INSTITUT FÜR INFORMATIK
UND PRAKTISCHE MATHEMATIK

Techniques for Modelling Structured Operational and Denotational Semantics Definitions with Term Rewriting Systems

Karl-Heinz Buth

Bericht Nr. 9414
Oktober 1994

CHRISTIAN-ALBRECHTS-UNIVERSITÄT
KIEL

Institut für Informatik und Praktische Mathematik der
Christian-Albrechts-Universität zu Kiel
Olshausenstr. 40
D – 24098 Kiel

Techniques for Modelling Structured Operational and Denotational Semantics Definitions with Term Rewriting Systems

Karl-Heinz Buth

Bericht Nr. 9414
Oktober 1994

e-mail: khb@informatik.uni-kiel.d400.de

Dieser Bericht enthält die Dissertation des Verfassers.

Referent: Prof. Dr. Hans Langmaack
Koreferent: Prof. Dr. Rudolf Berghammer

Abstract

A fundamental requirement for the application of automatic proof support for program verification is that the semantics of programs be appropriately formalized using the object language underlying the proof tool. This means that the semantics definition must not only be stated as syntactically correct input for the proof tool to be used, but also in such a way that the desired proofs can be performed without too many artificial complications. And it must be clear, of course, that the translation from mathematical metalanguage into the object language is correct.

The objective of this work is to present methods for the formalization of structured operational and denotational semantics definitions that meet these requirements. It combines techniques known from implementation of the λ-calculus with a new way to control term rewriting on object level, thus reaching a conceptually simple representation based on unconditional rewriting. This deduction formalism is available within many of the existent proof tools, and therefore application of the representation methods is not restricted to a particular tool.

Correctness of the representations is achieved by proving that the non-trivial formalizations yield results that are equivalent to the meta-level definitions in a strong sense. Since the representation algorithms have been implemented in form of executable programs, there is no need to carry out tedious coding schemes by hand. Semantics definitions can be stated in a format very close to the usual meta language format, and they can be transformed automatically into an object-level representation that is accessible to proof tools.

The formalizations of the two semantics definition styles are designed in a consistent way, both making use of the same modelling of the underlying mathematical basis. Therefore, they can be used simultaneously in proofs. This is demonstrated in a larger example, where an operational and a denotational semantics definition for a programming language are proved to be equivalent using the Larch Prover. This proof has been carried out by hand before, and so the characteristics of the automated proof can be made quite clear.

Keywords

automated program verification, de Bruijn index, denotational semantics, λ-calculus, $\lambda\sigma$-calculus, Larch Prover, programming language semantics, semantics equivalence proofs, structured operational semantics, term rewriting, transition systems

Zusammenfassung

Um bei der Programmverifikation automatische Beweisunterstützungssysteme erfolgreich einsetzen zu können, ist es nötig, die Semantik der betrachteten Programme auf adäquate Weise in der Objektsprache des Beweissystems darzustellen. Das heißt nicht nur, daß die geforderte syntaktische Form einzuhalten ist, sondern auch, daß die gewünschten Beweise ohne allzu viele künstliche Komplikationen durchgeführt werden können. Daneben muß natürlich sichergestellt werden, daß die Übersetzung von der mathematischen Metasprache in die Objektsprache korrekt durchgeführt wird.

Ziel dieser Arbeit ist es, für zwei Semantikdefinitionsstile solche Formalisierungsmethoden vorzustellen, nämlich für den strukturiert-operationellen und für den denotationellen. Durch Kombination von Techniken, die aus der Implementierung des λ-Kalküls bekannt sind, mit einer neuartigen Methode zur Kontrolle von Termersetzung auf Objektniveau wird dabei eine konzeptuell einfache Darstellung erreicht, die sich nur auf einfaches, bedingungsfreies Termersetzen stützt. Da dieser Deduktionsmechanismus in sehr vielen existierenden Beweissystemen implementiert ist, ist die Anwendung der Formalisierungsmethoden nicht auf eine bestimmtes Beweissystem beschränkt.

Für die nichttrivialen Formalisierungsschritte wird bewiesen, daß die ursprünglichen mathematischen Definitionen und ihre Repräsentationen in einem starken Sinne äquivalent sind, womit die Korrektheit der Methode sichergestellt wird. Die Formalisierungsalgorithmen sind implementiert in Form von lauffähigen Programmen, so daß die Methode keine aufwendigen Handcodierungen erfordert. Semantikdefinitionen können in einem Format notiert werden, das der üblichen mathematischen Notation sehr ähnlich ist, und die Transformation auf das Objektniveau des gewählten Beweissystems kann automatisch erfolgen.

Die Formalisierungen der beiden betrachteten Definitionsstile stützen sich auf dieselbe Modellierung der zugrundeliegenden mathematischen Basis. Daher ist es problemlos möglich, sie gleichzeitig in Beweisen zu verwenden. Dies wird anhand eines größeren Beispiels demonstriert, in dem für eine gegebene feste Sprache die Äquivalenz einer strukturiert-operationellen und einer denotationellen Semantikdefinition nachgewiesen wird. Das benutzte Beweissystem ist dabei der Larch Prover. Für dieses Problem liegt bereits ein konventioneller Handbeweis vor, so daß die Besonderheiten des automatisierten Beweises gut deutlich gemacht werden können.

Schlüsselworte

automatisierte Programmverifikation, de Bruijn-Index, denotationelle Semantik, λ-Kalkül, $\lambda\sigma$-Kalkül, Larch Prover, Semantik für Programmiersprachen, Semantikäquivalenzbeweis, strukturiert-operationelle Semantik, Termersetzung, Transitionssysteme

Contents

Chapter 1

Introduction

Due to their ever expanding computing power and miniaturization, modern computers are being increasingly used also in safety critical applications where a malfunction could entail severe damage to the property, to the health, or even to the life of persons.[1] Therefore, the concern about the absence of such malfunctions is increasing as well (if probably not at the same rate).[2] A malfunction is a behaviour that is incorrect with respect to some requirement, and hence the basic concern is about correctness of systems.

In the field of software, the only method that at least has the chance to guarantee correctness is the formal verification of programs against formal specifications. Of course, this method also has its problems, as correctly remarked by De Millo, Lipton and Perlis [30]; see also the discussion following Fetzer's article in the *Communications of the ACM* [42]. Like any human undertaking, verification is subject to errors and omissions, and the problem of capturing the requirements in a formal specification is far from being solved in a generally accepted way (cf. e. g. [41]). But under the assumption that the formal specification suitably reflects the requirements and that no errors occur during the process itself, verification is capable of mathematically proving the absence of errors in a program. This is a quality that other methods such as testing fail to achieve.[3]

The main practical obstacle for implementing verification as a standard part of software development processes is the size of the problem. Even for relatively small programs, the sheer length of the proofs to perform normally prevents all attempts to verify them in the usual mathematical proof style. But, luckily, there is an important aspect of these proofs that shows a way to apply verification after all. Most programs contain large parts that are "obviously" correct, i. e. do not require complicated lines of reasoning for their verification, and only small parts that include intricate algorithms. The verification of these latter parts can be quite as complicated as proving general mathematical theorems. For the former parts, however, often simple standard techniques such as case distinctions, inductive arguments, and calculations suffice.

This is the starting point for automatic support of program verification. Tools offering such support do not provide very much built-in ingenuity, despite all attempts to include a variety of heuristics. Therefore proving general mathematical theorems with an automated prover usually is a complicated task (see e. g. Shankar's proof [109] of Gödel's incompleteness theorem [55] with

[1]For overviews, see the proceedings of the SAFECOMP conference series [38, 53, 26, 50].
[2]See e. g. the regular RISKS forum in the *ACM SIGSoft Software Engineering Notes.*
[3]Remarked e. g. by Dijkstra ([34], p. 20).

the Boyer-Moore prover [16]). In such proofs, the effort of adequately formalizing proof structures can be considerable.

But the great strength of computers is the ability to apply simple techniques very often, and therefore proof tools can be adequately used at least for the simpler parts of verification problems. This may be even more true if the complicated parts of the proofs are completely performed by hand. In this case, it is often sufficient to work with a less complex formalization of the problem when dealing with the simpler parts, and thus the proof tool can work more efficiently.

A fundamental requirement for the application of automatic proof support for program verification is that the semantics of programs be appropriately formalized using the formal language underlying the proof tool (the so-called *object language*). This means that it must not only be stated in the syntactic form of correct input for the proof tool to be used, but also in such a way that the desired proofs can be performed without too many artificial complications, and, of course, it must be clear that the translation from mathematical metalanguage into the object language is correct.

The objective of this work is to present methods for the *formalization of structured operational and denotational semantics definitions* that meet these requirements. It combines techniques known from implementation of the λ-calculus with a new way to control term rewriting on object level, thus reaching a conceptually simple representation based on unconditional rewriting. This deduction formalism is available within many of the existent proof tools, and therefore application of the representation methods is not restricted to a particular tool.

Correctness of the representations is achieved by formally proving that the non-trivial formalizations yield results that are equivalent to the meta-level definitions (in a sense made precise later). Since the representation algorithms have been implemented in form of executable programs, there is no need to carry out complicated coding schemes by hand. Semantics definitions can be stated in a format very close to the usual meta language format, and they can be transformed automatically into an object-level representation that is accessible to proof tools.

The formalizations of the two semantics definition styles are designed in a consistent way, both making use of the same modelling of the underlying mathematical basis. Therefore, they can be used simultaneously in proofs. This is demonstrated in a larger example, where an operational and a denotational semantics definition for a programming language are proved to be equivalent. This proof has been carried out by hand before [82], and so the characteristics of the automated proof can be made quite clear.

Term Rewriting Systems

Replacement of equals by equals is one of the most basic steps in mathematical proofs. The idea of term rewriting is to formalize such replacements in the form of "directed equations" $l \longrightarrow r$ meaning that any instance of l may be substituted by a corresponding instance of r. The properties of "term rewriting systems" consisting of such rules have been examined since the beginning of the 1970s. By now, rewriting forms a theory of its own right, but it has also close links to algebraic specification where data types are defined by equations; for purposes of computation, as in rapid prototyping (cf. e. g. [73, 74]), the equations must become "directed", resulting in rewrite rules.

There exist many extensions of the simple replacement scheme $l \longrightarrow r$. The most prominent is *conditional rewriting*: here the rules have the form $c \Rightarrow l \longrightarrow r$, where c is a condition that has to be (rewritten to) true before the rule can be applied. A number of proposals have also been presented on how to extend rewriting with parts of the λ-calculus (e. g. by Dougherty [36, 37] or Loria-Saenz and Steinbach [87]); Klop [79] and Kahrs [77] have defined replacement systems that go beyond term rewriting and λ-calculus by generalizing both formalisms.

In this work, a method will be used to include a subset of λ-calculus into term rewriting without having to extend the rewriting formalism (the $\lambda\sigma$-calculus, presented by Abadi et al. [1]). Moreover, a new technique will be introduced that allows to control the rewriting of a certain class of terms without having to appeal to meta-level control techniques. In particular, this method will allow to work with a rule of the form

$$f(x) \longrightarrow \mathsf{if}\ b\ \mathsf{then}\ f(y)\ \mathsf{else}\ f(x) \quad .$$

The intended semantics of this rule is "*Only rewrite $f(x)$ to $f(y)$ if b holds; otherwise do not change $f(x)$.*" But of course, this rule may be applied infinitely often if b does not hold. The control mechanism to be introduced will be defined in such a way that the intended semantics is retained, but the possibility of constructing non-terminating rewriting sequences with this rule is eliminated.

Due to their conceptual simplicity, rewriting mechanisms are built into many tools for automated proof support: tools as different as the RAP system [73, 74], the Boyer-Moore Prover [16], the Larch Prover [51], the KIV system [65] or PAMELA [22] all include ways to define rewrite rules and to apply them in proofs.

Semantics Definition Styles

There are three main styles for defining the formal semantics of programming languages (cf. Stoy [113] or Nielson and Nielson [99]), viz. the operational, the denotational and the axiomatic approach. The borderlines between the styles, however, sometimes are fluid.

Operational semantics

The meaning of a program is defined by setting up an abstract machine M and interpreting the program in terms of machine instructions of M.

Historically, the operational style was the first one to develop. If M is defined in a "concrete" way, including many details of real hardware, the calculation of the operational semantics of a program comes quite close to the actual execution of the program on the computer. Therefore, operational semantics definitions can provide ideas on how to implement the programming language.

A method that has proved very convenient for the definition of the operational semantics of (concurrent) programming languages is the approach introduced by Plotkin [101] usually referred to as *Structured Operational Semantics* (SOS for short). The method consists of setting up a transition system that contains deduction rules defining the transition relation $\longrightarrow_{\mathcal{S}}$ between state configurations of programs. Examples for languages defined in this way include parts of

CSP [101], Ada [3] and Esterel [10]; process algebra [6] also uses SOS definitions. SOS rules define the semantics of constructs in a compositional way from the semantics of their subcomponents.

Since SOS definitions are sometimes too concrete for the purpose of reasoning about programs, it is useful to develop more abstract representations that make the definitions more accessible. Moller [94] and Aceto, Bloom and Vaandrager [2] have worked on using sets of equations for this purpose. Bosscher [13] describes aspects of implementing the algorithms from [2] whose aim it is to generate a set of equations that can be used to prove bisimulation properties between processes in a certain type of languages.

In Chapter 7 of the work on hand, an algorithm will be presented that also transforms SOS definitions, but with a different aim. Here, the result is a set of rewrite rules, and it will be proved that the rewriting sequences that can be constructed using these rules correspond to the transitions that are possible in the original SOS system in a very close way. Essentially, any SOS transition sequence has a simulating rewriting sequence, and all rewriting sequences related to the SOS system correspond to actual SOS transition sequences.

The main problem is that those deduction rules defining $\rightharpoonup_S$ put less restrictions on the use of free variables than the simple standard rewriting formalism does. The solution that will be presented uses the *$\lambda\sigma$-calculus* to completely abolish problems with such variables; it relies on the control mechanisms mentioned earlier. Since the form of $\lambda\sigma$-calculus that is used can be stated in form of a simple rewrite system, the whole simulation of SOS systems can be accomplished without extending the rewriting formalism. Hence, the simulation can be implemented in any tool supporting simple term rewriting.[4]

There are several examples for automated verification of compilers (code generators, to be more exact) based on operational semantics definitions (albeit not in SOS form). Among these are Young [116] using the Boyer-Moore Prover [16], Berghammer, Ehler, and Zierer [8] using the RAP system [73, 74], and Schmidt [105] using the TIP system [49, 48].

Denotational semantics

The meaning of (pieces of) programs is defined as an element of some mathematical value domain (hence the name "mathematical semantics" formerly used for this style). Usually, the semantic functions are recursively defined, the meaning of a construct being expressed compositionally in terms of the meaning of its subcomponents.

The main difficulty of denotational semantics is the definition of appropriate value domains for the different syntactical categories. Fundamental work in this area includes that of Milne and Strachey [92] and Scott [107]. Plotkin [100] and Smyth [110] address the special problems concerning nondeterminism where sets of results have to be considered.

Denotational semantics is regarded as most suitable for language designers since it provides the most concise definition of the meaning of language constructs.

Since denotational function definitions are usually given by equations in a format close to rewrite rules that have a "natural direction", there is no problem in setting up a simulating rewrite system. Here the main difficulty lies in finding an appropriate representation of the data types that are much more structured than those needed for SOS systems. The problem of how to model

[4]Only some small additional assumptions will have to be made.

quantifiers by rewrite systems will have to be dealt with; this entails all the problems of variable conflicts and substitutions. The partial solution of this problem that will be described in Chapter 8 again relies on the $\lambda\sigma$-calculus.

Among the published verification efforts in the area of proving compiler correctness using denotational semantics are Buth and Buth [20, 21] using the PAMELA system [22] and Broy [17] using the Larch Prover [51]. The latter work also makes use of operational definitions.

Axiomatic semantics

Proof rules are attached to (pieces of) programs; the meaning of a program is expressed in terms of its effect on predicates. This style originated from the work of Floyd [43] on the verification of flowcharts. Hoare [67] developed the use of correctness formulas of the form $\{p\}\, s\, \{q\}$ (the so-called *Hoare triples*) which are defined to be true if execution of a program s, when started in a state satisfying the predicate p, ends up in a state satisfying the predicate q (provided s terminates at all). Dijkstra [34] introduced the notion of *weakest preconditions*; the weakest precondition of a program s with respect to some postcondition q is the predicate that is true in an initial state iff s will, if started in this state, terminate in a state satisfying q. This form of *predicate transformer semantics* can also be regarded as denotational: the weakest precondition of programs depends on the weakest preconditions of its components in the same compositional way that can be observed in denotational definitions.

Since axiomatic definitions normally stay close to the syntax of programs and do not involve complicated mathematical domains, they are most suitable for programmers who want to verify their programs. Axiomatic semantics does not "directly" define the meaning of a program; therefore such definitions are often verified against a definition stated in another style (cf. de Bakker [29] or Loeckx and Sieber [86]).

An example for automated verification based on axiomatic semantics is given by Scott and Norrie [108]. They take a semantics definition in form of so-called laws (cf. Hoare et al. [69]), describing the relations between different syntactic constructs of a programming language P, and use a proof tool based on term rewriting (the Larch Prover [51]) to verify parts of a compiler for P.

Outline of the Work

After defining some notation, the work starts in Chapter 3 with a full account of term rewriting which is the basic formalism needed in the rest of the work. Important aspects introduced in this chapter are the modelling of substitutions, and a new mechanism that extends certain terms with information about how to rewrite these terms.

Both the simulation of SOS systems and of denotational definitions rely on a sufficiently strong basic system $\mathcal{B}$ that describes the underlying mathematical domains. In Chapter 4, the kinds of rules needed for the system $\mathcal{B}$ will be described, and how a large part of them can be very systematically (and even automatically) generated from a simple definition of the domains. Chapters 5 and 6 explain the syntactic properties of those kinds of semantics definitions to be dealt with, and Chapters 7 and 8 describe the modelling of SOS and denotational semantics definitions with rewrite systems.

An example implementation of the simulation is described in Chapters 9 and 10 where the *Larch Prover* [51] is used to replay a semantics equivalence proof originating from the ProCoS project [11, 14]. For a given programming language that is equipped which both an SOS semantics and a denotational semantics, the task is to prove equivalence of these definitions. This problem is particularly well-suited to test the simulation methods since it is concerned almost entirely with pure semantics of two different styles.

The appendices start with a description of two programs that generate input for the Larch Prover from simpler specifications. The first one transforms SOS systems into rewrite systems, and the second one generates that part of the basic rewrite system dealing with the data types.

In Appendix B, the full correctness and completeness proofs for the simulation results of Chapter 7 are provided, and finally, Appendix C compares two different modellings of quantified formulas with the Larch Prover.

Acknowledgements

I would like to thank Prof. Dr. Hans Langmaack for giving me the opportunity to work in an interesting research area and for his advice and support. Prof. Dr. Rudolf Berghammer gave many valuable comments on an earlier version of the thesis.

I had my first contacts with the Larch Prover during a three week's stay at Royal Holloway College in London where I visited Prof. Ursula Martin. Her group, in particular Victoria Stavridou, Mike Lai, and Phil Watson, helped a lot in getting acquainted with the tool. Ursula also hinted me at the $\lambda\sigma$-calculus, which is one of the crucial formalisms of this work.

Frits Vaandrager, Bard Bloom, and anonymous referees for the ESOP '94 conference provided many helpful suggestions for the simulation of SOS definitions, based on earlier and less complete versions of the work reported in Chapter 7 [23, 24].

My colleagues in Kiel, in particular Martin Fränzle, Yassine Lakhneche, Markus Müller-Olm, Ruben-Benjamin Reincke, and Prof. Debora Weber-Wulff, always provided an interested forum for discussion, and carefully read drafts of earlier versions of this work. Martin, Yassine and Markus also helped a lot with LaTeX problems and provided many useful macros.

But most of all I thank my wife and colleague Bettina Buth, who not only was a very careful proofreader and partner for discussions, but also a constant source of moral support and encouragement.

Chapter 2

Notation

Natural numbers

$\mathbb{N}$ denotes the set $\{1, 2, \ldots\}$ of positive natural numbers, $\mathbb{N}_0 =_{df} \mathbb{N} \cup \{0\}$ the set of non-negative natural numbers. For $n \in \mathbb{N}$, define $\boldsymbol{[n]} =_{df} \{1, \ldots, n\}$, and define $\boldsymbol{[0]} =_{df} \emptyset$.

Relations

Let M and N be sets, $n \in \mathbb{N}_0$, and $R, S, \longrightarrow \subseteq M \times M$ relations on M. The following notation will be used:

- $\boldsymbol{M \uplus N}$ denotes the union of disjoint sets M and N.
- $\boldsymbol{id_M} =_{df} \{(m, m) \mid m \in M\}$ denotes the identity relation on M.
- $\boldsymbol{R\,;S} =_{df} \{(m, n) \in M \times M \mid \exists p \in M : (m, p) \in R \wedge (p, n) \in S\}$ denotes the composition of R and S in diagrammatical order.
- Whenever suitable, infix notation for binary relations will be used.
- $\xrightarrow{\boldsymbol{n}} =_{df} \begin{cases} id_M & n = 0 \\ \xrightarrow{n-1}\,;\longrightarrow & n > 1 \end{cases}$

  Note that $\xrightarrow{1} = \longrightarrow$.
- $\xrightarrow{+} =_{df} \bigcup_{n \geq 1} \xrightarrow{n}$ denotes the transitive closure of $\longrightarrow$.
- $\xrightarrow{*} =_{df} \bigcup_{n \geq 0} \xrightarrow{n}$ denotes the reflexive and transitive closure of $\longrightarrow$.
- $\longleftarrow =_{df} \longrightarrow^{-1}$ denotes the inverse of $\longrightarrow$.
- $\longleftrightarrow =_{df} (\longrightarrow \cup \longleftarrow)$ denotes the symmetric closure of $\longrightarrow$.

A relation $\leq \subseteq M \times M$ is a **partial ordering** iff it is reflexive, transitive and antisymmetric. If $\leq$ is a partial ordering on M, $y \in M$, and $X \subseteq M$, let $\boldsymbol{X \leq y}$ stand for $\forall x \in X : x \leq y$, and define $\boldsymbol{<} =_{df} \leq \setminus id_M$.

A relation $\equiv \subseteq M \times M$ is an **equivalence relation** iff it is reflexive, transitive and symmetric.

Functions

Let M, N and N_1 be sets.

If a relation $f \subseteq M \times N$ is a **partial function** this is written as $\boldsymbol{f : M \nrightarrow N}$. The set of partial functions from M to N is denoted by $\boldsymbol{M \nrightarrow N}$. Define

- $\mathbf{dom}\,\boldsymbol{f} =_{df} \{m \in M \mid \exists n \in N : f(m) = n\}$ (the domain of f) and
- $\mathbf{rng}\,\boldsymbol{f} =_{df} \{n \in N \mid \exists m \in M : f(m) = n\}$ (the range of f)

If $f : M \nrightarrow N$ with $\operatorname{dom} f = M$, f is a **total function**, written as $\boldsymbol{f : M \longrightarrow N}$.

$\boldsymbol{M \longrightarrow N}$ denotes the set of total functions from M to N, and $\boldsymbol{M^{\mathbb{N}}} =_{df} \mathbb{N} \rightarrow M$ the set of sequences in M. The operator $\rightarrow$ associates to the right, i. e. $M \rightarrow N \rightarrow N_1 =_{df} M \rightarrow (N \rightarrow N_1)$.

Words

Let M be some set.

