ASF+SDF[2] ontwikkeld. Dit formalisme is speciaal geschikt voor het beschrijven van talen. Samen met een systeem dat hierbij hoort, de zogenaamde ASF+SDF Meta-environment, kunnen de taaldefinities uitgevoerd worden, wat weer leidt tot automatisch gegenereerde, taalspecifieke omgevingen.

Deel I van dit proefschrift bespreekt ervaringen met het gebruik van ASF+SDF. De meest in het oogspringende voordelen van ASF+SDF zijn de eenvoud van het formalisme (dat daarom makkelijk te leren en te begrijpen is), en de mogelijkheid om heel leesbare specificaties te schrijven. Minder duidelijk is echter wat nu de belangrijkste beperkingen van ASF+SDF zijn. Daarom worden in Deel I drie *case studies* besproken, die gebruikt worden om de problemen met ASF+SDF boven tafel te krijgen.

Hoofdstuk 3 bespreekt het ontwerp van een taal die ontwikkeld is door Cap Volmac en Bank MeesPierson. MeesPierson biedt zijn klanten een groot aantal "renteprodukten" aan. Om te overleven op de financiële markten, is het belangrijk om snel in staat te zijn nieuwe rente-produkten te lanceren, die voor de klant net iets aantrekkelijker zijn dan de produkten van de concurrent. Het introduceren van een nieuw produkt heeft echter belangrijke gevolgen voor de geautomatiseerde systemen van de bank: de financiële administratie moet er van op de hoogte zijn, en de management informatiesystemen moeten de extra rente-risico's die de bank door het nieuwe produkt loopt

[1] Elk jaar worden honderden nieuwe talen geintroduceerd.

[2] ASF+SDF = Algebraic Specification Formalism + Syntax Definition Formalism



Figuur S.2. Omgevingen genereren uit taal definities.

goed kunnen inschatten. Het wijzigen van deze software is een tijdrovende bezigheid. Daarom hebben CAP Volmac en MeesPierson besloten om een *taal*, genaamd RISLA, te ontwerpen waarin de karakteristieke eigenschappen van de diverse produkten kunnen worden vastgelegd. Gegeven een RISLA-beschrijving van een bepaald produkt, kan vervolgens de software nodig voor het verwerken van dit produkt automatisch *gegenereerd* worden. ASF+SDF is gebruikt tijdens het ontwerp van RISLA.

Hoofdstuk 4 laat zien hoe ASF+SDF gebruikt kan worden om de programmeertaal Pascal te definieren. De nadruk wordt gelegd op de *statische semantiek*, d.w.z. dat beschreven wordt welke fouten in een Pascal programma ontdekt kunnen worden zonder dat dat programma wordt uitgevoerd.

Hoofdtuk 5, ten slotte, illustreert het gebruik van ASF+SDF voor het verkrijgen van taalspecifieke hulpmiddelen voor een *specificatie-taal*, zoals gebruikt in het kader van *action semantics*. Deze laatste case study is veruit de meest gecompliceerde van de drie.

De vraag die overblijft is in hoeverre deze case studies problemen aan het licht hebben gebracht. Allereerst zij opgemerkt dat in alle gevallen ASF+SDF een zeer geschikt formalisme bleek om de betreffende taal te beschrijven. Toch liet het ontwerp van de taal voor Bank MeesPierson zien dat het voor commerciële toepassingen belangrijk is dat de link met de in het bedrijfsleven gangbare taal COBOL eenvoudiger te leggen is. De specificatie van de statische semantiek van Pascal toonde aan dat het gebruik van alleen *termherschrijven* om taaldefinities uit te voeren niet voldoende informatieve hulpmiddelen opleverde. Het gebruik van ASF+SDF voor het construeren van een omgeving voor action semantics, ten slotte, leidde tot een uitgebreide lijst van opmerkingen en aanbevelingen om ASF+SDF te verbeteren.

Voor het tweede probleem, dat zich voordeed in de Pascal studie, wordt een oplossing besproken in Deel II van dit proefschrift. In de Pascal case studie wordt een definitie gegeven van de fouten in een Pascal programma die gevonden kunnen worden door het programma te analyseren. Als we deze definitie uitvoeren krijgen we een (type) *checker* van Pascal programma's. Laten we nu aannemen dat we zelf een klein Pascal programma geschreven hebben en onze checker daarop loslaten. Waarschijnlijk hebben we wel een paar fouten gemaakt, en de checker zal die opsporen, en een lijstje van fouten opleveren. Dit vertelt ons *wat* er fout is.

Wat we ook willen weten is, met name wanneer het programma dat we aan het schrijven zijn groot is, *waar* in ons programma we de fout hebben gemaakt. De constatering die in hoofdstuk 4 is gedaan is dat de gegenereerde checker deze informatie niet kan opleveren wanneer we de checker verkregen hebben door de definitie simpelweg d.m.v. termherschrijven uit te voeren. In Deel II wordt *origin tracking* besproken, een uitbreiding van termherschrijven die automatisch de *origins* van bepaalde berekeningen bijhoudt. Voor elke tussenwaarde die het resultaat is van een enkele herschrijfstap (d.w.z. voor elke stap in de grote berekening), wordt een verzameling "wijzers" (de origins) naar de relevante stukken beginwaarde bijgehouden. Deze verzamelingen worden bijgehouden tot en met het eindresultaat, waar ze dan de gewenste origin informatie opleveren. Een moeilijkheid is onder meer de keuze welke informatie bijgehouden moet worden (niet teveel, maar ook niet te weinig!) Bovendien is het wenselijk dat deze informatie volledig *automatisch* wordt bijgehouden; de oorspronkelijke taaldefinitie mag geen wijzigingen ondergaan.

In Hoofdstuk 6 worden de algemene principes van origin tracking besproken. In Hoofdstuk 7 laten we zien hoe we, als we ons concentreren op een veelvoorkomende vorm van algebraische taaldefinities, *primitief-recursieve schema's*, de kwaliteit van de opgeleverde origins kunnen verbeteren. In Hoofdstuk 8 leggen we uit hoe we, als we de manier om taaldefinities op te schrijven uitbreiden met krachtige *hogere-orde* constructies, toch nog steeds op een goede manier origin tracking kunnen realiseren.

Titles in the ILLC Dissertation Series:

Transsentential Meditations; Ups and downs in dynamic semantics
Paul Dekker
ILLC Dissertation series 1993-1

Resource Bounded Reductions
Harry Buhrman
ILLC Dissertation series 1993-2

Efficient Metamathematics
Rineke Verbrugge
ILLC Dissertation series 1993-3

Extending Modal Logic
Maarten de Rijke
ILLC Dissertation series 1993-4

Studied Flexibility
Herman Hendriks
ILLC Dissertation series 1993-5

Aspects of Algorithms and Complexity
John Tromp
ILLC Dissertation series 1993-6

The Noble Art of Linear Decorating
Harold Schellinx
ILLC Dissertation series 1994-1

Generating Uniform User-Interfaces for Interactive Programming Environments
Jan Willem Cornelis Koorn
ILLC Dissertation series 1994-2

Process Theory and Equation Solving
Nicoline Johanna Drost
ILLC Dissertation series 1994-3

Calculi for Constructive Communication
Jan Jaspars
ILLC Dissertation series 1994-4

Executable Language Definitions
Arie van Deursen
ILLC Dissertation series 1994-5