$\boldsymbol{M^*} =_{df} \bigcup_{n \geq 0} (\,[n] \rightarrow M\,)$ denotes the set of finite sequences or words over M. The empty word in $[0] \rightarrow M$ is denoted by $\boldsymbol{\varepsilon}$. $\boldsymbol{M^+} =_{df} M^* \setminus \{\varepsilon\}$ denotes the set of non-empty words over M.

Let $w \in M^*$ and $m \in M$. If $n \in \operatorname{dom} w$, $\boldsymbol{w_n}$ is also written instead of $w(n)$. $|\,\boldsymbol{w}\,| =_{df} |\operatorname{dom} w|$ denotes the length of w. **Appending** m onto w results in the function $\boldsymbol{m \cdot w} : [|\,w\,| + 1] \rightarrow M$ in M^* defined for $n \in [|\,w\,| + 1]$ by

$$(m \cdot w)(n) =_{df} \begin{cases} m & \text{, if } n = 1 \\ w_{n-1} & \text{, if } n > 1 \end{cases}$$

If $\leq\; \subseteq M \times M$ is some partial ordering on M, then the **lexicographic extension** of $\leq$ onto M^* is a relation on M^* (also written as $\leq$) that is defined by $(m_1, m_2 \in M^*)$:

$$\begin{aligned} m_1 \leq m_2 =_{df}\; & (\,|\,m_1\,| \leq |\,m_2\,| \;\wedge\; \forall k \in \operatorname{dom} m_1 : m_1(k) = m_2(k)\,) \vee \\ & (\,\exists n \in \operatorname{dom} m_1 : |\,m_2\,| \geq n \;\wedge \\ & \quad (\,\forall k \in [n-1] : m_1(k) = m_2(k) \;\wedge\; m_1(n) < m_2(n)\,)) \end{aligned}$$

Quasi-quotes

Throughout this work, it will be important to be able to separate mathematical objects (*meta-level entities*) from their representations in form of character strings (*object-level entities*). In order to make this separation explicit, so-called "quasi-quotes" $\ulcorner \cdot \urcorner$ (sometimes also $\lceil \cdot \rceil$) will be used. If t is a term of the mathematical meta-language, then $\ulcorner t \urcorner$ denotes the printed representation of this term.

> Example: Let $f : \mathbb{N} \rightarrow \mathbb{N}, n \mapsto n + 4$. Then the mathematical term $f(42)$ stands for ("is") the natural number 46, whereas $\ulcorner f(42) \urcorner$ stands for ("is") is the string consisting of the characters "$f(42)$".

Meta-level variables inside of these quotes lead to representations that can be instantiated (hence the name "quasi-quotes": $\ulcorner \cdot \urcorner$ does not denote full quotation).

> Example: If the function *rep* mapping a natural number to a character string is defined by $\mathit{rep} : n \mapsto \ulcorner n + 8 \urcorner$, then $\mathit{rep}(34) = \ulcorner 34 + 8 \urcorner$.

Sometimes it will be necessary to exclude parts of an otherwise quoted expression from being quoted. For this purpose, "inverse quotes" $\llcorner \cdot \lrcorner$ (sometimes also $\lfloor \cdot \rfloor$) will be used.

> Example: Let again $f : \mathbb{N} \to \mathbb{N}, n \mapsto n + 4$. Then
> $$\ulcorner f(\llcorner f(2) \lrcorner + 4) \urcorner = \ulcorner f(6 + 4) \urcorner = \text{“}f(6+4)\text{”} \quad ,$$
> whereas
> $$\ulcorner f(f(2) + 4) \urcorner = \text{“}f(f(2)+4)\text{”} \quad .$$

These expression may be nested: inside $\llcorner \cdot \lrcorner$, again $\ulcorner \cdot \urcorner$ may be used, inside these quotes again $\llcorner \cdot \lrcorner$ and so on. Nesting quotes of the same kind can also be defined in a way that makes sense, but will not be used in this work.

The quasi-quote notation has been suggested by Quine [102] and is also used in denotational semantics (see Stoy [113], pp. 26 ff., where the concept is explained in detail). The inverse quotes define a concept similar to the "back quotes" used in Common Lisp [111].

Chapter 3

Term Rewriting Systems

This chapter starts with a review of the basic notions and properties of many-sorted term rewriting that will be used. In the next section, it will be described how let expressions as they are known from programming and specification languages such as Lisp [111] or VDM-SL [28] can be embedded into term rewriting. Such expressions introduce concepts of λ-calculus as they can be understood as applications of λ-defined functions to given arguments. Finally, a new concept of "contexts" will be introduced that can be used to control rewriting of a certain class of terms. This control mechanism is also expressible completely within term rewriting. Both these embeddings play the key roles in simulation of structured operational semantics definitions (cf. Chapter 7).

3.1 Concepts of Term Rewriting

In this section, only those aspects of term rewriting will be recapitulated that are needed later. In particular, most proofs will be omitted. More detailed expositions can be found e. g. in Huet and Oppen [72], Bergstra and Klop [9], Hofbauer and Kutsche [70], or Dershowitz and Jouannaud [33]. The approach is typed; this has the same advantages a type discipline has for program languages (cf. Goguen and Meseguer [56], p. 1):

- conceptual clarity is facilitated by making explicit the restrictions on the arguments and results of operations, and
- many errors can be detected by type checking before execution of a program (or, as in our case, rewriting of a term).

3.1.1 Basic Definitions

Definition 3.1 ($\mathcal{S}$-sorted set)

Let $\mathcal{S}$ be an index set. An **$\mathcal{S}$-sorted set** A is a family $\{A_S \mid S \in \mathcal{S}\}$ of sets. A is called **disjoint** iff $A_S \cap A_T = \emptyset$ for all distinct $S, T \in \mathcal{S}$. Given two $\mathcal{S}$-sorted sets A and B, an **$\mathcal{S}$-sorted function $\alpha : A \to B$** is an $\mathcal{S}$-indexed family $\{\alpha_S : A_S \to B_S \mid S \in \mathcal{S}\}$ of functions.

Definition 3.2 (signature)

A **many-sorted signature** is a pair $\langle \mathcal{S}, \Sigma \rangle$ where $\mathcal{S}$ is a set of **sorts** and $\Sigma = \{\Sigma_{w,S} \mid w \in \mathcal{S}^*, S \in \mathcal{S}\}$ is an $(\mathcal{S}^* \times \mathcal{S})$-sorted set. Elements of sets in Σ are called **function symbols** or **operators**.

If $f \in \Sigma_{w,S}$ for some $w = S_1 \cdots S_n \in \mathcal{S}^*, n \geq 0$ and $S \in \mathcal{S}$, this will also be written as $\boldsymbol{f : w \longrightarrow S}$ or $\boldsymbol{f : S_1, \ldots, S_n \longrightarrow S}$ (if $n \geq 1$). w is the **arity** of f and S is the **(result) sort** of f. If $f \in \Sigma_{\varepsilon,S}$ for some $S \in \mathcal{S}$, f is called a **constant** of sort S.

If the sort set $\mathcal{S}$ is clear from the context, Σ will be written instead of $\langle \mathcal{S}, \Sigma \rangle$.

Note that this definition allows *operator overloading*; there may be operators $f \in \Sigma_{w_1,S_1} \cap \Sigma_{w_2,S_2}$ for different $\langle w_1, S_1 \rangle$ and $\langle w_2, S_2 \rangle$.

From now on, assume a fixed set $\mathcal{S}$ of sorts and a fixed signature $\langle \mathcal{S}, \Sigma \rangle$ also denoted by Σ. Furthermore, assume a disjoint $\mathcal{S}$-sorted set V of **variables** such that V_S is enumerable for each $S \in \mathcal{S}$. The following abbreviations will be used:

- If $f \in \Sigma_{w,S}$ for some w and S that are not important in the current context, then $f \in \Sigma$ will be written instead.
- The sets V and $\bigcup_{S \in \mathcal{S}} V_S$ will be identified.
- If f is an $\mathcal{S}$-sorted function from V to some set X, $f(v)$ will also be written instead of $f_S(v)$ for $v \in V_S$ and $\mathrm{dom}\,(f)$ instead of $\bigcup\{\mathrm{dom}\,(f_S) \mid S \in \mathcal{S}\}$.

Since V is disjoint, the last two abbreviations are well-defined.

Definition 3.3 (term, $T\,(\Sigma, V)$)

t is a **term over $\boldsymbol{\Sigma}$ and $\boldsymbol{V}$ with type $\boldsymbol{S \in \mathcal{S}}$** iff one of the following conditions holds:

(1) t is a **variable**: $t = \ulcorner v \urcorner$ for some $v \in V_S$.

(2) t is a **constant**: $t = \ulcorner c \urcorner$ for some $c \in \Sigma_{\varepsilon,S}$.

(3) t is a **complex term**: $t = \ulcorner f\,(t_1, \ldots, t_n) \urcorner$ for some $n \in \mathbb{N}$ and $f \in \Sigma_{w,S}$ with $w = S_1 \ldots S_n$ such that for each $i \in [n]$, t_i is a term over Σ and V with type S_i.

$\boldsymbol{T\,(\Sigma, V)_S}$ denotes the set of all terms over Σ and V with type S, and $\boldsymbol{T\,(\Sigma, V)} =_{df} \bigcup_{S \in \mathcal{S}} T\,(\Sigma, V)_S$ denotes the set of all possible terms over Σ and V, regardless of their type. If $t \in T\,(\Sigma, V)_S$, define $\boldsymbol{type\,(t)} =_{df} S$; in this case, t will sometimes be called an **$\boldsymbol{S}$-typed term**.

For terms t, $\boldsymbol{var\,(t)}$ denotes the set of all variables that occur inside of t. A term t is called a **ground term** iff $var\,(t) = \emptyset$. The set of all ground terms over Σ with type S is denoted by $\boldsymbol{T\,(\Sigma)_S}$, and the set of all ground terms, regardless of their type, by $\boldsymbol{T\,(\Sigma)}$.

Complex terms whose outermost operator is some $f \in \Sigma$ will also be called **$\boldsymbol{f}$-terms**.

Note the use of the quasi-quotes; the separation of object-level from meta-level entities will become particularly important when the semantics of terms is considered (starting in Section 3.1.2). For this, a way is needed to notationally distinguish terms as syntactic objects from their semantics, which is also defined by means of terms. Although the notation for these latter terms is the same as for object-level terms, they are meta-level objects, and have to be interpreted in the mathematical context of the definition.

Explicit quoting of terms is not a common practice in the areas of term rewriting and algebraic specification. But in some situations, both kinds of terms will appear together in formulas, and then the exact distinction is important. When it is clear from the context, however, that a term is to be considered as belonging to object-level, the quotes $\ulcorner \cdot \urcorner$ will be dropped in order to stay closer to conventional notation.

Definition 3.4 ($\#comp\,(t)$)

> Let $t \in T\,(\Sigma, V)$. $\boldsymbol{\#comp\,(t)}$ denotes the number of direct subterms of t, i. e. $\#comp\,(t) =_{df} 0$ if t is a variable or a constant, and $\#comp\,(\ulcorner f(t_1, \ldots, t_n)\urcorner) =_{df} n$ if $f : S_1, \ldots, S_n \rightarrow S$ for some $n \in \mathbb{N}$ and $S_1, \ldots, S_n, S \in \mathcal{S}$.

Def. 3.3 only defines terms in prefix notation with parentheses. But the usual infix, postfix or "mixfix"[1] notation will also be used if this more convenient.

A general assumption throughout this work will be that all signatures Σ are **sensible** [72], i. e. that $T\,(\Sigma)_S \neq \emptyset$ for all sorts $S \in \mathcal{S}$ (in words: for each sort, there exists a ground term of that sort).

Definition 3.5 (occurrence in a term)

> Let $t \in T\,(\Sigma, V)$. The set $\boldsymbol{occ\,(t)}$ of **occurrences** in t is the smallest prefix-closed set of lists of natural numbers satisfying:
>
> (1) $\varepsilon \in occ\,(t)$.
>
> (2) If $t = \ulcorner f\,(t_1, \ldots, t_n)\urcorner$ and $u \in occ\,(t_i)$ for some $i \in [n]$, then $i \cdot u \in occ\,(t)$.

Definition 3.6 (subterm at an occurrence)

> Let $t, t_1, t_2 \in T\,(\Sigma, V)$ and $u \in occ\,(t)$. Then $\boldsymbol{t/u}$ is the subterm of t at occurrence u:

$$t/u =_{df} \begin{cases} t & \text{, if } u = \varepsilon \\ t_i/u' & \text{, if } t = \ulcorner f\,(t_1, \ldots, t_n)\urcorner \text{ and } u = i \cdot u' \text{ for some } i \in [n] \text{ and } u' \in \mathbb{N}^* \end{cases}$$

> t_2 is a **proper subterm** of t_1 ($\boldsymbol{t_2 \lhd t_1}$) iff there exists a $u \in occ\,(t_1) \setminus \{\varepsilon\}$ such that $t_1/u = t_2$.

[1]An example for an operator normally used with mixfix notation is the conditional operator of many programming languagues. It could be defined as `if _ then _ else _` : *Condition, Statement, Statement* $\rightarrow$ *Statement*, with the intended meaning (as e. g. in OBJ3, cf. Goguen and Winkler [57]) that the i-th argument be written in the place of the i-th "_".

Example 3.7 (occurrences and subterms)

Consider a signature Σ with sorts A, B, C; let x_A, x_B and x_C be variables for these sorts, and let there be the following operators in Σ:

$$\begin{array}{l} f : A, B \to C \\ g : B, C \to A \\ g : A \to C \\ h : B, C \to B \end{array}$$

Then for $t =_{df} f(g(h(x_B, x_C), g(x_A)), h(x_B, f(x_A, x_B))) \in T(\Sigma, V)_C$ we have:

$$\begin{array}{rcl} t/\varepsilon & = & t \\ t/112 & = & x_C \\ t/22 & = & f(x_A, x_B) \end{array}$$

□

Definition 3.8 (subterm replacement)

Let $t, t' \in T(\Sigma, V)$ and $u \in occ(t)$ such that $type(t/u) = type(t')$. Then $\boldsymbol{t[u \leftarrow t']}$ is the term that results from t by replacing t/u by t':

$$t[u \leftarrow t'] =_{df} \begin{cases} t' \text{, if } u = \varepsilon \\ \ulcorner f(t_1, \ldots, t_{i-1}, \llcorner t_i[u' \leftarrow t'] \lrcorner, t_{i+1}, \ldots, t_n) \urcorner \\ \quad \text{, if } t = \ulcorner f(t_1, \ldots, t_n) \urcorner \text{ and } u = i \cdot u' \text{ for some } i \in [n] \text{ and } u' \in \mathbb{N}^* \end{cases}$$

Example 3.9 (subterm replacement)

Consider the signature introduced in Example 3.7. If $t_1 =_{df} g(g(x_B, x_C))$, then $type(t/22) = type(t_1) = C$, and

$$t[22 \leftarrow t_1] = f(g(h(x_B, x_C), g(x_A)), h(x_B, g(g(x_B, x_C))))$$

□

Definition 3.10 (congruence relation)

An equivalence relation $\sim$ on $T(\Sigma, V)$ is a **congruence relation** iff

$$\forall t, t' \in T(\Sigma, V)\ \forall u \in occ(t) : t' \sim t/u \;\Rightarrow\; t[u \leftarrow t'] \sim t$$

If $\sim$ is a congruence relation on $T(\Sigma, V)$, the following holds for all operators $f \in \Sigma$ and $t, t' \in T(\Sigma, V)$:

$$t \sim t' \;\Rightarrow\; \ulcorner f(\ldots t \ldots) \urcorner \sim \ulcorner f(\ldots t' \ldots) \urcorner$$

Definition 3.11 (substitution)

A **substitution** is an $\mathcal{S}$-sorted partial function $\sigma : V \nrightarrow T(\Sigma, V)$ with finite domain $\text{dom}(\sigma)$ such that $\forall v \in \text{dom}(\sigma) : \mathit{type}(v) = \mathit{type}(\sigma(v))$. The set of all substitutions is denoted by $\boldsymbol{Subst}(\boldsymbol{\Sigma}, \boldsymbol{V})$. A substitution σ_1 is an **extension of a substitution** σ iff $\sigma \subseteq \sigma_1$. If $\text{rng}\,\sigma \subseteq T(\Sigma)$, σ is called a **ground substitution**.

A substitution $\sigma \in \mathit{Subst}(\Sigma, V)$ with $\text{dom}(\sigma) = \{v_1, \ldots, v_n\}$ and $\sigma(v_i) = t_i$ for $i \in [n]$ will be written as $\boldsymbol{\sigma = [\,t_1/v_1, \ldots, t_n/v_n\,]}$. Substitution application is usually written in postfix notation.

The homomorphic extension of substitutions onto $T(\Sigma, V)$ is defined in the usual way: Let $t \in T(\Sigma, V)$. Then

$$t\sigma =_{df} \begin{cases} v\sigma & \text{, if } t = v \in \text{dom}(\sigma) \\ t & \text{, if } t \in V \setminus \text{dom}(\sigma) \text{ or } t \text{ is a constant} \\ \ulcorner f(\llcorner t_1\sigma \lrcorner, \ldots, \llcorner t_n\sigma \lrcorner)\urcorner & \text{, if } t = \ulcorner f(t_1, \ldots, t_n)\urcorner, n \geq 1 \end{cases}$$

If $t_1 = t_2\sigma$ for $t_1, t_2 \in T(\Sigma, V)$ and $\sigma \in \mathit{Subst}(\Sigma, V)$, t_1 is called an **instance of $\boldsymbol{t_2}$**, and if $t_1 \in T(\Sigma)$, it is called a **ground instance**.

If $\sigma, \tau \in \mathit{Subst}(\Sigma, V)$, the composition of σ and τ (written as $\boldsymbol{\sigma\tau}$) is defined by: $\forall t \in T(\Sigma, V) : t(\sigma\tau) =_{df} (t\sigma)\tau$.

Definition 3.12 (extra variable)

Let $t \in T(\Sigma, V)$ and $v \in V$. v is an **extra variable with respect to $\boldsymbol{t}$** iff v does not occur as a subterm of t, i. e. $v \notin \mathit{var}(t)$.

Definition 3.13 (Bool, predicate term)

A sort that will always be assumed to be included in the set of sorts $\mathcal{S}$ is the sort Bool of representations of truth values. Together with this sort, the usual relational and propositional operators will be included. In particular, for each sort $S \in \mathcal{S}$ there will be an equality operator $= : S, S \rightarrow \mathsf{Bool}$.

Terms from $T(\Sigma, V)_{\mathsf{Bool}}$ will be called **predicate terms**.

Definition 3.14 (Σ-equation)

A **$\boldsymbol{\Sigma}$-equation** is a term $\ulcorner \lambda = \rho \urcorner$ where $\lambda, \rho \in T(\Sigma, V)_S$ for some $S \in \mathcal{S}$. An **equational theory** is a set of Σ-equations.

Definition 3.15 (Σ-formula)

The set $\boldsymbol{WFF(\Sigma)}$ of **$\boldsymbol{\Sigma}$-formulas** is the least set satisfying the following properties:

(1) every Σ-equation is in $\mathit{WFF}(\Sigma)$;
(2) if $G, H \in \mathit{WFF}(\Sigma)$, then $\ulcorner \neg G \urcorner, \ulcorner (G \wedge H) \urcorner \in \mathit{WFF}(\Sigma)$;
(3) if $x \in V_S$ and $G \in \mathit{WFF}(\Sigma)$, then $\ulcorner (\forall x \in S : G) \urcorner \in \mathit{WFF}(\Sigma)$.

Further logical operators such as $\vee$, $\Rightarrow$, $\Longleftrightarrow$ and $\exists$ are defined as abbreviations in the usual way. The usual precedence rules will be taken for granted and hence most of the brackets in formulas will be omitted.

Notation 3.16 (representation, M')

Let M be a class of mathematical objects such that all elements of M can be represented in the set of terms $T(\Sigma, V)$. Then the set of these representations is denoted by $\boldsymbol{M'_{(\Sigma,V)}}$, or, if Σ and V are clear from the context, just by $\boldsymbol{M'}$.

Representations are defined to allow reasoning about mathematical objects on a syntactic level. No assumptions are being made about uniqueness of representations; if a meta-level element of M has more than one object-level representation in $T(\Sigma, V)$ (which usually will be the case), then all of them are elements of M'.

Example 3.17

If the signature Σ includes a sort Nat that is intended to model the natural numbers $\mathbb{N}$, all Nat-sorted terms belong to the set $\mathbb{N}'$, i. e. $\mathbb{N}' = T(\Sigma, V)_{\mathsf{Nat}}$.

□

Definition 3.18 (term rewriting system)

A term rewriting system (**TRS**) over $T(\Sigma, V)$ is a finite set of **rewrite rules** $\ulcorner \lambda \longrightarrow \rho \urcorner \in T(\Sigma, V) \times T(\Sigma, V)$ such that $\mathit{var}(\rho) \subseteq \mathit{var}(\lambda)$ and $\mathit{type}(\rho) = \mathit{type}(\lambda)$. λ is called the **left-hand side** of the rule, and ρ is called the **right-hand side**.

For a TRS $\mathcal{R}$ over $T(\Sigma, V)$, the **rewriting relation** $\boldsymbol{\rightarrow_{\mathcal{R}}}$ is defined as follows: For terms $t_1, t_2 \in T(\Sigma, V)$, $t_1 \rightarrow_{\mathcal{R}} t_2$ iff there exists a rule $\lambda \longrightarrow \rho \in \mathcal{R}$, an occurrence u in t_1, and a substitution $\sigma \in \mathit{Subst}(\Sigma, V)$ such that $\lambda\sigma = t_1/u$ and $t_2 = t_1[u \leftarrow \rho\sigma]$. If $u = \varepsilon$, the rule is said to be **applied outermost** in t_1.

Definition 3.19 (f-rule)

Let $\mathcal{R}$ be a TRS and $f \in \Sigma$. A rule from $\mathcal{R}$ is called an $\boldsymbol{f}$**-rule** iff its left-hand side is an f-term.

Definition 3.20 ($t_1 \downarrow_{\mathcal{R}} t_2$)

Let $\mathcal{R}$ be a TRS, and let $t_1, t_2 \in T(\Sigma, V)$. Then $\boldsymbol{t_1 \downarrow_{\mathcal{R}} t_2}$ (t_1 and t_2 have a **common reduct**) iff there is a $t \in T(\Sigma, V)$ for which $t_1 \xrightarrow{*}_{\mathcal{R}} t \; {}_{\mathcal{R}}\xleftarrow{*} t_2$.

Definition 3.21 (normal form)

Let $\mathcal{R}$ be a TRS over $T(\Sigma, V)$, let $t, t_1 \in T(\Sigma, V)$. t is a **normal form with respect to** $\boldsymbol{\mathcal{R}}$ iff there exists no $t' \in T(\Sigma, V)$ with $t \rightarrow_{\mathcal{R}} t'$. (This is also written as $\boldsymbol{t \not\rightarrow_{\mathcal{R}}}$.) t_1 is a **normal form of** $\boldsymbol{t}$ iff $t \xrightarrow{*}_{\mathcal{R}} t_1$ and t_1 is a normal form with respect to $\mathcal{R}$.

Definition 3.22 (confluence, termination)

Let $\mathcal{R}$ be a TRS over $T(\Sigma, V)$.

(1) $\mathcal{R}$ is **confluent** iff
$\forall t_1, t_2, t_3 \in T(\Sigma, V) : t_2 \ {}_{\mathcal{R}}\!\xleftarrow{*} t_1 \xrightarrow{*}_{\mathcal{R}} t_3 \ \Rightarrow\ (\exists t_4 \in T(\Sigma, V) : t_2 \xrightarrow{*}_{\mathcal{R}} t_4 \ {}_{\mathcal{R}}\!\xleftarrow{*} t_3)$

(2) $\mathcal{R}$ is **(finitely) terminating** iff there exists no infinite sequence $\{t_i\}_{i \in \mathbb{N}}$ in $T(\Sigma, V)$ such that $\forall i \in \mathbb{N} : t_i \rightarrow_{\mathcal{R}} t_{i+1}$.

(3) $\mathcal{R}$ is **complete** iff $\mathcal{R}$ is confluent and terminating.

The completeness defined above has nothing to do with the completeness of a proof system (which is complete iff all valid formulas are provable). If we want to make the distinction clear, we will also write **TRS-complete** resp. **logically complete** for these two notions.

Theorem 3.23 (existence and uniqueness of normal forms)

Let $\mathcal{R}$ be a TRS over $T(\Sigma, V)$.

(1) If $\mathcal{R}$ is confluent, every $t \in T(\Sigma, V)$ has at most one normal form with respect to $\mathcal{R}$.
(2) If $\mathcal{R}$ is terminating, every $t \in T(\Sigma, V)$ has a normal form.
(3) If $\mathcal{R}$ is complete, every $t \in T(\Sigma, V)$ has exactly one normal form.

3.1.2 Models for Term Rewriting Systems

Term rewriting systems of the form presented in Section 3.1.1 together with their signature are special forms of algebraic specifications. This section will mainly follow Wirsing [115] in defining the semantics of such specifications.

The semantic counterpart of a signature Σ is a Σ-algebra:

Definition 3.24 ($\langle \mathcal{S}, \Sigma \rangle$-algebra)

Let $\langle \mathcal{S}, \Sigma \rangle$ be a signature. An **$\langle \mathcal{S}, \Sigma \rangle$-algebra** A consists of a family $\{A_S \mid S \in \mathcal{S}\}$ of subsets of A, called **carriers** of A, and a function $f_A^{w,S} : A_w \rightarrow A_S$ for each $f \in \Sigma_{w,S}$, where A_w is a one-point set if $w = \varepsilon$ and $A_w = A_{S_1} \times \ldots \times A_{S_n}$ if $w = S_1 \ldots S_n$ for some $n \geq 1$. If the set of sorts is clear from the context, instead of $\langle \mathcal{S}, \Sigma \rangle$-algebra also just **$\Sigma$-algebra** will be used, and if $\langle w, S \rangle$ is clear, $\boldsymbol{f_A}$ instead of $f_A^{w,S}$.

Alg($\mathcal{S}, \Sigma$) denotes the class of all Σ-algebras.

Definition 3.25 ($\langle \mathcal{S}, \Sigma \rangle$-homomorphism)

Let $\langle \mathcal{S}, \Sigma \rangle$ be a signature, and let A, B be $\langle \mathcal{S}, \Sigma \rangle$-algebras. An **$\langle \mathcal{S}, \Sigma \rangle$-homomorphism** $h : A \rightarrow B$ is an $\mathcal{S}$-sorted function $h = \{h_S : A_S \rightarrow B_S\}$ such that

- $\forall w \in \mathcal{S}^+ \, \forall S \in \mathcal{S} \, \forall f \in \Sigma_{w,S} \, \forall \overline{a} \in A_w : h_S(f_A(\overline{a})) = f_B(h_w(\overline{a}))$
  where for $n \geq 1, w = S_1 \ldots S_n$ and $\overline{a} = \langle a_1, \ldots, a_n \rangle : h_w(\overline{a}) =_{df} \langle h_{S_1}(a_1), \ldots, h_{S_n}(a_n) \rangle$.
- $\forall S \in \mathcal{S} \, \forall f \in \Sigma_{\varepsilon,S} : h_S(f_A) = f_B$.

If $\mathcal{S}$ is clear, h will also be called a **Σ-homomorphism**. A bijective Σ-homomorphism (i. e. a Σ-homomorphism consisting only of bijective functions) is called **Σ-isomorphism**.

Definition 3.26 (initial and terminal $\langle \mathcal{S}, \Sigma \rangle$-algebra)

Let $\langle \mathcal{S}, \Sigma \rangle$ be a signature. An $\langle \mathcal{S}, \Sigma \rangle$-algebra A is **initial** in a class K of $\langle \mathcal{S}, \Sigma \rangle$-algebras iff there is for each $\langle \mathcal{S}, \Sigma \rangle$-algebra $B \in K$ a unique $\langle \mathcal{S}, \Sigma \rangle$-homomorphism $h : A \rightarrow B$. A is **terminal** in K iff there is for each $\langle \mathcal{S}, \Sigma \rangle$-algebra $B \in K$ a unique $\langle \mathcal{S}, \Sigma \rangle$-homomorphism $h' : B \rightarrow A$.

Lemma 3.27 (cf. Wirsing [115])

Initial and terminal Σ-algebras are unique up to isomorphism.

Lemma 3.28 (term algebra)

If $T(\Sigma, V)_S \neq \emptyset$ for all $S \in \mathcal{S}$, then the so-called **term algebra $\boldsymbol{T(\Sigma, V)}$**) (also denoted by forms a Σ-algebra with carrier set $T(\Sigma, V)_S$ for all $S \in \mathcal{S}$ and $f_{T(\Sigma,V)}(t_1, \ldots, t_n) =_{df} \ulcorner f(t_1, \ldots, t_n) \urcorner$ for each $f : S_1, \ldots, S_n \rightarrow S \in \Sigma$ and $t_i \in T(\Sigma, V)_{S_i}$ for $i \in [n]$. $\boldsymbol{T(\Sigma)} =_{df} T(\Sigma, \emptyset)$ is also a Σ-algebra (the **ground term algebra**).

The latter fact follows from the general assumption that Σ is a sensible signature (see p. 17).

Definition 3.29 (valuation, interpretation of a term)

Let A be a Σ-algebra.

(1) A **valuation** is a function $v : V \rightarrow A$.

(2) Let v be a valuation. The **interpretation of a term $\boldsymbol{t}$ in $\boldsymbol{A}$ with respect to $\boldsymbol{v}$** is a function $\boldsymbol{v^*} : T(\Sigma, V) \rightarrow A$ defined inductively by:

- $v^*(\ulcorner x \urcorner) =_{df} v(\ulcorner x \urcorner)$ for all $x \in V$
- $v^*(\ulcorner f(t_1, \ldots, t_n) \urcorner) =_{df} f_A(v^*(t_1), \ldots, v^*(t_n))$
  for all $n \in \mathbb{N}_0, f : S_1, \ldots, S_n \rightarrow S \in \Sigma$ and $t_i \in T(\Sigma, V)_{S_i}$ for $i \in [n]$

Lemma 3.30 (v^* is a Σ-homomorphism)

Let A be a Σ-algebra and v a valuation. Then v^* is a Σ-homomorphism which is (for given V and A) the unique Σ-homomorphic extension of v to $T(\Sigma, V)$. If $t \in T(\Sigma)$, then $v^*(t)$ is identical for all valuations v; in this case, it is denoted by $\boldsymbol{t_A}$.

Corollary 3.31 ($T(\Sigma)$ is initial)

Let Σ be a sensible signature. Then the ground term algebra $T(\Sigma)$ is initial in the class of all Σ-algebras.

Again, this follows from the assumption that Σ is sensible.

Definition 3.32 (term generated Σ-algebra)

A Σ-algebra A is **term generated** iff, for each $S \in \mathcal{S}$ and $a \in A_S$, there is a $t \in T(\Sigma)_S$ with $a = t_A$, i. e. if $v^* : T(\Sigma) \to A$ is surjective for a valuation v (and hence, by Lemma 3.30, for all valuations v).

The class of all term-generated Σ-algebras is denoted by $\boldsymbol{Gen(\Sigma)}$.

Definition 3.33 (satisfaction of a formula)

Let A be a Σ-algebra, G a Σ-formula, E a set of Σ-formulas, and v a valuation. The relation **A satisfies G with respect to v ($A, v \models G$)** is defined inductively by:

(1) $A, v \models \ulcorner t = t' \urcorner$ iff $v^*(t) = v^*(t')$ for all terms t, t' of the same type.[2]

(2) $A, v \models \ulcorner \neg G \urcorner$ iff $(A, v \models G)$ does not hold.

(3) $A, v \models \ulcorner G \wedge H \urcorner$ iff $A, v \models G$ and $A, v \models H$.

(4) $A, v \models \ulcorner \forall x \in S : G \urcorner$ iff $A, v_1 \models G$ for all $v_1 : V \to A$ with $v_1(y) = v(y)$ for all $y \neq x$.

A satisfies G ($A \models G$) iff $A, v \models G$ for all valuations v. G is **valid** in a class K of Σ-algebras (**$K \models G$**) iff $A \models G$ for all $A \in K$.

The sets of Σ-algebras that satisfy a set of formulas are denoted by

$$\boldsymbol{Alg(\Sigma, E)} =_{df} \{A \in Alg(\Sigma) \mid A \models G \text{ for all } G \in E\}$$

$$\boldsymbol{Gen(\Sigma, E)} =_{df} \{A \in Gen(\Sigma) \mid A \models G \text{ for all } G \in E\}$$

Definition 3.34 (theory)

Let K be a class of Σ-algebras. Then define:

The **theory of K** is:

$$\boldsymbol{Th(K)} =_{df} \{G \in WFF(\Sigma) \mid K \models G\}$$

The **equational theory of K** is:

$$\boldsymbol{Th_{EQ}(K)} =_{df} \{G \mid G \text{ is a } \Sigma\text{-equation}, K \models G\}$$

If $K = \{A\}$ for a single Σ-algebra A, then $Th_{EQ}(A)$ and $Th(A)$ will also be written instead of $Th_{EQ}(K)$ and $Th(K)$, respectively.

[2]Note that "=" in $\ulcorner t = t' \urcorner$ is a syntactic entity denoting an (object-level) operator, whereas in $v^*(t) = v^*(t')$ it is a semantic entity denoting a (meta-level) predicate, i. e. an operation.

Definition 3.35 (algebraic specification)

Let E be a set of first-order axioms (i. e. $E \subseteq \mathit{WFF}(\Sigma)$). Then $\langle\, \Sigma, E \,\rangle$ is an **algebraic specification.** If E is a set of equations, $\langle\, \Sigma, E \,\rangle$ is called an **equational specification.**

For the definition of the semantics of an algebraic specification $\langle\, \Sigma, E \,\rangle$ there are three approaches:

- **loose semantics**: the set of models is

  $\boldsymbol{Mod(\Sigma, E)} =_{df} \mathit{Gen}(\Sigma, E)$, i. e. the set of all term generated algebras that satisfy all formulas in E.
- **initial semantics**: the set of initial models is

  $\boldsymbol{I(\Sigma, E)} =_{df} \{I \in \mathit{Gen}(\Sigma, E) \mid I \text{ is initial in } \mathit{Gen}(\Sigma, E)\}$.
- **terminal semantics**: the set of terminal models is

  $\boldsymbol{Z(\Sigma, E)} =_{df} \{Z \in \mathit{Gen}(\Sigma, E) \mid Z \text{ is terminal in } \mathit{Gen}(\Sigma, E)\}$.

Term-generated initial and terminal algebras can be characterized by the equalities among ground terms they satisfy:

Lemma 3.36

Let K be a class of Σ-algebras and let $B \in K$ be term-generated.

(1) B is initial in K iff for all $t, t' \in T(\Sigma)$

$$B \models \ulcorner t = t' \urcorner \iff \forall A \in K : A \models \ulcorner t = t' \urcorner$$

(2) B is terminal in K iff for all $t, t' \in T(\Sigma)$

$$B \models \ulcorner t = t' \urcorner \iff \exists A \in K : A \models \ulcorner t = t' \urcorner$$

Initial and loose semantics are most commonly used; see [115] for a discussion of the differences. Examples for specification or proof tools using these approaches are OBJ3 [57] or PVS [104] for the initial one and the Larch Prover [51] or RAP [73] for the loose one.

A TRS $\mathcal{R}$ over Σ and V corresponds to an equational specification $\langle\, \Sigma, E_{\mathcal{R}} \,\rangle$, where $\boldsymbol{E_{\mathcal{R}}} =_{df} \{\ulcorner \lambda = \rho \urcorner \mid \ulcorner \lambda \longrightarrow \rho \urcorner \in \mathcal{R}\}$. This equational theory corresponds to the rewriting relation of $\mathcal{R}$ in a natural way:

Theorem 3.37 (cf. Ehrig and Mahr [39], Corollary 5.15)

Let $\mathcal{R}$ be a TRS and $t_1, t_2 \in T(\Sigma, V)$. Then

$$\mathit{Mod}(\Sigma, E_{\mathcal{R}}) \models \ulcorner t_1 = t_2 \urcorner \iff I(\Sigma, E_{\mathcal{R}}) \models \ulcorner t_1 = t_2 \urcorner \iff t_1 \overset{*}{\longleftrightarrow}_{\mathcal{R}} t_2$$

$\boldsymbol{Th_{EQ}(\mathcal{R})} =_{df} Th_{EQ}(\mathit{Mod}(\Sigma, E_{\mathcal{R}}))$ is called the **equational theory generated by $\mathcal{R}$.**

Corollary 3.38

Let $\mathcal{R}$ be a complete TRS. Then $\mathcal{R}$ provides a decision procedure for equality in the equational theory generated by $\mathcal{R}$.

Proof

In order to check whether $\ulcorner t_1 = t_2 \urcorner \in Th_{EQ}(\mathcal{R})$ for given $t_1, t_2 \in T(\Sigma, V)$, one only has to generate the (according to Theorem 3.23) unique normal forms of t_1 and t_2. Iff they are identical, $t_1 \overset{*}{\longleftrightarrow}_{\mathcal{R}} t_2$ holds, and hence by Theorem 3.37 also $Mod(\Sigma, E_{\mathcal{R}}) \models \ulcorner t_1 = t_2 \urcorner$, i. e. $\ulcorner t_1 = t_2 \urcorner \in Th_{EQ}(\mathcal{R})$.

□

3.1.3 Confluence and Termination

Corollary 3.38 shows that it is very desirable to work with complete rewrite systems, and therefore criteria are needed for confluence and termination. First consider termination.

In general, it is undecidable:

Theorem 3.39 (Dauchet [27])

Termination of rewrite systems is undecidable even for systems consisting of just one rewrite rule.

This theorem is proved by assigning a single simulating rewrite rule to Turing machines. The result then follows from the undecidability of the halting problem (cf. Hermes [66]). If rewrite systems are restricted to ground systems (without variables in the rules), however, termination is decidable (cf. Huet and Lankford [71]), but such systems are often not very useful, in particular not in the context of this work.

For proving termination of given rewrite systems, a certain type of well-founded orderings is important:

Definition 3.40 (monotonic ordering, termination ordering)

Let $\succ$ be a partial ordering on $T(\Sigma, V)$. $\succ$ is called **monotonic** (with respect to term structure) iff:

$$\begin{array}{l} \forall t, t_1, t_2 \in T(\Sigma, V)\, \forall u \in occ(t) : \\ \quad type(t_1) = type(t_2) \ \wedge\ t_1 = t/u \ \wedge\ t_1 \succ t_2 \ \Rightarrow\ t \succ t[u \leftarrow t_2] \end{array}$$

If $\succ$ is monotonic, the following holds for all operators f in Σ and all terms t_1 and t_2 with the same type:

$$t_1 \succ t_2 \ \Rightarrow\ \ulcorner f(\ldots t_1 \ldots) \urcorner \succ \ulcorner f(\ldots t_2 \ldots) \urcorner$$

A **termination ordering** is a well-founded monotonic ordering on $T(\Sigma, V)$.

Termination orderings are exactly what is needed to prove termination of a TRS:

Theorem 3.41 (cf. Dershowitz [32])

A TRS $\mathcal{R}$ over $T(\Sigma, V)$ is terminating iff there exists a termination ordering $\succ$ over $T(\Sigma, V)$ such that $\lambda\sigma \succ \rho\sigma$ for all $\lambda \longrightarrow \rho \in \mathcal{R}$ and all $\sigma \in \mathit{Subst}(\Sigma, V)$.

Theorem 3.41 reduces termination proofs to construction of suitable termination orderings. By now, there exist many different approaches; see Dershowitz [32] or Steinbach [112] for an overview.

The usual method to prove confluence of a rewrite system uses critical pairs. In order to be able to introduce this notion properly, unification must be defined first:

Definition 3.42 (unification)

$t_1, t_2 \in T(\Sigma, V)$ are called **unifiable** iff there exists a $\sigma \in \mathit{Subst}(\Sigma, V)$ with $t_1\sigma = t_2\sigma$. Such a σ is called a **unifier** for t_1 and t_2.

Definition 3.43 (subsumption)

Let $\sigma, \tau \in \mathit{Subst}(\Sigma, V)$. σ is called **more general** than τ (written as $\boldsymbol{\sigma \leq \tau}$) iff there exists a $\rho \in \mathit{Subst}(\Sigma, V)$ with $\sigma\rho = \tau$. If $\sigma \leq \tau$, σ is said to **subsume** τ.

Lemma 3.44 (cf. Hofbauer and Kutsche [70])

Let $\sigma, \tau \in \mathit{Subst}(\Sigma, V)$. If $\sigma \leq \tau$ and $\tau \leq \sigma$, then there is a renaming substitution $\rho \in \mathit{Subst}(\Sigma, V)$ (i. e. $\rho(v) \in V$ for all $v \in V$) such that $\sigma\rho = \tau$. In this case, we write $\boldsymbol{\sigma \sim \tau}$.

Theorem 3.45 (Unification Theorem, cf. Robinson [103])

Let $t_1, t_2 \in T(\Sigma, V)$. If t_1 and t_2 are unifiable, then there exists a unifier $\sigma \in \mathit{Subst}(\Sigma, V)$ for t_1 and t_2 such that for all unifiers $\tau \in \mathit{Subst}(\Sigma, V)$ for t_1 and t_2 we have $\sigma \leq \tau$. Such a σ is called a **most general unifier (mgu)** for t_1 and t_2. If σ_1 and σ_2 are mgu's for t_1 and t_2 then $\sigma_1 \sim \sigma_2$, i. e. mgu's are unique up to renaming of variables.

Definition 3.46 (critical pair)

Let $\mathcal{R}$ be a TRS, and let $\ulcorner \lambda_1 \longrightarrow \rho_1 \urcorner, \ulcorner \lambda_2 \longrightarrow \rho_2 \urcorner \in \mathcal{R}$. Without loss of generality it may be assumed that $\mathit{var}(\ulcorner \lambda_1 \longrightarrow \rho_1 \urcorner) \cap \mathit{var}(\ulcorner \lambda_2 \longrightarrow \rho_2 \urcorner) = \emptyset$.[3]

Let $u \in \mathit{occ}(\lambda_1)$ be a non-variable occurrence (i. e. $\lambda_1/u \notin V$), and let σ be an mgu of λ_1/u and λ_2. Then the equation

$$\rho_1\sigma = (\ \lambda_1[u \leftarrow \rho_2]\ )\sigma$$

is called a **critical pair** with respect to $\mathcal{R}$.

[3]This can always be achieved by renaming the variables in one of the rules.

Example 3.47

Consider the system $\mathcal{R}_1$ that consists of the rules

$$r_1 : x + 0 \longrightarrow x$$
$$r_2 : s(y) + z \longrightarrow s(y + z)$$

A critical pair of these rules is the equation

$$s(y) = s(y + 0)$$

(the occurrence in r_1 being ε and the mgu $[s(y)/x, 0/z]$).

□

Critical pairs characterize exactly those overlappings of left-hand sides of rules that may lead to violations of confluence. This is expressed by

Theorem 3.48 (cf. Knuth and Bendix [81])

Let $\mathcal{R}$ be a terminating TRS. $\mathcal{R}$ is confluent iff $t_1 \downarrow t_2$ for all critical pairs $\ulcorner t_1 = t_2 \urcorner$ with respect to $\mathcal{R}$.

For a finite rewrite system there are only finitely many critical pairs. If the system is terminating, then moreover the finitely many normal forms of all the terms in these pairs can be effectively computed. This proves

Corollary 3.49

Confluence of terminating rewrite systems is decidable.

Because termination in general is undecidable (see Theorem 3.39), however, confluence in general is also undecidable (see Klop [80]).

Since the critical pairs of a rewrite system $\mathcal{R}$ belong to its equational theory, this theory remains unchanged if the critical pairs are ordered into rewrite rules and added to $\mathcal{R}$. This is the idea of *completion*; if all critical pairs can be added to $\mathcal{R}$, then the resulting system is complete by Theorem 3.48 (provided it is terminating) but the equational theory has not changed. Many completion algorithms have been described, the best-known being that of Knuth and Bendix [81] ; for an overview see Buchberger [19]. Note that, due to the undecidability of confluence in general, completion algorithms may fail to terminate if the rewrite system $\mathcal{R}$ is not terminating.

3.2 Term Rewriting and the λ-Calculus

For the simulation of operational semantics definitions, terms of the form

$$\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 \tag{3.1}$$

will be used since they provide a convenient way to give names to intermediate results: x is a name for the result of "evaluating" e_1. (3.1), however, is equivalent to the term

$$(\lambda x.e_2)\, e_1 \tag{3.2}$$

and thus the λ-calculus enters the stage (see Barendregt [7]).

But not the whole of it will be needed since only terms of the very restricted form (3.2) will be used, that is, the application of a λ-abstraction to an argument. By β-reduction, (3.2) becomes

$$e_2[e_1/x] \tag{3.3}$$

where the substitution $[e_1/x]$ is assumed not to introduce clashes in variable names here. So one only has to provide a way to evaluate substitution applications like (3.3). This is not completely trivial, however, because of the condition that there must not be any conflicts in variable names.[4]

In the literature, several calculi have been introduced that only use term rewriting and no other mechanisms for the evalutation of (3.3); an overview is given by Lescanne [85]. Basically, these calculi provide methods to handle substitution applications like (3.2) explicitly and not implicitly on meta-level as it is usually done in the λ-calculus (cf. Barendregt [7]).

In the next two subsections, an example for a calculus dealing with explicit substitutions will be described, viz. the $\lambda\sigma$-calculus as introduced by Abadi et al. [1]. Like in all the calculi mentioned in [85], the key idea for the prevention of problems with variable names is to abolish them completely and use de Bruijn indices instead. So this concept will be introduced first, followed by the substitution manipulation and application operators of the $\lambda\sigma$-calculus and the rules defining them.

In the pure form of [1], however, the $\lambda\sigma$-calculus is not quite useful for the applications in this work. The modifications that are necessary to adapt it to the special problems will be described in the third of the following subsections. Finally, this section will close with a short glance at other methods to combine λ-calculus and term rewriting.

3.2.1 De Bruijn Notation

Semantically, the names of bound variables in λ-terms are completely irrelevant; $\lambda x.e$ can be α-converted into any other term $\lambda y.e[y/x]$ provided y does not occur free in e. The idea of de Bruijn (cf. [18]) is to exploit this fact by using positive integers ("indices") instead of variable names; the integer n then corresponds to the n-th surrounding λ-abstraction:

$$\lambda x.\lambda y.xy \qquad \text{becomes} \qquad \lambda.\lambda.21$$

Of course this interpretation of n has to be respected when an operation such as β-reduction takes place that eliminates a λ. Consider the term $(\lambda.a)b$. β-reduction should replace all occurrences

[4]Technically, not the let operator, but rather the operators to be introduced in Section 3.2.2 that work with substitutions will be included in the underlying signature. The let form (3.1), however, will be used as a more intuitive representation of (3.2) or (3.3).

of 1 in a by b. But there may also be free occurrences of other indices 2, 3 ... in a, as in $\lambda.21$ where 2 refers to a λ outside of this term. All these free indices must be decremented by 1 since β-reduction removes the λ around a. This could be expressed by the infinite substitution $[1/2, 2/3, 3/4, \ldots]$. Together with the replacement for 1, this gives the following first attempt for a modelling of the β rule:

$$(\lambda.a)b \twoheadrightarrow_\beta a[b/1, 1/2, 2/3, 3/4, \ldots] \tag{3.4}$$

Note, however, that the substitution application operator $\_[\_]$ used in (3.4) has not been defined so far. The main problem in its definition is the avoidance of variable conflicts. This is usually done by proper renaming, and in the evaluation of $a[b/1, 1/2, 2/3, 3/4, \ldots]$ one also has to perform something similar. The problem arises when one comes across subterms of the form $\lambda.c$. In c, 1 must not be replaced by b, since this is exactly the variable conflict to be avoided: in $\lambda.c$, 1 refers to a bound variable. So 1 must remain unchanged, and 2 must be replaced by b instead. The other indices in c must be decremented for the same reason as above. Furthermore, all indices in b must be incremented by 1 since b is inserted into a term with an additional λ. All this results in the following rule which gives a part of the definition of the operator $\_[\_]$:

$$(\lambda.c)[b/1, 1/2, 2/3, 3/4, \ldots] \twoheadrightarrow \lambda.(c[1/1, b[2/1, 3/2, 4/3, \ldots]/2, 2/3, 3/4, \ldots]) \tag{3.5}$$

Notice that this is not a rewrite rule because the representation of substitutions is not finite. The $\lambda\sigma$-calculus solution to this problem as given by Abadi et al. [1] will be shown in the next subsection.

3.2.2 The $\boldsymbol{\lambda\sigma}$-calculus of Abadi et al.

Following [1], the first step in defining the **$\boldsymbol{\lambda\sigma}$-calculus** is to fix the syntax of terms and substitutions. For the untyped case[5], this is done in the following way:

Definition 3.50 (syntax of the untyped $\lambda\sigma$-calculus)

The syntax of terms and substitutions for the untyped $\lambda\sigma$-calculus is given by the following BNF-like grammar:

terms:	$a, b ::= 1 \mid a_1\, a_2 \mid \lambda.a \mid a[s]$
substitutions:	$s, t ::= id \mid \uparrow \mid a \cdot s \mid s_1 \circ s_2$

The term constructors provided are the first de Bruijn index (there is no need for the other indices, see below), application, λ-abstraction, and substitution application. The four substitution constructors[6] provide a way to give finite representations for infinite substitutions like the ones in (3.4) and (3.5). Their intended meaning is the following:

(1) ***id*** stands for the **identity substitution** $[i/i \mid i \geq 1]$.

[5] Untyped rewriting fits into the framework of Section 3.1 by assuming a universal sort as the type of all terms.

[6] Note that the substitutions occurring in this section differ from those in Def. 3.11 in only replacing de Bruijn indices, but no other variables. Moreover, they may not have finite representations of the form $[t_1/v_1, \ldots, t_n/v_n]$.

(2) $\uparrow$ stands for the **shift substitution** $[i+1/i \mid i \geq 1]$.
Since $1[\uparrow] = 2, (1[\uparrow])[\uparrow] = 3$, and so on, it suffices to just have the index 1; n is coded by $1[\uparrow^(n-1)]$, where $\uparrow^n$ is the composition of n shifts. Define $\uparrow^0 =_{df} id$.

(3) $\boldsymbol{a \cdot s}$ denotes the **cons** of a onto s, i. e. the substitution $[a/1, i[s]/i+1 \mid i \geq 1]$.
For example
$a \cdot id = [a/1, i[id]/i+1 \mid i \geq 1] = [a/1, i/i+1 \mid i \geq 1] = [a/1, 1/2, 2/3, \ldots]$
$1 \cdot \uparrow = [1/1, i[\uparrow]/i+1 \mid i \geq 1] = [1/1, i+1/i+1 \mid i \geq 1] = id$

(4) $\boldsymbol{s \circ u}$ denotes the **composition** of s and u: $[(i[s])[u]/i \mid i \geq 1]$.
For example:
$id \circ u = [(i[id])[u]/i \mid i \geq 1] = [i[u]/i \mid i \geq 1] = u$
$\uparrow \circ (a \cdot s) = [(i[\uparrow])[a \cdot s]/i \mid \imath \geq 1] = [(i+1)[a \cdot s]/i \mid i \geq 1] = [i[s]/i \mid i \geq 1] = s$

Using these definitions, (3.4) and (3.5) become

$$(\lambda.a)b \twoheadrightarrow a[b \cdot id] \tag{3.6}$$

$$(\lambda.c)[s] \twoheadrightarrow \lambda.(c[1 \cdot (s \circ \uparrow)]) \tag{3.7}$$

Note that the above description only gives an informal motivation for the substitution operators. The formal definition of their semantics is given by the following set of equations that defines the equational theory of the untyped $\lambda\sigma$-calculus. There is one rule Beta which is the equivalent of the classical β-reduction rule, and fourteen rules for the evaluation of substitutions and substitution applications, together called Sigma.

Definition 3.51 (rules of the untyped $\lambda\sigma$-calculus)

As in Def. 3.50, let a and b be variables for terms, and s and t for substitutions. The untyped $\lambda\sigma$-calculus is given by the following set of equations:

Beta	$(\lambda.a)b = a[b \cdot id]$
VarId	$1[id] = 1$
VarCons	$1[a \cdot s] = a$
App	$(ab)[s] = (a[s])(b[s])$
Abs	$(\lambda.a)[s] = \lambda.(a[1 \cdot (s \circ \uparrow)])$
Clos	$(a[s])[t] = a[s \circ t]$
IdL	$id \circ s = s$
ShiftId	$\uparrow \circ id = \uparrow$
ShiftCons	$\uparrow \circ (a \cdot s) = s$
Map	$(a \cdot s) \circ t = a[t] \cdot (s \circ t)$
Ass	$(s_1 \circ s_2) \circ s_3 = s_1 \circ (s_2 \circ s_3)$
Id	$a[id] = a$
IdR	$s \circ id = s$
VarShift	$1 \cdot \uparrow = id$
SCons	$1[s] \cdot (\uparrow \circ s) = s$

All the above equations can be oriented from left to right, yielding an (up to now untyped) rewrite system for the evaluation of λ-terms.

Example 3.52 (application of the $\lambda\sigma$-calculus)

Consider the λ-term $((\lambda.\lambda.1[\uparrow])(\lambda.1))\,(\lambda.1\,1)$. In usual notation with named variables, this term would be written as $((\lambda x.\lambda y.x)\,(\lambda x.x))\,(\lambda x.x\,x)$, which is transformed by β-reduction first into $(\lambda y.(\lambda x.x))\,(\lambda x.x\,x)$, and finally into $\lambda x.x$.

In the $\lambda\sigma$-calculus, this evaluation is performed taking the following steps:

$$\begin{array}{lll}
 & ((\lambda.\lambda.1[\uparrow])(\lambda.1))\,(\lambda.1\,1) & \\
\rightarrow_{\mathsf{Beta}} & ((\lambda.1[\uparrow])[(\lambda.1)\cdot id])\,(\lambda.1\,1) & \\
\rightarrow_{\mathsf{Abs}} & (\lambda.((1[\uparrow])[1\cdot(((\lambda.1)\cdot id)\circ\uparrow)]))\,(\lambda.1\,1) & \\
\rightarrow_{\mathsf{Clos}} & (\lambda.(1[\uparrow\circ(1\cdot(((\lambda.1)\cdot id)\circ\uparrow))]))\,(\lambda.1\,1) & \\
\rightarrow_{\mathsf{ShiftCons}} & (\lambda.(1[((\lambda.1)\cdot id)\circ\uparrow]))\,(\lambda.1\,1) & \\
\rightarrow_{\mathsf{Map}} & (\lambda.(1[(\lambda.1)[\uparrow]\cdot(id\circ\uparrow)]))\,(\lambda.1\,1) & \\
\rightarrow_{\mathsf{VarCons}} & (\lambda.((\lambda.1)[\uparrow]))\,(\lambda.1\,1) & \\
\rightarrow_{\mathsf{Abs}} & (\lambda.(\lambda.(1[1\cdot(\uparrow\circ\uparrow)])))\,(\lambda.1\,1) & \\
\rightarrow_{\mathsf{VarCons}} & (\lambda.(\lambda.1))\,(\lambda.1\,1) & [\,\triangleq(\lambda y.(\lambda x.x))\,(\lambda x.x\,x)\,] \\
\rightarrow_{\mathsf{Beta}} & (\lambda.1)[(\lambda.1\,1)\cdot id] & \\
\rightarrow_{\mathsf{Abs}} & \lambda.(1[1\cdot((\lambda.1\,1)\cdot id)]) & \\
\rightarrow_{\mathsf{VarCons}} & \lambda.1 & [\,\triangleq\lambda x.x\,]
\end{array}$$

□

The rewrite system Beta ∪ Sigma has some pleasant properties with respect to term rewriting:

Theorem 3.53 (Abadi et al. [1])

(1) Sigma is a complete rewriting system.

(2) Beta ∪ Sigma is confluent on closed ground terms and on terms only containing variables for terms, but not for substitutions.

In (2), a term is called **closed** (in the sense of λ-calculus) if it does not contain free de Bruijn indices, i. e. no indices referring to λ's outside of the term (as e. g. in $\lambda.2$).

For the purposes of this work, the weak form of confluence in (2) suffices since variables for substitutions will not occur. There are, however, other calculi that are also confluent on open terms; see Lescanne [85].

Since Sigma is complete, every term a has exactly one Sigma-normal form which will be denoted by $\boldsymbol{\sigma(a)}$ (cf. Theorem 3.23). Of course, Beta ∪ Sigma cannot be expected to be terminating since it is supposed to provide an execution model for the λ-calculus. That this is actually the case is expressed by the next theorem:

Theorem 3.54 (Abadi et al. [1])

(1) Let $a, a_1, \ldots, a_n$ be terms of the $\lambda\sigma$-calculus for some $n \geq 1$. Then the effect of the meta-level substitution $[a_1/1, \ldots, a_n/n]$ on the term a can be calculated with the rewriting system Sigma:

$$a[a_1/1, \ldots, a_n/n] = \sigma(\ulcorner a[\,a_1\cdot\ldots\cdot a_n\cdot\llcorner\uparrow^{(n-1)}\lrcorner\,]\urcorner)$$

(2) A β-reduction step can be implemented by first applying the rule Beta and then calculating the Sigma-normal form.

3.2.3 Modifications

3.2.3.1 Adding Types

The first modification that is necessary to adapt the pure $\lambda\sigma$-calculus to the problems dealt with in this work is the introduction of types. Abadi et al. already show the way how this can be done. Their emphasis in this part, however, lies more on type checking problems than on implementation aspects since they deal with the full λ-calculus and not only with the small part that is needed in the setting of the work on hand.

The de Bruijn form of typed first order terms is derived in the same way as that for untyped terms; e. g.

$$\lambda x : A.\lambda y : B.xy \qquad \text{becomes} \qquad \lambda A.\lambda B.21$$

In principle, there is no problem in adapting Beta and Sigma to the typed situation. Note, however, that one set of these rules is needed for each type that may occur as an argument type, and therefore the whole system may become rather large. Moreover, different de Bruijn indices for different argument types are needed. In the applications, however, mostly only one argument type will be needed, and therefore the typed system Sigma will be essentially the same as the untyped one.

Let S be the sort for the arguments. The $\lambda\sigma$-rewriting system is based on a sort Subst of substitutions, and the operators together with their arities are the following:

$id : \rightarrow \mathsf{Subst}$	identity
$\uparrow : \rightarrow \mathsf{Subst}$	shift substitution
$\_\cdot\_ : S, \mathsf{Subst} \rightarrow \mathsf{Subst}$	cons
$\_\circ\_ : \mathsf{Subst}, \mathsf{Subst} \rightarrow \mathsf{Subst}$	composition

Moreover, for each sort T different from Subst:

$\_[\_] : T, \mathsf{Subst} \rightarrow T$	substitution application

3.2.3.2 Function Constants

In order to be useful for the simulations described in later chapters, the $\lambda\sigma$-calculus must be merged with ordinary term rewriting. As an immediate consequence the terms to which substitutions have to be applied are of a more general kind than those of pure $\lambda\sigma$-calculus as defined above. In particular, the effect of applying substitutions to complex terms $f(t_1, \ldots, t_n)$ has to be defined.

From a higher-order point of view, such a term is just a special case of an application term where the function that is applied is described by a constant operator. Since constants are not affected by substitutions, one would like to have the rule

$$f(t_1, \ldots, t_n)[u] \rightarrow f(t_1[u], \ldots, t_n[u]) \tag{3.8}$$

as an instance of App from Section 3.2.2. But, of course, it does not suffice to have one such rule, because there are no variables for operators in first-order terms. So for each operator $f \in \Sigma$ (and for each of its possible arities, if it is overloaded), the corresponding version of (3.8) must be added to the rewriting system.

Having all these instances of rule (3.8), the rule App itself will not be needed anymore, since there will be no other application terms than those just mentioned. The rules Beta and Abs can be omitted as well, since explicit λ-abstractions will never occur.

For constants $c : \to T$ there could also be a rule of the form (3.8): $c[u] \to c$. But for the handling of the substitution of constants, an alternative method is chosen. Since in the special application it is always known beforehand to which terms substitutions will be applied, each constant (or even each term not containing a $\lambda\sigma$-calculus variable) can be enclosed by a special **protection operator** $\boldsymbol{I}$: $T \to T$ for each type T. For this operator there is not the rule (3.8) but instead:

$$I(t)[u] \to t$$

In this way, there is also no problem with the occurrence of "new" constants that are added to the system only after the rewriting system is set up (for example by actions of the proof tool during proofs with these rewrite rules). There is no need to know the actual names of these constants; it suffices to know they are different from the de Bruijn indices. So those occurrences where these constants may appear can be safely surrounded by the appropriate protection operator.

3.2.4 Merging Term Rewriting and λ-Calculus

Instead of embedding parts of λ-calculus into term rewriting as described in the previous sections, one could alternatively merge the two formalisms and express the desired rules in the combined system. The main advantage of such more elaborate formalisms is that rules may become simpler structured since some of their complexity can be moved into the reduction process itself.

The properties of such systems have already been investigated. Dougherty [36, 37] has examined confluence and termination of systems that include rewrite systems and λ-calculus; under certain conditions, properties of the rewriting system are inherited by the combination. Loria-Saenz and Steinbach [87] have applied techniques for proving termination of rewrite systems to a combination of higher-order rewriting and λ-calculus.

Kahrs [77] presents a generalization of the simple addition of rewriting and λ-calculus. For this purpose, four levels of so-called *λ-rewriting systems* (LRS's) are defined. The complexity of terms and rules increases across these levels, until finally abstractions and applications may also occur on the left-hand sides of rules. The LRS type 2 that is most interesting for our application allows these only on right-hand sides. Confluence and termination of an LRS of type 2 are determined by the rewriting relation, provided the system respects certain (rather reasonable) restrictions. *Combinatory reduction systems* (CRS's) as introduced by Klop [79] are another generalization of both term rewriting and λ-calculus; the rules in such systems are still more general than those in LRS's.

Since all these approaches extend simple rewriting, they cannot be used except with a special-purpose tool that is able to handle the extensions. Kahrs describes the implementation of such systems; but they are intended as an interpreter for LRS's and CRS's, respectively, and not so much as a proof support system. So the use of real extensions of term rewriting is not a suitable alternative if existing proof tools are to be used.

3.3 Controlling Rewriting with Contexts

Normalization of terms with a set of rewrite rules (i. e. rewriting until a normal form is reached) normally does not allow to control how often and in which order certain rules are used for rewriting.[7] The simulation of operational semantics definitions in Chapter 7, however, requires that the rewriting of a certain class F of terms be controlled in a very specific way in order to achieve a faithful simulation of the semantics rules.

This class F is the set of f-terms for some fixed operator $f \in \Sigma_{w,s}$ (for the sake of simpler exposition, assume for the moment $|\, w \,| = 1$), and there are two ways in which the rewriting of such terms must be controllable:

(1) It must be possible to normalize an f-term in such a way that it is only rewritten once with an outermost application of an f-rule, even if more such rules could be applied. The rewriting of subterms must not be affected by this restriction.

(2) It must be allowed to define rules of the form

$$f\,(t_1) \twoheadrightarrow \textsf{if } b \textsf{ then } f\,(t_2) \textsf{ else } f\,(t_1) \qquad (b \in T\,(\Sigma, V)_{\textsf{Bool}}, t_1, t_2 \in T\,(\Sigma, V)_w)$$

in such a way that the rewriting system does not become non-terminating (which it would if rules of this kind were added without modifications).

(1) is the wish to stop normalization automatically at a certain point, and (2) requests the inclusion of a form of conditional rewriting ("only if b holds, rewrite $f(t_1)$ to $f(t_2)$", see e. g. Kaplan [78] or Bergstra and Klop [9]).[8]

The f-terms themselves and the rewriting mechanism as defined in Def. 3.18 shall remain unchanged. So the desired goal can only be reached by extending f-terms with the appropriate control information. For (1), this requires some form of counter that has to be decremented after an f-rule has been applied to an (extended) f-term. For (2), a kind of switch is needed that is turned to "off" when the testing condition b turns out be (more exactly, rewrite to) **false**. Since such a switch is needed for every f-rule in the system, the control information can be gathered in a context of the following form:

Definition 3.55 (context)

Let $f \in \Sigma$, and let $l_1 \twoheadrightarrow r_1, \ldots, l_n \twoheadrightarrow r_n$ be all the f-rules in the rewrite system $\mathcal{R}$. Then a **context for *f*** is an element of $\{0, 1, *\} \times \{\textsf{on}, \textsf{off}\}^n$. For $a \in \{0, 1, *\}$ and $s_1, \ldots, s_n \in \{\textsf{on}, \textsf{off}\}$, the context $\langle\, a, s_1, \ldots, s_n \,\rangle$ is called an ***a*-context**.

Contexts contain a counter component ($0, 1$ or $*$) and a switch for each f-rule. Instead of terms $f\,(t)$, now terms $f\,(t) \,@\, \langle\, a, s_1, \ldots, s_n \,\rangle$ are rewritten, where @ is the special context application operator, $a \in \{0, 1, *\}$ and $s_1, \ldots, s_n \in \{\textsf{on}, \textsf{off}\}$. The intended interpretation for a is:

[7] There are other models of rewriting where some kind of control is possible. In *priority rewriting* e. g. (see Baeten, Bergstra and Klop [4]), the order in which rules are checked for applicability can be changed. In that approach, a rewrite system consists of a *list* of rules rather than of a *set*.

[8] Note, however, that such rules are only requested for f-terms, not for all other terms as well.

- no more top-level rewriting steps, if $a = 0$,
- at most one top-level rewriting step, if $a = 1$,
- no limit on the number of top-level rewriting steps, if $a = *$.

The interpretation for the s_i is that application of the i-th rule is allowed if $s_i = \mathsf{on}$ and disallowed otherwise.

But of course it does not suffice to modify the term that is to be rewritten. The control mechanism must also be built into the f-rules $l_1 \twoheadrightarrow r_1, \ldots, l_n \twoheadrightarrow r_n$. This is done by supplying all f-terms in all l_i and r_j with appropriate contexts. The exact form of these contexts depends on the specific rule, but the following requirements should be met:

(1) For $i \in [n]$, the $(i+1)$-th component of a context corresponds to the switch for the rule $l_i \twoheadrightarrow r_i$. If it is off, the rule should not be applicable. It also should not be applicable, if the counter component is 0. Therefore, the context for l_i should be of the form

$$\langle\, a, s_1, \ldots, s_{i-1}, \mathsf{on}, s_{i+1}, \ldots, s_n \,\rangle$$

where $a \in \{1, *\}$. The restriction that $l_i \twoheadrightarrow r_i$ should only be used if the use of another rule $l_j \twoheadrightarrow r_j$, say, is also allowed can be expressed by letting $s_j = \mathsf{on}$ (the same holds for disallowed rules and $s_j = \mathsf{off}$). If there is no such link, s_j should be a variable; this will be the normal case.

(2) There is also the possibility to have two new rules for one old one. If $l_i \twoheadrightarrow r_i$ is to be used both in single-step and in multi-step mode, it is necessary to have one rule with a 1-context for l_i and one rule with an $*$-context.

(3) If the context for l_i is a 1-context, the context for an f-term on the right-hand side should be a 0-context if it corresponds to a "successful" application. Otherwise it should be a 1-context with the switch for this rule turned off: the i-th rule

$$f\,(t_1) \twoheadrightarrow \mathsf{if}\ b\ \mathsf{then}\ f\,(t_2)\ \mathsf{else}\ f\,(t_1)$$

should become (assume that b, t_1 and t_2 do not contain f-terms)

$$\begin{aligned} &f\,(t_1)\,@\,\langle\, 1, s_1, \ldots, s_{i-1}, \mathsf{on}, s_{i+1}, \ldots, s_n \,\rangle \twoheadrightarrow \\ &\quad \mathsf{if}\ b\ \mathsf{then}\ f\,(t_2)\,@\,\langle\, 0, s'_1, \ldots, s'_{i-1}, \mathsf{on}, s'_{i+1}, \ldots, s'_n \,\rangle \\ &\qquad\quad \mathsf{else}\ f\,(t_1)\,@\,\langle\, 1, s_1, \ldots, s_{i-1}, \mathsf{off}, s_{i+1}, \ldots, s_n \,\rangle \end{aligned}$$

where $s'_1, \ldots, s'_n$ have to be defined suitably. A reasonable definition would be to set all s'_i to on, since there is no need to forbid the use of rules if one rule has been successfully applied. Switching rule i off in the else case is the "de-activation" that was mentioned above.

These requirements can only be implemented if the f-rules satisfy some **regularity conditions**:

(1) f-terms on the left-hand sides of rules must not be nested, because otherwise, it becomes difficult to retain the correspondence between the rule switches and the rules. The idea behind contexts is

- to add a suitable context to an f-term only on outermost level,
- to normalize the term together with its context, and
- finally to remove the context which then has become useless (see Section 3.3.1)

(2) In order to be able to assign appropriate contexts to the f-terms on the right-hand sides (condition (3) above), one must require that the final outcome of these is determined by f-terms. This means that the bodies of the right-hand sides (after removing **if** conditions and **let** clauses) must consist only of f-terms.

Example: The rules

$f(t_1) \longrightarrow \textsf{if } b \textsf{ then } f(t_2) \textsf{ else } f(t_3)$ and

$f(t_1) \longrightarrow \textsf{let } x = e \textsf{ in } f(t_2)$

satisfy the condition, whereas

$f(t_1) \longrightarrow \textsf{if } b \textsf{ then } f(t_2) \textsf{ else } g(t_3)$ and

$f(t_1) \longrightarrow \textsf{let } x = f(t_2) \textsf{ in } e$

do not (in the last example, only the body of **let** is important, not the term on the right-hand side of the **let** clause).

3.3.1 Elimination of Contexts

Since contexts are just a technical means to control the rewriting of f-terms, but not an original part of the description of the domain modelled by these terms, an operator must be defined that removes contexts that are not needed anymore. This operator will be called ***eval*** as it is responsible for guaranteeing that the first component of its argument (an f-term) is rewritten ("evaluated") according to the restrictions given by the second component (a context). If *CtPair* is the sort representing pairs of elements of type S (as constructed by f) and contexts, then $eval : CtPair \rightarrow S$.

In order to determine what contexts can be considered as "not needed anymore", the three possible types of contexts have to be examined:

(1) Terms in 0-contexts are completely evaluated; so the following type of rule is needed:

$$eval(\, s \,@\, \langle\, 0, \ldots \rangle\,) \longrightarrow s \tag{3.9}$$

where s is a variable of type S.

(2) Terms in 1-contexts still have to be rewritten (with rules for f-terms) exactly once. So these contexts must not disappear, and hence there must not be a rule for this case.

(3) Terms in *-contexts may be rewritten arbitrarily often. So the context alone does not provide the desired information, and the controlled term itself is used to determine whether rewriting is finished.

A special feature of the f-terms that will occur in Chapter 7 is that there exists a subclass of these terms that represent "terminal" elements that should be irreducible. Let t be a syntactic representation of such elements; then the following type of rule is needed:

$$eval(\, t \,@\, \langle\, *, \ldots \rangle\,) \longrightarrow t \tag{3.10}$$

In order to be able to implement this kind of rule, it must be decidable from the syntactic form of an f-term whether it denotes a terminal element or not. In the examples, this will be achieved by requiring that a general term pattern be given for the terms t denoting terminal elements.

Usually, the introduction of contexts restricts the rewriting relation of a given system $\mathcal{R}$. Certain unwanted rewriting sequences are prevented, but no additional sequences become possible. This is because every rewriting step in the system with contexts corresponds to a step in the original system; the terms themselves are not changed.

Chapter 4

The Basic Rewriting System $\mathcal{B}$

4.1 The Underlying Theory

The aim of this work is to provide a way to use semantics definitions within a rewriting-based proof tool. So let L be a language for which a semantics definition is given. It does not suffice to implement the rules defining the semantics of L in form of rewrite rules. Mathematical semantics definitions also rely heavily on a proper understanding of the standard operators that are used to define the semantics functions. The intended meaning of these operators (their standard interpretation) is given in form of a special Σ-algebra:

Definition 4.1 (standard interpretation $\mathcal{E}$, $=_E$)

The **standard interpretation** of the operators in Σ is defined by a Σ-algebra $\boldsymbol{\mathcal{E}}$. The equality induced by this algebra is denoted by $=_{\boldsymbol{E}}$, i. e.

$$\forall t_1, t_2 \in T(\Sigma, V) : t_1 =_E t_2 \Leftrightarrow_{df} \mathcal{E} \models \ulcorner t_1 = t_2 \urcorner$$

For a faithful implementation of the semantics definitions, $\mathcal{E}$ has to be modelled as well. This means that a rewriting system has to be supplied whose equational theory is a suitable approximation of $Th_{EQ}(\mathcal{E})$. In the following, this "basic" rewriting system will be denoted by $\boldsymbol{\mathcal{B}}$ and its equational theory by $=_{\boldsymbol{\mathcal{B}}}$, i. e. (cf. Theorem 3.37)

$$\forall t_1, t_2 \in T(\Sigma, V) : t_1 =_{\mathcal{B}} t_2 \Leftrightarrow_{df} Th_{EQ}(\mathcal{B}) \models \ulcorner t_1 = t_2 \urcorner \Leftrightarrow t_1 \overset{*}{\longleftrightarrow}_{\mathcal{B}} t_2$$

In principle, $=_{\mathcal{B}}$ should equal $=_E$, but in practice this is not always necessary (see Section 4.3 below).

One should note that the signature Σ is not necessarily the same for all language definitions. Which operators are needed to model a semantics definition depends heavily on the actual language and its meta-level description. Of course, this implies that $\mathcal{E}$ and $=_E$ are not fixed, either, and therefore the same holds for the basic rewriting system $\mathcal{B}$. Only the general structure of $\mathcal{B}$ can be described, but it is not possible to say in general exactly which rules it is made up of.

Essentially, $\mathcal{B}$ consists of two parts. The first one contains rules that correspond to explicit laws in the definition of the language L that is being considered. In the examples, these laws are the ones defining the static semantics of L. As such definitions are usually of a rather simple structure, they can be transformed into rewrite systems quite easily. Examples for static semantics rules and their transformation can be found in Section 10.2.2.

The second, much larger group of rules contains the definition of the "data types" (syntactic and semantic domains) that are used. Typically, these domains are defined without explicit mentioning of the laws that are assumed to hold, and so one has to find a way to generate rewrite rules from the domain definitions. How this can be done is the subject of the next section.

Essentially, data type rules fall into two classes. Rules in the first class describe those operators that concern the structure of data, such as constructors and selectors. These rules can be derived from the definition of the data structures very systematically (this process can even be automated). Rules in the second class concern the basic data types, however, and since these types have rather individual properties, the rules for these types have to be defined individually as well.

One type that is always included in the type $\mathbb{B}$ of truth values, and among the most important operators for this type are the quantifiers. Section 4.2.5.1 shows how the problems with representing bound variables in quantified formulas can to some extent be solved with the technique that has already been used for the modelling of **let** terms, viz. the $\lambda\sigma$-calculus.

4.2 Data Type Rules

Most of the rules that are contained in the basic rewriting system $\mathcal{B}$ deal with the mathematical domains that are used to describe the application area and the problem to be solved. The operators occurring in these rules are constructors, selectors and recognizers for the domains. In this section, it will be explained how domains are defined and what kinds of definitions are always included in the rules modelling them. Besides these "standard rules" there may be more rules that result from the definition of special mathematical structures on the defined domains, e. g. cpo definitions occurring in denotational definitions (see Section 6.1). How such structures are taken care of will be demonstrated later (see Section 8.2).

As in VDM (cf. Jones [76]), the domains are defined by a set of recursive equations.[1] Not the full general VDM-SL form (cf. Dawes [28]), but only a restricted version will be used. An abstract definition of the form of domain equations is the following:

Definition 4.2 (domain equations)

Domain equations are defined by the following grammar:

$$\begin{array}{rcl}
\textit{dom-equation} & ::= & \textit{idf} \text{ `}=\text{' } \textit{dom-expr} \\
\textit{dom-expr} & ::= & \textit{union-dom} \mid \textit{function-dom} \mid \textit{map-dom} \\
\textit{union-dom} & ::= & \textit{product-dom}_1 \text{ `}|\text{' } \ldots \text{ `}|\text{' } \textit{product-dom}_n \ (n \geq 1) \\
\textit{product-dom} & ::= & \textit{elem-dom}_1 \text{ `}\times\text{' } \ldots \text{ `}\times\text{' } \textit{elem-dom}_n \ (n \geq 1) \\
\textit{elem-dom} & ::= & \textit{dom-name} \mid \textit{basic-dom} \mid \textit{token-dom} \\
\textit{dom-name} & ::= & \textit{idf}
\end{array}$$

[1]The existence of a solution for such a system of equations is assumed to be guaranteed by an suitable underlying mathematical formalism (cf. Gunter and Scott [61]).

$$\begin{aligned}
\textit{basic-dom} &::= \mathsf{Bool} \mid \mathsf{Nat} \mid \ldots \\
\textit{token-dom} &::= \textit{token} \\
\textit{function-dom} &::= \textit{product-dom}\ `\!\rightarrow\!'\ \textit{elem-dom} \\
\textit{map-dom} &::= \textit{product-dom}\ `\!\xrightarrow{m}\!'\ \textit{elem-dom}
\end{aligned}$$

The meaning of the different sorts of domains possible for *dom-expr* is the following:

- *idf* stands for an identifier denoting a domain that is being defined with a system of domain equations. It will be assumed that for each domain identifier there is at most one domain equation where it occurs on the left-hand side. A domain identifier only occurring on right-hand sides of the domain equations of a system denotes a domain that is left unspecified in this system. In VDM, such an identifier would be declared **not yet defined.**
- The "basic domains" are considered as predefined. At least the truth values (Bool) and natural numbers (Nat) are assumed to be among these.
- "Tokens" are special symbols denoting a one-element set consisting of just that symbol. Tokens will be written in the special representation token.

In the following, assume that $D_1, \ldots, D_n$ and D' all are denotations for some domains constructed in the form defined above. Instead of writing "the domain denoted by D" it will simply be written "the domain D".

- If $D = D_1 \mid \ldots \mid D_n$, then D denotes the disjoint union of the domains $D_1, \ldots, D_n$.
- If $D = D_1 \times \ldots \times D_n$, then D denotes the domain of trees whose root is labelled D and whose subtrees are unnamed and elements of the domains $D_1, \ldots, D_n$. In VDM-SL (cf. Dawes [28]), this is written as $D :: D_1, \ldots, D_n$.
- If $D = D_1 \times \ldots \times D_n \rightarrow D'$, then D denotes the domain of partial functions from the product of $D_1, \ldots, D_n$ into D'. Such functions may be infinite.
- If $D = D_1 \times \ldots \times D_n \xrightarrow{m} D'$, then D denotes the domain of finite maps from the product of $D_1, \ldots, D_n$ into D'.

This syntax is rather restrictive in not allowing very complex domain constructions. But the effect of an equation with a complex expression can always be achieved by a set of simpler equations: Instead of

$$T_1 = T_2 \times (T_3 \rightarrow T_4) \times (T_5 \mid T_6)$$

the following mathematically equivalent system can be used:

$$\begin{aligned}
T_1 &= T_2 \times T_7 \times T_8 \\
T_7 &= T_3 \rightarrow T_4 \\
T_8 &= T_5 \mid T_6.
\end{aligned}$$

There is one complex construction, however, that is allowed by Def. 4.2: the alternatives in a union domain may be domain products and not just elementary. The motivation for allowing this kind of complexity in *domain expressions* was the wish to have less complexity in frequently occurring *terms*. Domain equations of the form

$$T = A \times B \mid C \times D \times E \mid F$$

often result from abstract syntax definitions. In reasoning about semantics, however, abstract syntax terms appear quite frequently.

Consider the mathematically equivalent form of the domain equations above:

$$\begin{aligned} T &= T_1 \mid T_2 \mid F \\ T_1 &= A \times B \\ T_2 &= C \times D \times E \end{aligned}$$

With this definition, the syntactic terms would become larger (and hence, less readable) than with the original equation. The reason is that for each new domain that has a name, injection and projection operators are introduced that must be used to construct terms (see Section 4.2.1 for details), and therefore extra levels of nodes would have to be introduced in the abstract syntax trees.

The following provides the definitions that are needed for modelling domain equations by term rewriting systems. *Italic* font will be used to write mathematical domains, and sans serif font to write the corresponding representations.

No claim will be made that a domain is completely characterized by the rules presented in this section. A provably complete axiomatization needs a much larger number of rules, as shown by Nickl [98]. In particular, the modelling of function domains requires many rules that will not be introduced here. For the consequences of this incompleteness, see Section 4.3.

4.2.1 Union Domains

Consider a domain equation

$$T = T_{11} \times \ldots \times T_{1n_1} \mid \ldots \mid T_{m1} \times \ldots \times T_{mn_m}$$

where $m \geq 1, n_i \geq 1$ for $i \in [m]$ and each of the T_{ij} is elementary, i. e. the name of a basic or defined domain or a token. Assume that for each of the identifiers T_{ij} in this equation there has been defined a sort T_{ij}, and that there are variables v_{ij} for each such sort T_{ij}.

Four kinds of operators are needed to model a union domain:

- **constructors** for T that generate an element of one of the product subdomains and inject it into T;
- **projection operators** mapping elements of T into the appropriate subdomain;
- **recognizers** signalling whether or not an element of T belongs to a given subdomain;
- **selectors** accessing components of a product subdomain.

Product domains are also considered as unions with just one subdomain (that is a product).

The form of the operators depends on the structure of the subdomains. There are four different possible cases for each $i \in [m]$:

(1) $n_i = 1$ and T_{i1} is a token.

In this case T_{i1} itself is taken as an element of T and the one-point subdomain consisting only of T_{i1} is not considered. So the constructor is the constant $\mathsf{T}_{i1} : \rightarrow \mathsf{T}$. The recognizer for this case is simply equality to T_{i1}; $t \in T_{i1}$ is modelled as $\mathsf{t} = \mathsf{T}_{i1}$.

Since the subdomain is not considered, there is no need for projection and selection operators in this case.

(2) $n_i = 1$ and T_{i1} is not a token.

In this case, T_{i1} is just (the name of) a plain subdomain of T. Selectors are not needed since T_{i1} is syntactically not a real product.[2] The operators introduced are:

$\mathsf{mk\text{-}T} : \mathsf{T}_{i1} \rightarrow \mathsf{T}$	constructor
$\mathsf{is\text{-}T}_{i1} : \mathsf{T} \rightarrow \mathsf{Bool}$	recognizer
$\mathsf{to\text{-}T}_{i1} : \mathsf{T} \rightarrow \mathsf{T}_{i1}$	projector

(3) $n_i > 1$ and T_{i1} is a token.

Here T_{i1} is considered not as a real component of the product, but rather as a label for elements of this subdomain. Therefore T_{i1} is included in the names of the operators and omitted from their arguments. Since the subdomain does not have a simple name, no projection operator is introduced; the other operators are:

$\mathsf{mk\text{-}T}_{i1}\mathsf{\text{-}T} : \mathsf{T}_{i2},\ldots,\mathsf{T}_{in_i} \rightarrow \mathsf{T}$	constructor
$\mathsf{is\text{-}T}_{i1}\mathsf{\text{-}T} : \mathsf{T} \rightarrow \mathsf{Bool}$	recognizer
$\mathsf{s\text{-}}j : \mathsf{T} \rightarrow \mathsf{T}_{i,j+1}$	selector for component $j \in \{2, \ldots, n_i - 1\}$

(4) $n_i > 1$ and T_{i1} is not a token.

Here the i–th component domain is a simple product. As in the previous case, no projector is introduced. Since this subdomain does not possess a name, a recognizer is not introduced either, but only

$\mathsf{mk\text{-}T} : \mathsf{T}_{i1},\ldots,\mathsf{T}_{in_i} \rightarrow \mathsf{T}$	constructor
$\mathsf{s\text{-}}j : \mathsf{T} \rightarrow \mathsf{T}_j$	selector for component $j \in [n_i]$

The following set of rules for the sort `T` representing the domain T (provided the operators used are actually declared) is defined:

- selector/constructor rules ($j \leq n_i, n_i > 1$):

  $\mathsf{s\text{-}}j\ (\mathsf{mk\text{-}T}\ (\mathsf{v}_{i1},\ldots,\mathsf{v}_{in_i})) \longrightarrow \mathsf{v}_{ij}$

- projection/recognizer rules ($n_i = 1$):

  $\mathsf{is\text{-}T}_{i1}\ (\mathsf{mk\text{-}T}\ (\mathsf{v}_{i1}))$
  $\mathsf{to\text{-}T}_{i1}\ (\mathsf{mk\text{-}T}\ (\mathsf{v}_{i1})) \longrightarrow \mathsf{v}_{i1}$

- disjointness rules ($i \neq j$):

  $\mathsf{mk\text{-}T}\ (\mathsf{v}_{i1},\ldots,\mathsf{v}_{in_i}) = \mathsf{mk\text{-}T}\ (\mathsf{v}_{j1},\ldots,\mathsf{v}_{jn_j}) \longrightarrow \mathsf{false}$
  $\mathsf{is\text{-}T}_{i1}\ (\mathsf{mk\text{-}T}\ (\mathsf{v}_{j1},\ldots,\mathsf{v}_{jn_j})) \longrightarrow \mathsf{false}$ $n_i = 1, i \neq j, T_{i1}$ not a token

  For token components, the operators names are replaced accordingly.

[2] Note, however, that there may be a domain equation defining T_{i1} as a product. In order to access the components of elements of T_{i1}, one first has to project into T_{i1} and then to select the appropriate component domain.

In order to enhance readability, instead of the prefix operators s-j sometimes the postfix operators $\downarrow j$ will be used.

4.2.2 Map and Function Domains

Consider a domain equation

$$T = T_1 \times \ldots \times T_m \xrightarrow{\text{m}} T_{m+1}$$

where $m \geq 1$ and each of the T_i is a domain identifier with typical variable v_i. Again assume the definition of sorts T_i with typical variables v_i as in the previous section.

Three operators are introduced for such a function domain: the **empty map** of sort T, the **extension (modification)** of a given map with an additional argument list and result, and the **application** of a map to an argument tuple. So there are

- empty: $\rightarrow$ T (empty map)
- ext: T, $\mathsf{T}_1,\ldots,\mathsf{T}_{m+1} \rightarrow$ T (extension)
- app: T, $\mathsf{T}_1,\ldots,\mathsf{T}_m \rightarrow \mathsf{T}_{m+1}$ (application)

  or, if $m = 1$, with an infix operator:

  . : T, $\mathsf{T}_1 \rightarrow \mathsf{T}_2$

The application operators provide a way to circumvent the restriction to first-order logic. One only has to introduce application for each map or function domain; if this has been done, variables for functions (more exactly, for sorts representing functions) may be used on argument positions that are reserved for operators. They must still not occur, of course, on ordinary operator positions.

There is only one general rule that describes the effect of applying an extended function:

```
app (ext (v,$v_1$,...,$v_{m+1}$), $v_1$',...,$v_m$') $\longrightarrow$
  if $v_1$ = $v_1$' $\wedge$ ... $\wedge$ $v_m$ = $v_m$'
    then $v_{m+1}$ else app (v, $v_1$',...,$v_m$')
```

The effect of applying the empty map is mathematically not defined. Therefore no rule is asserted for this case. Since there are no other rules for application, the term `app(empty,...)` itself denotes an error; there is no need for an extra error element. This situation should not arise anyhow; it indicates an incorrect mathematical modelling of the problem.

Modelling a function domain

$$T = T_1 \times \ldots \times T_m \rightarrow T_{m+1}$$

is similar to modelling a map domain. Since functions are total, however, there is no "empty" function. Furthermore, maps are finitely generated from empty and ext, whereas functions are not necessarily finite at all. Function "extension" means setting the result value for a particular argument ("function overwrite"). So the rules for functions are like the rules for maps described above.

4.2.3 Additional VDM Domain Constructors

In full VDM-SL, there are some additional domain constructors that were not needed for the purposes of this work. In this section, a brief description shall be given how these constructors could be represented using the representations of the last section.

Sets If $D = D_1\text{–}\mathsf{set}$, then D denotes the domain of all finite subsets of D.

Such a domain can also be modelled by a map domain $D' = D \xrightarrow{m} \underline{\mathsf{in}}$, where $\underline{\mathsf{in}}$ is some arbitrary token: $d = \{d_1, \ldots, d_N\} \in D$ is represented as the map $d' \in D'$ with $\operatorname{dom} d' = \{d_1, \ldots, d_N\}$.

Sequences If $D = D_1^*$, then D denotes the domain of finite sequences of elements of D_1.

Such a domain can be modelled by a map $D' = \mathsf{Nat} \xrightarrow{m} D_1$ and the additional law

$$\forall d \in D' \, \forall n \in \mathsf{Nat} : n > 1 \;\wedge\; n \in \operatorname{dom} d \;\Rightarrow\; (n-1) \in \operatorname{dom} d$$

which ensures that the domain of $d' \in D'$ is an interval $[n] \subseteq \mathsf{Nat}$.

Named trees If $D :: Id_1 : D_1, \ldots, Id_n : D_n$, then D denotes the trees whose root is labelled D and whose subtrees are also labelled (with $D_1, \ldots, D_n$) and contained in $D_1, \ldots, D_n$.

Named trees of this kind can be simulated by unnamed ones; the difference is mainly the existence of selector functions $s_{Id_1}, \ldots, s_{Id_n}$, but these can be expressed by the selectors $s_1, \ldots, s_n$ selecting the i-th subtree.

Tuples $D = D_1 \times \ldots \times D_n$ in VDM-SL means that D denotes the set of all tuples from the product of $D_1, \ldots, D_n$. This is different from $D :: D_1, \ldots, D_n$: If

$$D = D_1 \times \ldots \times D_n, \quad D' = D_1 \times \ldots \times D_n \qquad (*)$$

then D and D' denote the same domains, whereas in

$$D :: D_1, \ldots, D_n, \quad D' :: D_1, \ldots, D_n \qquad (**)$$

they denote different domains: the structure of the trees is the same, but the trees themselves are labelled differently.

Tuples can be simulated with trees provided the law

$$\forall d_1 \in D_1, \ldots, d_n \in D_n : \mathsf{mk}\text{–}D(d_1, \ldots, d_n) = \mathsf{mk}\text{–}D'(d_1, \ldots, d_n)$$

is added whenever the situation $(*)$ is desired.

4.2.4 Type Rules

The rewrite rules generated from SOS deduction rules (cf. Chapter 7) assume the possibility to deduce the type of given terms that are formed using the VDM domain constructors. Therefore suitable operators and rules have to be supplied.

First a sort Type is needed for all the possible types of terms, and for each domain T an operator

$$\mathsf{type} : T \rightarrow \mathsf{Type} \tag{4.1}$$

and an element of Type that corresponds to T:

$$\_T \; : \; \rightarrow \mathsf{Type} \tag{4.2}$$

The first set of rules describes that each element of T has indeed type $\_T$:

$$\mathsf{type}(\mathsf{v}) \rightarrow \_T \tag{4.3}$$

where v is the typical variable of sort T. The second set of rules guarantees that every equality about types can be decided by rewriting. For each two distinct domains T_1, T_2, there is the rule:

$$\_T_1 = \_T_2 \rightarrow \mathsf{false} \tag{4.4}$$

Example 4.3 (deciding type conditions)

Assume that there are sorts $\mathsf{S}, \mathsf{T} \in \mathcal{S}$, that $\mathsf{t} \in T(\Sigma, V)_{\mathsf{S}}$, and that the condition

$$\mathsf{type}(\mathsf{t}) = \_\mathsf{T} \tag{4.5}$$

has to be decided. By construction (4.2), there are type constants $\_\mathsf{S}$ and $\_\mathsf{T}$ in Σ. If $\mathsf{T} = \mathsf{S}$, applying the rule (4.3) (with T for T) to (4.5) results in $\_\mathsf{T} = \_\mathsf{T}$, which is subsequently rewritten to true by the rules for the basic domain Bool (see the next section). If, on the other hand, $\mathsf{T} \neq \mathsf{S}$, then applying rule (4.3) yields $\_\mathsf{T} = \_\mathsf{S}$ which is immediately rewritten to false by the rules (4.4).

□

Finally, an operator $\#\mathsf{comp} : S \rightarrow \mathsf{Nat}$ is needed for each sort S that can be used to calculate the number of subcomponents of a tuple term. It is defined by the following set of rules:

$$\#\mathsf{comp}(\mathsf{mk\text{-}T}(\mathsf{v}_1, \ldots, \mathsf{v}_n)) \rightarrow n \tag{4.6}$$

for all sorts T and operators $\mathsf{mk\text{-}T} : \mathsf{T}_1, \ldots, \mathsf{T}_n \rightarrow \mathsf{T}$ where $\mathsf{v}_i \in V_{\mathsf{T}_i}$ for $i \in [n]$.

4.2.5 Rules for the Basic Domains

The most important basic domain is $\mathbb{B}$, containing the truth values and represented by the sort Bool (see Def. 3.13). It is particularly important because every term denoting a condition is Bool-typed, and strong assumptions about rewriting of conditions will be needed (see below in Section 4.3). Moreover, a sort Nat for representations of natural numbers will be needed to implement the rule (4.6).

The exact form and amount of rules needed depends on the actual implementation. But at least the following has to be assumed:

- $\mathcal{B}$ includes a complete axiomatization of propositional logic with the usual operators.
- For each sort $S \in \mathcal{S}$, there has to be an operator

  $$\mathsf{if}\ \_\ \mathsf{then}\ \_\ \mathsf{else}\ \_ : \mathsf{Bool}, S, S \to S$$

  together with the usual rules for the conditional (let $t_1, t_2 \in T(\Sigma, V)_S$ for some $S \in \mathcal{S}$):

  $$\mathsf{if\ true\ then}\ t_1\ \mathsf{else}\ t_2 \longrightarrow t_1$$
  $$\mathsf{if\ false\ then}\ t_1\ \mathsf{else}\ t_2 \longrightarrow t_2$$

The structure and amount of rules for the natural numbers and the other basic domains also depends on the actual problem. Here it will be assumed that the operators needed are axiomatized in sufficient completeness; what this means will become clear in Section 4.3.

4.2.5.1 Quantifiers in Rewrite Rules

Up to now, all the variables in rewrite rules were implicitly universally quantified, and no explicit quantification was used. The laws defining cpo's, however, also feature quantifiers on inner positions that can only be moved outward by turning them into existential quantifiers. Take e. g. the law that in a cpo $\langle M, \sqsubseteq \rangle$ the lub z of a chain X is smaller than all upper bounds of X (see Def. 6.1):

$$\forall z' \in M : (\forall x \in X : x \sqsubseteq z') \Rightarrow z \sqsubseteq z' \tag{4.7}$$

The prenex normal form of this formula is:

$$\forall z' \in M\ \exists x \in X : \neg\, x \sqsubseteq z' \ \vee\ z \sqsubseteq z' \tag{4.8}$$

In this section, we will see how universal quantifiers can be introduced as term constructors such that (4.7) becomes a valid rewrite rule. The method cannot deal with fully general formulas, however, due to unsolvable problems with scoping. Consider e. g. the Boolean formula

$$\forall x \in X : p(x) \tag{4.9}$$

where the operator p is defined by

$$p(y) \Leftrightarrow_{df} \forall x \in X : q(x, y) \quad . \tag{4.10}$$

If (4.10) is turned into a rewrite rule (directed from left to right, since definitions should be unfolded) and applied to (4.9) in term rewriting style, the result is

$$\forall x \in X : (\, \forall x \in X : q(x,x)\,) \quad . \tag{4.11}$$

Of course, this result is semantically wrong because one of the x's should have been renamed, resulting e. g. in

$$\forall x \in X : (\, \forall z \in X : q(z,x)\,) \quad . \tag{4.12}$$

But this renaming of variables depends on the context in which they occur, and since term rewriting is a context-free process, it is not possible to properly include renaming.

A similar problem has already occurred in Section 3.2: In λ-calculus, there is also the need to cope with scopes of variables and renaming. The solution of that section was to abolish variable names completely and to use the $\lambda\sigma$-calculus. This does not help here, however, with the problem about quantified variables. The difference between the two situations is that in the $\lambda\sigma$-calculus, β-reduction is stated as a rewrite rule, which allows to control term manipulations on object (term) level by defining the proper substitutions. In contrast to that, the steps from (4.9) to (4.11) are performed with a rule (viz. the definition of a rewrite step) only specified on meta level. Only in very special cases there is the possibility to influence this meta level rule by term level objects directly (see Chapter 7).

This means that for reasoning about quantified formulas, one either has to resort to deduction methods that go beyond term rewriting, or to restrict oneself to cases where conflicts about variable names cannot occur. So assume that the latter is the case, i. e. that in the actual application formulas that contain nested quantifiers ranging over the same sort do never occur.[3]

Universal quantification Basically, there are two situations in which one has to deal with a universally quantified formula $\forall x \in S : p$ during a proof: either in an attempt to prove it or in an attempt to put it to use by specializing x to some suitable term t of the same sort. First consider specializing. Essentially, it means that t must be substituted for x throughout p and that the quantifier must be removed, following the rule

$$\frac{\forall x \in S : p}{p[t/x]} \tag{4.13}$$

Usually, this rule has the side condition that t be free for x in p, but in the case where variable conflicts do not occur this may be dropped.

If this rule is to be modelled by a rewrite rule corresponding to

$$(\, \forall x \in S : p\,) \;\Rightarrow\; p[t/x] \tag{4.14}$$

a way to perform the substitution by term rewriting has to be implemented. In Section 3.2, such an implementation has already been described, viz. the $\lambda\sigma$-calculus. It can be employed here as

[3]Nested quantifiers ranging over different sorts are no problem, since the set of variables is disjoint.

well, solving also the problem of representing the bound variable x: it is replaced by the first de Bruijn index $\_\mathsf{x}$ of sort S. So universal quantification can be modelled with the operator

$$\mathsf{forall} : \mathsf{Type}, \mathsf{Bool} \to \mathsf{Bool}$$

and specialization can be implemented with the rewrite rule

$$\mathsf{forall}(\_\mathsf{S}, \mathsf{p}) \;\Rightarrow\; \mathsf{p}[\mathsf{t} \cdot id] \tag{4.15}$$

where $\_\mathsf{S}$ is the constant of sort Type representing the sort S, $\_[\_]$ is the substitution application, id the identity substitution defined in Section 3.2.2, and t is a variable of sort S.

In order to prove the universal formula, the usual direct way is to prove the formula p under the assumption that x is some arbitrary, but fixed element of S ("let $x \in S \ldots$"). Since the de Bruijn index $\_\mathsf{x}$ is a constant of sort S without any defining rewrite rules (except for those describing its interaction with substitutions), this proof method can be modelled by the simple rewrite rule

$$\mathsf{forall}(\mathsf{tp}, \mathsf{true}) \tag{4.16}$$

where $\mathsf{tp} \in V_{\mathsf{Type}}$. If a base formula p can be rewritten to true (i. e. proved by rewriting), then the generalized formula $\mathsf{forall}\ (\mathsf{tp},\ \mathsf{p})$ can also be proved using this rule:

$$\begin{array}{lll} \mathsf{forall}(\mathsf{tp}, \mathsf{p}) & \xrightarrow{*} \mathsf{forall}(\mathsf{tp}, \mathsf{true}) & [\text{ Proof for } \mathsf{p}\ ] \\ & \longrightarrow \mathsf{true} & [\text{ rule } (4.16)\ ] \end{array}$$

Existential quantification can be modelled similarly by an operator

$$\mathsf{exists} : \mathsf{Type}, \mathsf{Bool} \to \mathsf{Bool}$$

with the rule

$$\mathsf{p}[\mathsf{t} \cdot id] \;\Rightarrow\; \mathsf{exists}(\_\mathsf{S}, \mathsf{p}) \tag{4.17}$$

to be used for proving existentially quantified formulas. The usual way to make use of such a formula that is contained in a hypothesis is to drop the quantifier and consider its variable as a constant. This can be modelled by the rule

$$\mathsf{exists}(\_\mathsf{S}, \mathsf{p}) \;\Rightarrow\; \mathsf{p} \tag{4.18}$$

since the de Bruijn index corresponding to the quantified variable is already defined as a constant.

An example proof Because of the fundamental restrictions of this method, it has not been used in large proofs; instead, other deduction techniques provided by the proof tool used (the Larch Prover) have been applied. In Appendix C, however, a small example from cpo theory is presented that was treated using both methods.

4.3 Properties of $\mathcal{B}$

Since the main interest is to show how semantics definitions can be modelled by rewrite systems, it will be assumed that $\mathcal{B}$ behaves "reasonably well": $\mathcal{B}$ shall not obstruct any rewriting processes that are necessary to simulate semantics definitions. Since the simulating rules will contain conditions on objects of the underlying data types, this implies that there must be sufficiently many rules such that any such condition is decidable by rewriting, and of course all these decisions must be correct with respect to the standard interpretation of the operators in Σ as described in $\mathcal{E}$. So the following is required:

Requirement 4.4 (correctness of $\mathcal{B}$)

For all $t \in T(\Sigma, V)_{\mathsf{Bool}}$: If $t \xrightarrow{*}_{\mathcal{B}} \mathsf{true}$, then $t =_E \mathsf{true}$.

Requirement 4.5 (completeness of $\mathcal{B}$)

For all $t \in T(\Sigma, V)_{\mathsf{Bool}}$: If $t =_E \mathsf{true}$, then any rewriting sequence in $\mathcal{B}$ starting from t eventually ends in the term true.

where Σ is the signature of the basic system, V the corresponding set of variables and Bool the sort of the truth value representations true and false. Note that $t =_E \mathsf{true} \iff t_{\mathcal{E}} = \mathsf{tt}$, where $t_{\mathcal{E}}$ is the standard interpretation of the term t in the Σ-algebra $\mathcal{E}$, see Def. 4.1 and Lemma 3.30.

The strong form of completeness guarantees that t is rewritten to true whichever possible rewriting sequence for t is chosen. If only the existence of one sequence $t \xrightarrow{*}_{\mathcal{B}} \mathsf{true}$ were required, there also might be other sequences that are not terminating and hence do not yield a result at all. (Correctness precludes the existence of sequences that end with false.)

In the light of Corollary 3.38, one might suspect that TRS-completeness (i. e. completeness in the term rewriting sense) together with correctness of each of the rules in $\mathcal{B}$ might be sufficient to guarantee the above requirements. But these two assumptions do not guarantee that the rules really suffice to decide every possible term; in particular, the empty system is TRS-complete and it obviously only contains correct rules. So one must additionally assume that there are "enough" rewriting sequences:

Requirement 4.6 (assumptions about $\mathcal{B}$)

(A) $\mathcal{B}$ shall be **correct**, i. e.

$$\forall t_1, t_2 \in T(\Sigma, V) : t_1 \rightarrow_{\mathcal{B}} t_2 \;\Rightarrow\; t_1 =_E t_2$$

(B) $\mathcal{B}$ shall **decide conditions**, i. e.

$$\forall t \in T(\Sigma, V)_{\mathsf{Bool}} : t \xrightarrow{*}_{\mathcal{B}} \mathsf{true} \text{ or } t \xrightarrow{*}_{\mathcal{B}} \mathsf{false}$$

(C) $\mathcal{B}$ shall be **TRS-complete**.

Lemma 4.7

The requirements 4.6 imply those in 4.4 and 4.5.

Proof

Correctness: Follows easily by induction from (A) and from the fact that $\mathsf{true}_{\mathcal{E}} = \mathsf{tt}$.

Completeness: Let $t \in T(\Sigma, V)_{\mathsf{Bool}}$ with $t_{\mathcal{E}} = \mathsf{tt}$. Then one can infer from (B) that $t \xrightarrow{*}_{\mathcal{B}} \mathsf{true}$ ($t \xrightarrow{*}_{\mathcal{B}} \mathsf{false}$ is not possible because $\mathcal{B}$ is correct and $\mathsf{false}_{\mathcal{E}} = \mathsf{ff} \neq \mathsf{tt}$).

Since $\mathcal{B}$ is TRS-complete, it is terminating; so there are no infinite rewriting sequences starting in t. Let $t_1 \in T(\Sigma, V)$ with $t \xrightarrow{*}_{\mathcal{B}} t_1$ and $t_1 \not\rightarrow_{\mathcal{B}}$. Then

$$\mathsf{true} \; {}_{\mathcal{B}}\!\xleftarrow{*} t \xrightarrow{*}_{\mathcal{B}} t_1,$$

and by confluence of $\mathcal{B}$ (from (C)), there exists a $t' \in T(\Sigma, V)$ such that

$$\mathsf{true} \xrightarrow{*}_{\mathcal{B}} t' \; {}_{\mathcal{B}}\!\xleftarrow{*} t_1.$$

Since both true and t_1 are normal forms, it follows that $t_1 \equiv \mathsf{true}$.

□

(A) is the most basic requirement; an incorrect rewrite system is simply useless. So the rules of $\mathcal{B}$ must be written with great care; no incorrect rule can be tolerated.

(B) is the condition about "enough" rewriting sequences. If (B) is violated, then there exists a Boolean term t not reducible to true or false; if (C) holds we may assume that t is a normal form. This is obviously an obstacle for any attempt to prove that t holds; but since the rewriting proof simply gets stuck at t, this fault of $\mathcal{B}$ can easily be detected and the rules that are missing to normalize t can be added.

If $\mathcal{B}$ is not terminating, then proofs (reductions of Bool-sorted terms) can fail by not reaching a normal form, and if $\mathcal{B}$ is not confluent, they can fail by reaching a normal form different from true and false.[4] Especially the combination of both behaviours might lead to situations where it is difficult to see whether there is a problem with the rules in $\mathcal{B}$ and where this problem lies. But in practice, this is hardly the case. Non-terminating rewriting can be excluded by considering only rewriting sequences of some maximal length; if this limit is chosen suitably and it is reached during a proof attempt, then this indicates a termination problem that should be detectable by analysing the critical rewriting sequence. And if a Bool-sorted term is reduced to a normal form different from true or false, the situation is similar to the case where requirement (B) was violated: Some rules are missing, and from the "wrong" normal form, it should be possible to deduce what these rules are.

The reason for not strictly insisting on a complete system $\mathcal{B}$ fulfilling (B) and (C) is pragmatic. Such a system would have to consist of a much larger number of rules.[5] Since the performance of a proof tool is strongly related to the number of objects is has to deal with, trying to achieve a complete system would therefore greatly decrease the efficiency of the tool that is used to implement the system $\mathcal{B}$. Moreover, not all of the rules needed to make a system theoretically complete are really used in actual applications.

[4] If $\mathcal{B}$ is correct, a proof that reaches false indicates an invalid conjecture.

[5] This can be seen by comparing the few rules modelling function domains in Section 4.2.2 with the larger number of rules set up for this purpose by Nickl in her algebraic specification of domain constructions that is provably complete [98].

Summing up: violations of (B) and (C) in practice only lead to failing proof attempts, but they do not undermine the logical basis that is consistent as long as (A) holds. Furthermore, the reasons for the failures are deducible from the rewriting sequences that failed to produce the normal forms true or false.

Chapter 5

Structured Operational Semantics

In this chapter, first the basic notions for operational semantics definitions in the style of Plotkin [101] are introduced. Such definitions make use of transition systems whose transition relation is described by a set of deduction rules. The second section of this chapter presents a new general format for these rules and explains on a syntactic level what it means to apply such a deduction rule. Examples from a real SOS definition (taken from Lakhneche [82]) are used for illustration. Finally, the new rule format is compared with some of formats that can be found in the literature.

5.1 Transition Systems

The operational definition of the semantics of a programming language L is accomplished by first defining an abstract machine M and then interpreting the constructs of L by means of the machine instructions of M. In Plotkin's approach, M is given in the form of a transition system:

Definition 5.1 (transition system)

(1) A **transition system** is a triple $(\Gamma, T, \longrightarrow)$, where Γ is the set of **configurations**, $T \subseteq \Gamma$ is the set of **terminal configurations**, and $\longrightarrow \subseteq \Gamma \times \Gamma$ is the **transition relation** satisfying $T \cap \mathrm{dom}(\longrightarrow) = \emptyset$.

(2) A **labelled transition system** is a tuple $(\Gamma, T, A, \longrightarrow)$, where Γ and T are as in (1), A is a set of **labels**, and $\longrightarrow \subseteq \Gamma \times A \times \Gamma$ is the **transition relation** satisfying $\forall\, (\gamma_1, a, \gamma_2) \in \longrightarrow : \gamma_1 \notin T$. For $(\gamma_1, a, \gamma_2) \in \longrightarrow$ we write $\boldsymbol{\gamma_1 \xrightarrow{a} \gamma_2}$.

Labelled systems are introduced because they ease the modelling of interactions with the environment: $\gamma_1 \xrightarrow{a} \gamma_2$ means that the step from configuration γ_1 to γ_2 is taken while performing some action a together with the program's environment. Typical actions of that kind are communication events.

There is, however, no greater expressive power in labelled systems. They can be simulated by unlabelled systems whose configurations have an extra component that contains the sequences of labels:

Lemma 5.2

Let $\mathcal{S} = (\Gamma, T, A, \longrightarrow)$ be a labelled transition system, and $\mathcal{S}' = (\Gamma', T', \longrightarrow')$ be the unlabelled system defined by

- $\Gamma' =_{df} \Gamma \times A^*, T =_{df} T \times A^*$ and
- $\longrightarrow' =_{df} \{((\gamma_1, l), (\gamma_2, l\,a)) \mid \gamma_1, \gamma_2 \in \Gamma, a \in A, l \in A^*, \gamma_1 \xrightarrow{a} \gamma_2\}$.

Then $\mathcal{S}$ and $\mathcal{S}'$ are equivalent in the following sense:

$$\forall\, n \in \mathbb{N}_0\, \forall\, \gamma_0, \ldots, \gamma_n \in \Gamma\, \forall\, a_1, \ldots, a_n \in A :$$
$$\gamma_0 \xrightarrow{a_1} \gamma_1 \xrightarrow{a_2} \ldots \xrightarrow{a_n} \gamma_n \iff (\gamma_0, \varepsilon) \longrightarrow' (\gamma_1, a_1) \longrightarrow' \ldots \longrightarrow' (\gamma_n, a_1 . . . a_n)$$

Proof

By induction on n.

If $\underline{n = 0}$, there is nothing to prove.

If $\underline{n > 0}$, let $\gamma_0, \ldots, \gamma_n \in \Gamma, a_1, \ldots, a_n \in A$.

Case "$\Rightarrow$": Assume $\gamma_0 \xrightarrow{a_1} \gamma_1 \xrightarrow{a_2} \cdots \xrightarrow{a_n} \gamma_n$. Then by induction hypothesis

$$(\gamma_0, \varepsilon) \longrightarrow' (\gamma_1, a_1) \longrightarrow' \cdots \longrightarrow' (\gamma_{n-1}, a_1 \ldots a_{n-1})$$

Since $\gamma_{n-1} \xrightarrow{a_n} \gamma_n$, it follows by definition of $\longrightarrow'$: $(\gamma_{n-1}, a_1 \ldots a_{n-1}) \longrightarrow' (\gamma_n, a_1 \ldots a_{n-1} a_n)$, and this gives the desired result.

Case "$\Leftarrow$": Assume $(\gamma_0, \varepsilon) \longrightarrow' \ldots \longrightarrow' (\gamma_{n-1}, a_1 \ldots a_{n-1}) \longrightarrow' (\gamma_n, a_1 . . . a_n)$. The induction hypothesis yields $\gamma_0 \xrightarrow{a_1} \gamma_1 \xrightarrow{a_2} \cdots \xrightarrow{a_{n-1}} \gamma_{n-1}$. Since $(\gamma_{n-1}, a_1 \ldots, a_{n-1}) \longrightarrow' (\gamma_n, a_1 \ldots a_n)$, it follows from the definition of $\longrightarrow'$ that also $\gamma_{n-1} \xrightarrow{a_n} \gamma_n$.

□

The aim is to show how transition sequences can be simulated by term rewriting sequences. By Lemma 5.2, we now know that it suffices to consider only unlabelled transition systems.

5.2 Deduction Systems

The transition relation of actual transition systems is defined by a deduction system. Since reasoning about transition sequences is required on a syntactic level, in this section deduction systems will be defined with an emphasis on the terms representing configurations.

The starting point in defining a transition system is the definition of the configurations. So assume that the two sets Γ and T are given. Furthermore, assume a signature Σ and a set of variables V such that $T(\Sigma, V)$ contains all the terms needed to express configurations, contexts, and other mathematical objects. Special subsets of $T(\Sigma, V)$ are $\boldsymbol{\Gamma'}$ and $\boldsymbol{T'}$, representing schemata for configurations and terminal configurations, respectively. If no confusion can arise, configurations and their term representations will be identified.

In this work, the transition relations $\longrightarrow$ are defined by means of a special kind of deduction system:

Definition 5.3 (SOS deduction system)

An **SOS deduction system** for Γ and T consists of the term sets $T(\Sigma, V)$, Γ' and T', and inference rule schemata of the following kind:

$$\frac{\vdash \bigwedge_{i=1}^{p} b_i \;\wedge\; \bigwedge_{j=1}^{n} \gamma_j \xrightarrow{L_j} \gamma'_j \;\wedge\; \bigwedge_{k=1}^{q} B_k}{\vdash \overline{\gamma} \longrightarrow \overline{\gamma}'}$$

where

(1) $n, p, q \in \mathbb{N}_0$,

(2) $\overline{\gamma}, \gamma_1, \ldots, \gamma_n \in \Gamma' \setminus T'$ are non-terminal configuration terms,

(3) $\overline{\gamma}', \gamma'_1, \ldots, \gamma'_n \in \Gamma'$ are (possibly terminal) configuration terms, and $\gamma'_1, \ldots, \gamma'_n$ only contain subterms that are variables, constants, or complex terms with a mk- as their outermost operator (see Section 4.2.1).

(4) $b_1, \ldots, b_p, B_1, \ldots, B_q \in T(\Sigma, V)_{\mathsf{Bool}}$ are basic predicate terms not containing quantifiers, the connectives $\wedge$ and $\vee$, or the operator $\longrightarrow$,

(5) $\bigcup_{i=1}^{p} var(b_i) \subseteq var(\overline{\gamma})$, i. e. no extra variables with respect to $\overline{\gamma}$ in the b_i,

(6) $\bigcup_{i=1}^{n} var(\gamma_i) \subseteq var(\overline{\gamma})$, i. e. no extra variables with respect to $\overline{\gamma}$ in the γ_i,

(7) $\forall k \in [q] : var(B_k) \not\subseteq var(\overline{\gamma})$, i. e. extra variables with respect to $\overline{\gamma}$ in each of the B_k,

(8) $[var(\overline{\gamma}') \cup \bigcup_{k=1}^{q} var(B_k)] \subseteq [var(\overline{\gamma}) \cup \bigcup_{j=1}^{n} var(\gamma'_j)]$, i. e. all variables of the rule are contained in (initial and final) configuration terms,

(9) $\forall j \in [n] : L_j \in \{1, *\}$. If $L_j = *$, then $\gamma'_j \in T'$. Here, L_j is not a label in the sense of Def. 5.1, but a modifier for the relation symbol $\longrightarrow$, indicating whether the relation itself ($L_j = 1$) or its reflexive and transitive closure ($L_j = *$) is meant.[1]

Note that the condition $\overline{\gamma} \in \Gamma' \setminus T'$ ensures that the requirement $\mathrm{dom}(\longrightarrow) \cap T = \emptyset$ is observed.

The b_i can be thought of as preconditions for the transition, and the B_k as postconditions (see the explanations after Def. 5.5).

The condition in (3) on the syntactic form of the γ'_j restricts them essentially to tuples of variables, constants, or other tuples (an injection term $\ulcorner \mathsf{mk} - \mathsf{T}(\mathsf{x}) \urcorner$ also counted as a tuple of length 1). The γ'_j merely serve as patterns for the targets of transitions starting in the γ_j, with the extra variables as the parts that may be modified; the restricted syntactic form of the γ'_j just allows to give names to subcomponents. If a term t is desired as one of the γ'_j, then a new extra variable x can be used instead, and the equation $\ulcorner x = t \urcorner$ can be added to the B_k.

In Section 5.2.2 below, this format will be compared with other rule formats for SOS definitions. In order to have a short name for it, it will be called ***ptp/t* format** (for "predicates-transitions-predicates / transition" or "precondition-transitions-postconditions / transition").

[1]Since $\xrightarrow{1} = \longrightarrow$, in the following the labels $L_j = 1$ will be omitted.

Example 5.4

As a simple example for the *ptp/t* format, consider the following SOS system (having no semantic significance or connection to the definitions used in the other examples). Configurations are either processes or a special symbol signalling termination:

$$\Gamma =_{df} \mathit{Proc} \cup T \qquad T =_{df} \{\underline{\mathsf{ended}}\}$$

where the language of processes contains an operator $\Box$:

$$\mathit{Proc} \ni p ::= p_1 \Box p_2 \mid \ldots$$

Assume a predicate $\mathit{special} : \mathit{Proc} \to \mathbb{B}$, and consider the rule

$$\frac{\vdash \mathit{special}(p_1) \ \wedge \ p_1 \longrightarrow p_1' \ \wedge \ p_2 \xrightarrow{*} \underline{\mathsf{ended}} \ \wedge \ \mathit{special}(p_1')}{\vdash p_1 \Box p_2 \longrightarrow \underline{\mathsf{ended}}}$$

In this example,

- $p = 1$, with $b_1 = \ulcorner \mathit{special}(p_1) \urcorner$;
- $n = 2$, with $L_1 = 1, \gamma_1' = \ulcorner p_1' \urcorner$ (this is an extra variable with respect to $p_1 \Box p_2$) and $L_2 = *, \gamma_2' = \ulcorner \underline{\mathsf{ended}} \urcorner$ (note that γ_2' is a terminal configuration which is required by condition (9) of Def. 5.3);
- $q = 1$, with $B_1 = \ulcorner \mathit{special}(p_1') \urcorner$ (contains an extra variable)

$\Box$

The semantics of this kind of inference rules is as usual: An instance of the conclusion is established if a corresponding instance of the hypothesis[2] can be established using the rules of the system. All variables of the rules are implicitly universally quantified.

As a consequence of this definition, the restriction to conjunctions in the hypothesis does not limit the expressive power of the formalism. Any quantifier-free hypothesis can be implemented by first transforming it into disjunctive normal form and then splitting the rule into several rules with the same conclusion, each component of the disjunction forming the hypothesis of a separate rule. All these rules have the required form, and their collection is semantically equivalent to the original, single rule.

For the formal treatment of transition sequences, a more syntactic definition for the semantics of our inference rules is needed. Remember that $=_E$ is the equality induced by the standard interpretation $\mathcal{E}$ (cf. Def. 4.1, p. 39).

[2]Note that there can be more than one corresponding instance if not all extra variables of the rule occur in the conclusion.

Definition 5.5 (application of an SOS deduction rule)

Let $(\Gamma, T, \longrightarrow)$ be a transition system represented by the SOS deduction system given by the term sets $T(\Sigma, V), \Gamma'$, and T', and a set of SOS deduction rules. Let a particular rule R of this system be given as

$$R: \quad \frac{\vdash \bigwedge_{i=1}^{p} b_i \ \wedge \ \bigwedge_{j=1}^{n} \gamma_j \xrightarrow{L_j} \gamma'_j \ \wedge \ \bigwedge_{k=1}^{q} B_k}{\vdash \overline{\gamma} \longrightarrow \overline{\gamma}'}$$

as in Def. 5.3. Let $\gamma, \gamma' \in \Gamma'$. Then we can derive $\gamma \longrightarrow \gamma'$ **using rule** $\boldsymbol{R}$ iff there is a substitution σ on *var* $(\overline{\gamma})$ such that

(1) $\gamma =_E \overline{\gamma}\sigma$

(2) $\forall i \in [p] : b_i\sigma =_E$ true

(3) There is an extension σ' of σ onto *var* (R) such that

(3.1) $\forall j \in [n] : \gamma_j\sigma' \xrightarrow{L_j} \gamma'_j\sigma'$ using the rules of the system

(3.2) $\forall k \in [q] : B_j\sigma' =_E$ true

(3.3) $\gamma' =_E \overline{\gamma}'\sigma'$

Remarks:

(1) If there are no extra variables with respect to $\overline{\gamma}$ that occur in R, then $\sigma' = \sigma$.

(2) Basically, the existence of σ' means the existence of suitable configurations $\gamma_j\sigma'$ and $\gamma'_j\sigma'$.

(3) The equalities $b_i\sigma =_E$ true and $B_j\sigma' =_E$ true are consequences of the underlying equational specification.

The b_i model "preconditions" that restrict the set of possible initial configurations for the rule, whereas the B_j model "postconditions" that must hold for the final configurations in the hypothesis.

(4) If $\gamma_j \xrightarrow{*} \gamma'_j$ occurs in the hypothesis, then γ_j must be (the representation of) a terminal configuration. The reason is that otherwise there would be no way of knowing when to stop the transition process in the hypothesis.

When dealing with practical semantics definitions, one may come across rules that do not quite conform with the rather strict *ptp/t* format. But most of these rules can be easily transformed to fit into it:

(1) Rules usually have *side conditions* restricting their applicability. These conditions can safely be included into the hypothesis.

(2) In non-trivial examples, there are often *several transition systems interconnected*: The hypotheses $\gamma_j \xrightarrow{L_j} \gamma'_j$ do not necessarily refer to the same transition relation. This is the case when one large system has been built up from smaller ones for reasons of modularity.

As an example, consider a language of statements of which one possible kind is an assignment $x := e$ where x is some variable and $e \in$ *Expr* is some expression. Let both statements $s \in$ *Stmt* and expressions depend on environments $\rho \in$ *Env*. So the sets of configurations for the SOS systems are *Stmt* $\times$ *Env* and *Expr* $\times$ *Env*, respectively, with terminal configuration sets *Env* and *Val* (some set of values). Then the system for statements could make use of the expression system in the following way:

$$\frac{\vdash \langle\, e, \rho \,\rangle \longrightarrow_{Expr} v}{\vdash \langle\, x := e, \rho \,\rangle \longrightarrow_{Stmt} \rho[v/x]}$$

($\rho[v/x]$ denotes the environment ρ, modified in such a way that the value v is stored at position x.)

The definitions can easily be adapted to this case by requiring rules to be of the form

$$R: \quad \frac{\vdash \bigwedge_{i=1}^{p} b_i \ \wedge \ \bigwedge_{j=1}^{n} \gamma_j \xrightarrow{L_j}_j \gamma'_j \ \wedge \ \bigwedge_{k=1}^{q} B_k}{\vdash \overline{\gamma} \longrightarrow \overline{\gamma}'}$$

where the $\longrightarrow_j$ are the transition relations of the respective systems. Furthermore, the sets of terms have to be extended appropriately to include all representations of the different configurations.

When reasoning about such systems built up from several smaller systems, usually all the transition relations can and will be collected under one global relation unless there is a need to discriminate the subrelations.

From Def. 5.5, it can immediately be deduced that transitions can be blurred as long as configurations stay within $=_E$ congruence classes:

Lemma 5.6

Let $\gamma, \gamma', \gamma_1, \gamma'_1 \in \Gamma'$ with $\gamma =_E \gamma_1, \gamma' =_E \gamma'_1$, and let R be a rule of the SOS deduction system defining the transition relation $\longrightarrow_{\mathcal{S}}$. Then $\gamma \longrightarrow_{\mathcal{S}} \gamma'$ using rule R iff $\gamma_1 \longrightarrow_{\mathcal{S}} \gamma'_1$.

Proof

Follows from conditions (1) and (3.3) in Def. 5.5 and the fact that $=_E$ is transitive.

□

By Lemma 5.6, an SOS deduction sequence can be put together from single steps $\gamma_1^{(1)} \longrightarrow_{\mathcal{S}} \gamma_1^{(2)}$, $\gamma_2^{(1)} \longrightarrow_{\mathcal{S}} \gamma_2^{(2)}, \ldots$ not only if $\gamma_i^{(2)} = \gamma_{i+1}^{(2)}$ for all i; for this purpose, it suffices to require $\gamma_i^{(2)} =_E \gamma_{i+1}^{(2)}$. In order to have a consistent treatment for deduction sequences of any length, the sequences of length 0 are defined accordingly:

Definition 5.7

Let $(\Gamma, T, \longrightarrow)$ be a transition system as in Def. 5.5, and let $\gamma, \gamma' \in \Gamma$. Then $\gamma \xrightarrow{0}_{\mathcal{S}} \gamma'$ $\Leftrightarrow_{df}$ $\gamma =_E \gamma'$.

This definition emphasizes once more that transitions with $\longrightarrow_{\mathcal{S}}$ are taken modulo $=_E$.

When comparing transition sequences with rewriting sequences, an "exact" version of SOS-rule application will also be needed:

Definition 5.8 (exact application of an SOS deduction rule)

Let $(\Gamma, T, \longrightarrow)$ be a transition system represented by the SOS deduction system given by the term sets $T(\Sigma, V), \Gamma'$, and T', and a set of SOS deduction rules. Let a particular rule R of this system be given as

$$R: \quad \frac{\vdash \bigwedge_{i=1}^{p} b_i \wedge \bigwedge_{j=1}^{n} \gamma_j \xrightarrow{L_j} \gamma_j' \wedge \bigwedge_{k=1}^{q} B_k}{\vdash \overline{\gamma} \longrightarrow \overline{\gamma}'}$$

as in Def. 5.3. Let $\gamma, \gamma' \in \Gamma'$. Then we can derive $\gamma \longrightarrow^{=} \gamma'$ **using rule R** iff there is a substitution σ on $var(\overline{\gamma})$ such that

(1) $\gamma = \overline{\gamma}\sigma$

(2) $\forall i \in [p] : b_i\sigma =_E \mathsf{true}$

(3) There is an extension σ' of σ onto $var(R)$ such that

(3.1) $\forall j \in [n] : \gamma_j\sigma' \xrightarrow{L_j} \gamma_j'\sigma'$ using the rules of the system

(3.2) $\forall k \in [q] : B_j\sigma' =_E \mathsf{true}$

(3.3) $\gamma' = \overline{\gamma}'\sigma'$

This definition differs from the previous one only in conditions (1) and (3.3); there only matching modulo $=_E$ was required, but in Def. 5.8 it is demanded that a configuration match exactly the left-hand side of the conclusion of a rule. The relation between the two definitions is expressed by

Lemma 5.9

Let $\gamma, \gamma' \in \Gamma'$. Then

$$\gamma \longrightarrow_{\mathcal{S}} \gamma' \iff \exists \tilde{\gamma}, \tilde{\gamma}' \in \Gamma' : \gamma =_E \tilde{\gamma} \longrightarrow_{\mathcal{S}}^{=} \tilde{\gamma}' =_E \gamma'$$

Proof

Obvious from the definitions.

□

As an additional requirement for SOS deduction systems, it will be demanded that the rules do not permit non-terminating proof attempts. The simplest example for a rule that is forbidden by this is

$$\frac{\vdash \gamma \longrightarrow \gamma'}{\vdash \gamma \longrightarrow \gamma'}$$

It has the form of Def. 5.3, but it cannot be used for deriving any transition according to Def. 5.5. A way to exclude such unpleasant behaviour is to demand that all transitions in the premise of a rule be smaller than the transition in the conclusion with respect to some well-founded ordering.

The existence of such an ordering is sufficient to prevent non-terminating proof attempts (see Section 3.1.3).

Since the problem only occurs if transitions between representations of actual configurations (without variables) are to be proved, it suffices to consider only "ground" (variable-free) instances of the rules. This leads to the following

Requirement 5.10 (well-foundedness of SOS systems)

For every SOS system, there must be given a well-founded ordering $\sqsubseteq$ on transitions such that for every ground instance

$$\frac{hyp}{conc}$$

of a rule of the system and all transitions t_1 in *hyp* and t_2 in *conc*, $t_1 \sqsubseteq t_2$ holds.

It is rather natural to have such well-foundedness requirements; Aceto, Bloom, and Vaandrager, e. g. , construct in [2] a well-founded ordering of this kind based on weights assigned to the operators.

5.2.1 Examples of SOS Definitions

The examples for operational semantics definitions are taken from Lakhneche [82] where the semantics of a programming language named $\mathrm{PL}_0^{\mathrm{R}}$ is defined both operationally and denotationally. Not all the details that are contained will be described, but just as much as is needed to understand the format of the rules.

Consider the following part of the semantics definitions for $\mathrm{PL}_0^{\mathrm{R}}$ expressions:

$$Expr \ni exp \ ::= \ int \mid var \mid mop \ exp \mid \ldots$$

where *int* stands for representations of integers, $var \in Name$ for variable identifiers, and *mop* for monadic operators.

Basic domains for the semantics definition include

$val \in \mathsf{Val}$	expression values
$loc \in \mathsf{Loc}$	storage locations
$\rho \in \mathsf{OpEnv} = Name \to \mathsf{Loc}$	operational environments
$\sigma \in \Sigma = \mathsf{Loc} \to \mathsf{Val}$	stores
$\delta \in \mathsf{Dict}$	static environments (mapping names to types)

Let $\boldsymbol{\alpha}$ be a basic interpretation that gives values to all the constants. (For the purpose of this exposition, it need not be defined in more detail.)

The transition system for expressions is indexed with a static environment δ and an operational (dynamic) environment ρ. The configuration sets are

$$\begin{aligned}
&\Gamma^{\delta}_{Expr} =_{df} \\
&\quad \{\langle\, exp, \sigma \,\rangle \in Expr \times \Sigma \mid exp \text{ can be typed as } \mathsf{Integer} \text{ or } \mathsf{Boolean} \text{ under } \delta\} \\
&\quad \cup \mathsf{T}_{Expr} \\
&\mathsf{T}_{Expr} =_{df} \mathsf{Val}.
\end{aligned}$$

In the following, the decoration "$'$" will be suppressed when talking about sets of term representations unless an explicit distinction is important.

The indexing with δ and ρ is written in the rules as $\rho \vdash_\delta \ldots$ But the form of Def. 5.3 can be easily retained by including the indices as additional components of the configurations. (This is done in the implementation of the rules described in Chapter 10.)

The first example for a rule is an example for an axiom schema where $n, p, q = 0$:

$$\text{(EO1)} \qquad \rho \vdash_\delta \langle\, int, \sigma \,\rangle \longrightarrow_{Expr} \alpha\,(int)$$

This rule basically says that the semantics of an integer is determined by the underlying interpretation α.

The next example is an inference rule schema with $n = q = 1, p = 0$:

$$\text{(EO6)} \qquad \frac{\rho \vdash_\delta \langle\, exp, \sigma \,\rangle \longrightarrow_{Expr} val \wedge val \neq \texttt{error}}{\rho \vdash_\delta \langle\, mop\ exp, \sigma \,\rangle \longrightarrow_{Expr} \alpha\,(\, mop\,)\,(val)}$$

Here, we have an occurrence of an extra variable: *val* is not contained in the starting configuration $\langle\, mop\ exp, \sigma \,\rangle$, but rather is (part of) the result of an intermediate transition step. `error` is a constant element of Val indicating faulty evaluations.

In order to see a rule that does not conform with Def. 5.3, consider the semantics of $\mathrm{PL}_0^{\mathrm{R}}$ blocks:

$$Block \ni blk \ ::= \ decl \ : \ blk \mid proc \mid \rho \ : \ blk$$

A block is either a sequential process, or it is a block with a preceding variable declaration. For the semantics definition, a third possibility is a block with a preceding operational environment (this mixture of syntactic and semantic elements, however, cannot occur in program texts).

The sets of configurations are again indexed with a static environment δ:

$$\begin{aligned} \Gamma^{\delta}_{Block} &=_{df} \\ &\{\langle\, blk, \sigma \,\rangle \in Block \times \Sigma \mid blk \text{ is statically well-formed under } \delta\} \\ &\cup \mathsf{T}_{Block} \\ \mathsf{T}_{Block} &=_{df} \{\texttt{terminated}, \texttt{stopped}, \texttt{invalid}\} \times \Sigma \end{aligned}$$

Terminal configurations contain a state and a flag that indicates in what way this state was reached.

One of the rules for blocks has a non-computational (non-constructive) hypothesis:

$$\text{(OB4)} \qquad \frac{\rho \oplus \rho_1 \vdash_{\delta \oplus \delta_1} \langle\, decl : blk, \sigma \,\rangle \xrightarrow{\tau}_{Block} \langle\, \rho_2 : blk, \sigma \,\rangle \wedge \rho_1 : \delta_1 \wedge \operatorname{dom}\delta \cap \operatorname{dom}\delta_1 = \emptyset}{\rho \vdash_\delta \langle\, \rho_1 : decl : blk, \sigma \,\rangle \xrightarrow{\tau}_{Block} \langle\, \rho_1 \oplus \rho_2 : blk, \sigma \,\rangle}$$

$\rho \oplus \rho_1$ is the modification of ρ by ρ_1. $\rho_1 : \delta_1$ means that every identifier is mapped by ρ_1 to a value of a type that is allowed by δ_1. This rule expresses that declarations can be evaluated in sequential order provided no type conflicts occur, i. e. a δ_1 with the above properties exists. δ_1 is the interesting bit of the rule: It is an extra variable that is not contained in the conclusion at all, and furthermore, it is already part of the left-hand side of the transition in the hypothesis. This last point renders the rule non-constructive: There is no explicit hint how δ_1 should be derived from the known parameters when the rule is to be applied. This is the reason why the left-hand sides of transitions in the premise of a rule are required not to contain extra variables; rules of the form (OB4) will not be considered further.

5.2.2 Other Rule Formats

In the literature, some formats for SOS rules have been presented and their properties have been investigated. In this section, three of them will be reviewed (the format proposed by De Simone, the GSOS format of Bloom, Istrail and Meyer, and the *tyft/tyxt* format of Groote and Vaandrager), and their relation to the *ptp/t* format of Def. 5.3 will be examined. All the other formats have been developed for the specification of concurrent systems; therefore the transition relations that are specified by such rules are all labelled (but see Lemma 5.2 on page 54). So assume a signature Σ, a set of variables V and a set of labels A.

Definition 5.11 (De Simone rule format [31])

A **De Simone rule** is an inference rule schema of the form

$$\frac{\vdash \bigwedge_{i\in I} x_i \xrightarrow{a_i} y_i}{\vdash f(x_1,\ldots,x_n) \xrightarrow{a} t}$$

where

(1) $f \in \Sigma_{w,s}$ with $|w| = n \in \mathbb{N}$,
(2) $I \subseteq [n]$,
(3) $x_1,\ldots,x_n$ and y_j for $j \in I$ are all distinct variables,
(4) $t \in T(\Sigma,\{x'_1,\ldots,x'_n\})$, where for $i \in [n]$ we let $x'_i =_{df} y_i$ if $i \in I$ and $x'_i = x_i$ otherwise, and each x'_i occurs at most once in t,
(5) $a, a_i \in A$ (for $i \in [n]$).

Obviously, this format is a restricted version of the *ptp/t* format. The terms in transitions are required to be simpler here, and there is no possibility to define Boolean conditions not expressed by transitions.[3] The restrictions about extra variables are essentially the same in both rule formats.

A more general format than the De Simone format is the *tyft/tyxt* format:

Definition 5.12 (*tyft/tyxt* format; Groote and Vaandrager [60])

Let $f \in \Sigma_{w,s}$ with $|w| = n \geq 0$.

A ***tyft* rule** is an inference rule schema of the form

$$\frac{\vdash \bigwedge_{i\in I} t_i \xrightarrow{a_i} y_i}{\vdash f(x_1,\ldots,x_n) \xrightarrow{a} t}$$

where

[3] The inclusion of such conditions, however, should introduce no additional difficulties. The evaluation of conditions could for example be encoded by introducing a new transition relation $\longrightarrow_{\mathsf{Bool}}$, defined by SOS rules in the appropriate format.

(1) I is some index set,
(2) $x_1, \dots, x_n$ and y_j for $j \in I$ are all distinct variables,
(3) $t, t_i \in T(\Sigma, V)$ (for $i \in I$),
(4) $a, a_i \in A$ (for $i \in [n]$).

A ***tyxt*** **rule** is of the form

$$\frac{\vdash \bigwedge_{i \in I} t_i \xrightarrow{a_i} y_i}{\vdash x \xrightarrow{a} t}$$

where

(1) I is some index set,
(2) x and y_j for $j \in I$ are all distinct variables,
(3) $t, t_i \in T(\Sigma, V)$ (for $i \in I$),
(4) $a, a_i \in A$ (for $i \in [n]$).

A ***tyft/tyxt*** **system** is a system of rules whose format is either *tyft* or *tyxt*.

Definition 5.13 (well-founded *tyft/tyxt* rule)

Let R be a *tyft/tyxt* rule as in Def. 5.12; let the transitions of the premise of R be $\{t_i \xrightarrow{a_i} y_i\}$. The **dependency graph** of R is a directed graph with $\bigcup_{i \in I} \mathit{var}(t_i \xrightarrow{a_i} y_i)$ as the set of nodes and $\{\langle x, y \rangle \mid x \in \mathit{var}(y_i), i \in I\}$ as the set of edges.

R is called **well-founded** iff there are no infinite chains of edges in its dependency graph.

Example 5.14 (cf. Groote and Vaandrager [60])

An example for a *tyft/tyxt* rule that is not well-founded is the following:

$$R : \frac{\vdash f(x, y) \xrightarrow{a} y', g(x', y') \xrightarrow{b} y}{\vdash x \xrightarrow{c} x'}$$

The dependency graph of R is

$$\begin{tikzcd} x \arrow[r] & y' \arrow[rr, bend left] & & y \arrow[ll, bend left] & x' \arrow[l] \end{tikzcd}$$

It contains a cycle; y and y' cannot be determined independently.

□

Definition 5.15 (pure *tyft/tyxt* rule)

Let R be a *tyft/tyxt* rule as in Def. 5.12. R is called **pure** iff it is well-founded and all the variables in R occur on the left-hand side of the conclusion or on the right-hand side of a transition in the premise.

There are some differences between the *tyft/tyxt* format and the *ptp/t* format:

(1) In *tyft/tyxt* format, the complexity of terms is more restricted.

(2) The conditions on extra variables are more restrictive in *ptp/t* format.

(3) The *tyft/tyxt* format does not allow additional Boolean conditions.

(4) In *ptp/t* format, many-step transitions are allowed in premises if they lead to terminal configurations.

(3) can be neglected as in the De Simone case above.[4] But (1) and (2) make the *tyft/tyxt* and *ptp/t* formats incomparable. Groote and Vaandrager [60] demonstrate that more complex terms should not be allowed since otherwise pleasant mathematical properties of *tyft/tyxt* systems would disappear; so (1) is a "hard" difference that makes the *ptp/t* format more general in this aspect. On the other hand, there are no conditions on extra variables in the general *tyft/tyxt* Def. 5.12. But as the limitations in Def. 5.3 are quite important to achieve the simulation properties of Chapter 7, (2) is a hard difference that makes the *tyft/tyxt* format more general in this aspect.

The two kinds of formats become closer to each other if only pure *tyft/tyxt* systems are considered. In this case, the difference with respect to extra variables is that a premise like

$$t \longrightarrow x \;\wedge\; f(x) \longrightarrow y$$

(where x and y are extra variables with respect to the left-hand side of the conclusion) is legal in pure *tyft/tyxt* format, but not allowed *ptp/t* format. In principle, there is no problem in also allowing such premises in Def. 5.3, but so far, it did not seem necessary. So this difference is not really hard; pure *tyft/tyxt* systems can be viewed as special cases of *ptp/t* systems.

A format that deviates farther is the GSOS format:

Definition 5.16 (GSOS format[5]; cf. Bloom, Istrail and Meyer [12])

A GSOS rule is of the form

$$\frac{\vdash \bigwedge\{x_i \xrightarrow{a_{ij}} y_{ij} \mid i \in [n], j \in [m_i]\} \;\wedge\; \bigwedge\{x_i \stackrel{b_{ij}}{\not\longrightarrow} \mid i \in [n], j \in [n_i]\}}{\vdash f(x_1, \ldots, x_n) \xrightarrow{a} t}$$

where:

(1) $f \in \Sigma_{w,s}$ with $|\,w\,| = n$

[4] In fact, there is a rule format called *path* **format** (for "predicates and *tyft/tyxt* hybrid format") that extends the *tyft/tyxt* format by Boolean conditions (see Baeten and Verhoef [5]).

[5] Originally, the "G" in GSOS stood for "guarded" since there was an additional condition about guardedness of fixed point expressions (which will not be considered here). If this aspect is neglected, the "G" might also stand for "grand" since this format is very general (see Aceto, Bloom, and Vaandrager [2]).

(2) all occurring variables are distinct
(3) for $i \in [n] : m_i, n_i \geq 0$
(4) $t \in T(\Sigma, \{x_i, y_{ij} \mid i \in [n], j \in [m_i]\})$
(5) $a, a_{ij}, b_{ik} \in A$ for $i \in [n], j \in [m_i], k \in [n_i]$

Besides the differences in term complexity and in allowing Boolean conditions and many-step transitions as in the other two cases, the main difference between the GSOS format and all of the other three rule formats is the existence of negative premises $x_i \overset{b_{ij}}{\not\longrightarrow}$. Without such premises, GSOS rules are special cases of pure *tyft/tyxt* rules, and hence they fit into *ptp/t* format. But negative premises in general cannot be expressed with rules according to Def. 5.3. There the only possibility is to demand that a configuration be terminal, a property which is syntactically checkable. But the semantic property that a non-terminal configuration is stuck, i. e. without a $\longrightarrow$ successor, cannot be expressed. So this aspect of the GSOS format goes beyond the *ptp/t* format. A *ptp/t* system describes transitions only in terms of possible events, whereas a GSOS system is also able to consider events that are not possible.

The introduction of the three formats of this section was motivated by the wish to guarantee certain pleasant mathematical properties of the transition relation defined by an SOS system. The objective of this work, however, is to take an SOS definition with rules in a not too restrictive format and to simulate the resulting transition relation by the rewrite relation of a term rewriting system as closely as possible. If the transition relation is well-behaved, so is the rewriting relation. But if the transition relation fails to possess some desirable property, this is also true for the rewriting relation.

Chapter 6

Denotational Semantics

Following the examination of structured operational semantics definitions, now denotational descriptions of programming languages shall be considered. As in the previous chapter, the emphasis will be on syntactic aspects concerning the representation of such descriptions rather than on the underlying mathematical theory. This chapter does not present new material, but rather aims at presenting a generally accepted background for denotational definitions.

The structure of such definitions follows a rather strict scheme:

(1) First, the **syntax** of the language L whose semantics is to be defined is described.

(2) Second, the domains of **semantic values** are introduced.

(3) Third, the **semantic functions** are defined that map syntactic objects to semantic values.

This format is widely used in the literature as it facilitates a clear separation of the various aspects of such definitions. Examples can be found e. g. in Stoy [113], Gordon [58], de Bakker [29], Loeckx and Sieber [86] and Mosses [95]. The examples of Lakhneche [82] also comply with it.

Following the above structure, this chapter starts with an overview of the syntactic description technique commonly used, succeeded by a short introduction to the problem of constructing appropriate semantic domains. The latter section also contains a short introduction into the theory of complete partial orders. This mathematical structure provides a suitable setting for the semantic definition of recursive structures such as loops or procedures.

Finally, the commonly used syntactic form for the definition of denotational functions mapping syntactic to semantic objects will be explained. Although the mathematical background of such definitions is usually much more complex than in the SOS case, their syntactic form will turn out to be very regular and rather simple. This section also includes some examples from actual semantics definitions taken from Lakhneche [82]; these definitions will also be used in the proofs described in Chapter 10.

6.1 Definition of the Syntactic and Semantic Domains

6.1.1 Syntactic Domains

The first part of a definition of this kind describes the **syntax** of the language L. Usually, abstract syntax (see McCarthy [89]) is used, based on a definition of the syntactic structure by means of a context-free grammar in Backus-Naur form [97]. Basically, this is equivalent to a definition of a set of equations between the syntactic domains. The variables for syntactic structures in the grammar can be replaced by the corresponding domains, and the operators combining syntactic entities by corresponding operators on the syntactic domains. As a simple example, consider a part the syntax of $\mathrm{PL}_0^{\mathrm{R}}$ expressions given by Lakhneche [82]:

> *An expression is a constant symbol $int \in Int$, `TRUE` or `FALSE`, a variable, or it is constructed from sub-expressions and operator symbols.*
>
> $exp \in Expr$
>
> $exp \quad ::= \quad var \mid int \mid \mathtt{TRUE} \mid \mathtt{FALSE} \mid\ mop\ exp \mid exp_1\ dop\ exp_2$

6.2 Semantic Domains

Usually, the mathematical spaces of semantic values are defined by a set of recursive equations. Among the operators used in these equations are (disjoint) union ($\_ \cup \_$, also written $\_ \mid \_$ or $\_ + \_$), product ($\_ \times \_$), function construction ($\_ \rightarrow \_$) and tuple ($\_^*$). (See Gordon [58] for a short introduction.)

In general, there are no *sets* satisfying a set of domain equations, as the example

$$E = E \rightarrow E$$

shows: no set is equal to its own function space, and even if "=" is interpreted as "is isomorphic to", this relation does not hold for sets containing more than one element. But if the more complicatedly structured *domains* are used instead of sets, together with an appropriate interpretation of the operators (and the equality) mentioned above, domain equations do have a solution. The mathematical theory of domains will not be considered here, however; see Stoy [113] or Gunter and Scott [61] for a detailed account.

A problem related to the recursive definition of sets (or domains) is that of recursive definition of functions, in particular that of the existence of fixed points. This problem occurs when values shall be assigned to structures such as loops or recursive procedures. If such constructs are part of the language L, a mathematical structure must be used that guarantees the existence of fixed points, and moreover, it should allow the selection of a specific one in order to make the definition as exact as possible.

Complete partial orders as defined in the following provide a comparatively simple setting in which the existence of a certain kind of fixed point (the least defined one) can be guaranteed for a well-defined class of functions (the continuous ones). The exposition mainly follows de Bakker [29].

Definition 6.1 (least upper bound)

Let $\langle M, \sqsubseteq \rangle$ be a partially ordered set, $X \subseteq M$ and $z \in M$. z is called the **least upper bound (lub)** of X (written $\boldsymbol{z = \bigsqcup X}$) iff

(1) $X \sqsubseteq z$ (i. e. z is an upper bound of X) and

(2) $\forall z' \in M : X \sqsubseteq z' \;\Rightarrow\; z \sqsubseteq z'$ (i. e. all other upper bounds are larger than z).

Definition 6.2 (chain)

Let $\langle M, \sqsubseteq \rangle$ be a partially ordered set. An **(ascending) chain** in M is a sequence $\langle x_i \rangle_{i \in \mathbb{N}}$ in M such that for all $i \in \mathbb{N} : x_i \sqsubseteq x_{i+1}$.

Definition 6.3 (complete partial order)

Let $\langle M, \sqsubseteq \rangle$ be a partially ordered set. $\langle M, \sqsubseteq \rangle$ is a **complete partial order (cpo)** iff

(1) there exists a least element $\bot \in M$, i. e. with $\bot \sqsubseteq m$ for all $m \in M$, and

(2) each chain $\langle x_i \rangle_{i \in \mathbb{N}}$ in M has a lub $\bigsqcup\limits_{i \geq 0} x_i$.

The following trivial lemma is useful to compute the lub of finite chains as these can be modelled by infinite ones that become constant at a certain index:

Lemma 6.4 (end-constant chains)

Let $\langle M, \sqsubseteq \rangle$ be a partially ordered set, $\langle c_i \rangle_{i \in \mathbb{N}}$ a chain in M and $n \in \mathbb{N}$ with $c_i = c_n$ for all $i \geq n$. Then $\bigsqcup\limits_{i \geq 0} c_i = c_n$.

Definition 6.5 (monotonic function)

Let $\langle C_1, \sqsubseteq_1 \rangle$ and $\langle C_2, \sqsubseteq_2 \rangle$ be partially ordered sets. A function $f : C_1 \to C_2$ is called **monotonic** iff, for all $x, y \in C_1$, $x \sqsubseteq_1 y$ implies $f(x) \sqsubseteq_2 f(y)$.

The set of monotonic functions from C_1 to C_2 is denoted by $\boldsymbol{C_1 \xrightarrow{\text{mon}} C_2}$.

The set of monotonic functions between cpo's forms a cpo (compare Appendix C, where it is proved that the set of all functions also forms a cpo with the same partial ordering):

Lemma 6.6 ($C_1 \xrightarrow{\text{mon}} C_2$ is a cpo, Lemma 3.9 of de Bakker [29])

Let $\langle C_1, \sqsubseteq_1 \rangle$ and $\langle C_2, \sqsubseteq_2 \rangle$ be cpo's. Then the relation $\sqsubseteq$ on $C_1 \xrightarrow{\text{mon}} C_2$ defined by

$$\forall f_1, f_2 \in C_1 \xrightarrow{\text{mon}} C_2 : (f_1 \sqsubseteq f_2 \Leftrightarrow_{df} \forall x \in C_1 : f_1(x) \sqsubseteq_2 f_2(x))$$

is a partial ordering, and $\langle C_1 \xrightarrow{\text{mon}} C_2, \sqsubseteq \rangle$ is a cpo. If $\langle f_i \rangle_{i \in \mathbb{N}}$ is a chain of functions in $C_1 \xrightarrow{\text{mon}} C_2$, then $\bigsqcup\limits_{i \geq 0} f_i = \lambda\, x.\, \bigsqcup\limits_{i \geq 0} f_i(x)$, i. e. for all $x \in C_1 : (\bigsqcup\limits_{i \geq 0} f_i)(x) = \bigsqcup\limits_{i \geq 0} (f_i(x))$.

Actually, it suffices for the proof of this lemma that only C_2 is a cpo and C_1 just an arbitrary partial order.

Definition 6.7 (fixed point, μf)

Let C be a cpo, $f : C \to C$ and $x \in C$.

(1) x is called a **fixed point** of f iff $f(x) = x$.

(2) x is called the **least fixed point** of f ($x = \boldsymbol{\mu f}$) iff x is a fixed point of f and $x \sqsubseteq y$ for each fixed point y of f.

Definition 6.8 (continuous function)

Let $\langle C_1, \sqsubseteq_1 \rangle$ and $\langle C_2, \sqsubseteq_2 \rangle$ be cpo's and $f : C_1 \xrightarrow{\text{mon}} C_2$. f is called **continuous** iff for every chain $\langle x_i \rangle_{i \in \mathbb{N}}$ in C_1 : $f(\bigsqcup_{i \geq 0} x_i) \sqsubseteq \bigsqcup_{i \geq 0} f(x_i)$.[1] The set of continuous functions from C_1 to C_2 is denoted by $[\boldsymbol{C_1} \longrightarrow \boldsymbol{C_2}]$.

Lemma 6.9 (Lemma 5.2 of de Bakker [29])

Let $\langle C_1, \sqsubseteq_1 \rangle$ and $\langle C_2, \sqsubseteq_2 \rangle$ be cpo's and $f : C_1 \xrightarrow{\text{mon}} C_2$. Then f is continuous iff $\bigsqcup_{i \geq 0} f(x_i) = f(\bigsqcup_{i \geq 0} x_i)$ for all chains $\langle x_i \rangle_{i \in \mathbb{N}}$ in C_1.

Proof

The "if" part is trivial. For the "only if" part, let $f \in [C_1 \longrightarrow C_2]$ and $\langle x_i \rangle_{i \in \mathbb{N}}$ be a chain in C_1. The goal is to prove $\bigsqcup_{i \geq 0} f(x_i) \sqsubseteq f(\bigsqcup_{i \geq 0} x_i)$. ($\langle f(x_i) \rangle_{i \in \mathbb{N}}$ is a chain because f is monotonic, hence $\bigsqcup_{i \geq 0} f(x_i)$ exists.)

Since $x_i \sqsubseteq \bigsqcup_{i \geq 0} x_i$ for all $i \geq 0$, it follows by monotonicity of f that $f(x_i) \sqsubseteq f(\bigsqcup_{i \geq 0} x_i)$ for all $i \geq 0$. So $f(\bigsqcup_{i \geq 0} x_i)$ is an upper bound of the chain $\langle x_i \rangle_{i \geq 0}$, and hence $\bigsqcup_{i \geq 0} f(x_i) \sqsubseteq f(\bigsqcup_{i \geq 0} x_i)$.

□

Lemma 6.10 (Lemma 5.3 of de Bakker [29])

Let C, C' be cpo's. Then $\langle [C_1 \longrightarrow C_2], \sqsubseteq \rangle$ is a cpo, where $\sqsubseteq$ is the ordering defined in Lemma 6.6.

Theorem 6.11 (Fixed Point Theorem, Theorem 5.8 of de Bakker [29])

Let C be a cpo and $f \in [C \longrightarrow C]$. Then f has a least fixed point μf satisfying

$$\mu f = \bigsqcup_{i \geq 0} f^i(\bot_C) \quad .$$

[1] The latter lub exists since f is monotonic.

By Theorem 6.11, each continuous function has a least fixed point. Moreover, it can be effectively approximated by successively calculating the values in the chain $\langle f^i(\bot_C) \rangle_{i \geq 0}$ (this process is called *fixed point iteration*).

In the denotational definitions considered later on, the semantics of all the constructs in the languages will be functions taking their arguments and values in some cpo's. Least fixed points of continuous functions will be used to capture the meaning of recursive constructs like loops or procedure calls; for examples, see below in Section 6.3.1.

In the literature, cpo's $\langle M, \sqsubseteq \rangle$ are also defined in ways that differ from Def. 6.3. A least element is mostly required, but the other condition varies. Instead of demanding that each *chain* have a lub, one can demand that this hold for arbitrary *totally ordered subsets* of M (see Loeckx and Sieber [86]) or, still more general, for *directed sets*[2] (see Gunter and Scott [61]).

When solutions for general recursive domain equations are needed, the simple concept of cpo's alone does not suffice anymore. But it forms the basis for domain theory; the kinds of domains that provide the solutions are cpo's with certain additional properties [61].

The approach taken here, however, will follow de Bakker and only consider the special case of countable chains. More complex structures than cpo's will not be considered, either. This is not too severe a restriction, as can be seen by the examples de Bakker is able to treat with this approach.

6.3 Format of Denotational Function Definitions

From a syntactic point of view, denotational semantics definitions of the form we will present shortly are much simpler than operational ones. Basically, they consist of a set of conditional equations that must obey a rather strict format.

In the following, it will be assumed that the syntactic categories of the language L whose semantics is to be defined are given by a context-free grammar. As in the operational case, let Σ be a signature and V a set of variables such that all mathematical objects that are needed can be represented in $T(\Sigma, V)$. As usual, mathematical objects and their term representations will be identified if no confusion can arise.

The denotational semantics for a syntactic category C is given by a function

$$[\![\cdot]\!] : C \to D$$

where D is some value domain. In the examples, it will be a cpo; typically, elements of D will be functions rather than simple data (see below in Section 6.3.1). The standard mathematical meta-language used to define semantic functions contains the following constructs:

- **conditional expressions**:

  $\mathsf{if}\ \mathit{condition}\ \mathsf{then}\ \mathit{expression}_1\ \mathsf{else}\ \mathit{expression}_2$

  An alternative notation used e. g. by Stoy [113] and Gordon [58] is the following:

  $\mathit{condition} \to \mathit{expression}_1, \mathit{expression}_2$

[2]A set $A \subseteq M$ is **directed** iff every finite subset of A has an upper bound in M.

- **explicit λ-abstractions** to construct unnamed functions;
- the **μ-operator** mapping a continuous function to its least fixed point;
- **let expressions** or, as an alternative form, where expressions (see Section 3.2):

$$\ulcorner \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 \urcorner \quad \equiv \quad \ulcorner e_2\ \mathsf{where}\ x = e_1 \urcorner$$

- the usual **mathematical term language**, in particular applications of expressions denoting functions to other expressions.[3]

The definition of such a function $[\![\cdot]\!]$ is required to be given in a very regular way:

Definition 6.12 (denotational function definition in clausal form)

Let C be a syntactic category defined by the context-free productions

$$C \ni q ::= q_1 \mid \dots \mid q_n \qquad (n \geq 1)$$

and $[\![\cdot]\!] : C \to D$ be the semantic function for D. The definition of $[\![\cdot]\!]$ is given in **clausal form** iff it consists of n clauses of the following structure

$$[\![c_i]\!]\ a_{i1} \dots a_{ip_i} = \begin{cases} \mathit{result}_{i1} & \text{, if } \mathit{condition}_{i1} \\ \vdots & \\ \mathit{result}_{im_i} & \text{, if } \mathit{condition}_{im_i} \end{cases} \tag{6.1}$$

where $m_i \geq 1$ and $p_i \geq 0$ for all $i \in [n]$ and for all $i \in [n]$, $j \in [m_i]$ and $k \in [p_i]$:

(1) $c_i \in T(\Sigma, V)_{C'}$ corresponds to the $i-$th possible structure q_i for C.

(2) The a_{ik} are additional arguments for the function $[\![\cdot]\!]$.
If $p_i > 0$, then $D = D_1 \to \dots \to D_{ip_i}$ for some domains $D_1, \dots, D_{ip_i}$ and $a_{ik} \in D'_{ik}$.

(3) $\mathit{result}_{ij} \in D'$.

(4) $\mathit{condition}_{ij} \in T(\Sigma, V)_{\mathsf{Bool}}$ without λ-abstractions.

(5) $\mathit{var}(\mathit{result}_{ij}) \cup \mathit{var}(\mathit{condition}_{ij}) \subseteq \mathit{var}(c_i)$.

(6) Either all the result_{ij} for a fixed $i \in [n]$ are λ-abstractions with the same bound variables, or none of them is.

(7) Occurrences of semantic functions in the result_{ij} and $\mathit{condition}_{ij}$ may only refer to proper subcomponents of c_i.

The additional arguments a_{ik} correspond to the case where the domain D is functional and the function $[\![\cdot]\!]$ is defined in a curried form (cf. Curry and Feys [25]), i. e. as

$$[\![\cdot]\!] : C \to D_1 \to \dots \to D_{ip_i}$$

instead of

$$[\![\cdot]\!] : C, D_1, \dots, D_{ip_{i-1}} \to D_{ip_i} \quad .$$

[3]Note that such terms can be represented in a first-order language by providing an application operator. Let A be a sort representing functions from $C \to D$ for some domains C and D (represented by C' and D', resp.), $f \in V_A$ and $c \in V_{C'}$. Then $\mathsf{apply}(f, c) \in D'$, where $\mathsf{apply} : A, C' \to D'$. (Compare Section 4.2.2.)

By (4), the conditions are simple terms without occurrences of explicitly defined functions, a complication that hardly ever occurs in practice and only would add unnecessary overhead to the simulation in Chapter 8. Condition (5) makes sure that the semantics of a construct c only depends on c (and its subcomponents), but not on other entities. (6) guarantees a uniform treatment of additional parameters for $[\![\cdot]\!]$. (7) is the requirement that denotational definitions be **compositional**, i. e. that the semantics of an expression be computable from the semantics of its subexpressions.

The meaning of a clause of the form (6.1) is a set of m_i conditional equations:

$$\begin{array}{rrcl}
\{ & \mathit{condition}_{i1} & \Rightarrow & [\![c_i]\!]\ a_{i1} \ldots a_{ip_i} = \mathit{result}_{i1}\ , \\
& \neg\mathit{condition}_{i1}\ \wedge\ \mathit{condition}_{i2} & \Rightarrow & [\![c_i]\!]\ a_{i1} \ldots a_{ip_i} = \mathit{result}_{i2}\ , \\
& \ldots, & & \\
& \bigwedge_{j<m_i} \neg\mathit{condition}_{ij}\ \wedge\ \mathit{condition}_{im_i} & \Rightarrow & [\![c_i]\!]\ a_{i1} \ldots a_{ip_i} = \mathit{result}_{im_i}\ \}\ ,
\end{array}$$

i. e. the conditions in (6.1) have to be checked from top to bottom to determine the correct result for an instance of c_i.

In order to make sure that a denotational definition specifies a function that is effectively computable an additional requirement is imposed on it that is similar to the one for SOS definitions (Requirement 5.10):

Requirement 6.13 (well-foundedness of denotational definitions)

For every system consisting of definitions in clausal form as defined in Def. 6.12, there must be given a termination ordering $\prec$ such that the right-hand sides of the denotational equations are smaller than the left-hand sides, i. e. $\mathit{result}_{ij}\sigma \prec [\![c_i]\!]\,\sigma$ and $\mathit{condition}_{ij}\sigma \prec [\![c_i]\!]\,\sigma$ for all $i \in [n], j \in [m_i]$ and ground substitutions $\sigma \in \mathit{Subst}\,(\Sigma, V)$.

This requirement should be easy to fulfil since by (7) in Def. 6.12, semantic functions on the right-hand sides of equations in (6.1) only refer to subcomponents of the argument on the left-hand side, and therefore the subterm ordering $\rhd$ could serve as the termination ordering (applied to arguments of semantic functions only, of course).

6.3.1 Examples for Denotational Definitions

As in the previous chapter, the formal definition will be illustrated with slightly simplified examples from the semantics definition for $\mathrm{PL}_0^{\mathrm{R}}$ expressions.

In addition to the basic domains from Section 5.2.1, the following occur:

$\mathit{env} \in \mathsf{DenEnv} = \mathit{Name} \to \mathsf{Loc} \uplus \{\bot_{\mathsf{DenEnv}}\}$	denotational environments
$\mathsf{DenVal} = \mathsf{Val} \uplus \{\bot_{\mathsf{DenVal}}\}$	denotational values

Both of these domains are provided with a partial ordering that makes them cpo's.

The semantic function $\mathcal{E}$ for expressions has the arity

$$\mathcal{E} : \mathit{Expr} \longrightarrow \mathsf{DenEnv} \longrightarrow \Sigma \longrightarrow \mathsf{DenVal}$$

Just like the SOS definition on p. 60, the definition of $\mathcal{E}$ depends on the underlying interpretation α for the operators. Example clauses are

$$\mathcal{E}[\![int]\!]\, \mathsf{env}\, \sigma \overset{\text{def}}{=} \alpha(int)$$

$$\mathcal{E}[\![\, mop\ exp]\!]\, \mathsf{env}\, \sigma \overset{\text{def}}{=} \begin{cases} \alpha(\, mop\,)\,(\mathcal{E}[\![exp]\!]\, \mathsf{env}\, \sigma)\,, & \text{if } \mathcal{E}[\![exp]\!]\, \mathsf{env}\, \sigma \in \mathsf{Val} \setminus \{\mathtt{error}\} \\ \mathtt{error}\,, & \text{otherwise} \end{cases}$$

In the equations above, it can be seen that the condition may be omitted if there is only one case, and that one is allowed to write “otherwise” instead of “if true”.

An example for the use of fixed points is the semantics definition for **while** loops in $\mathrm{PL}_0^{\mathrm{R}}$. Besides the domains already introduced, some more are needed for this definition:

	Com	communication events
	$\mathsf{Com}^\omega = (\mathbb{N} \to \mathsf{Com}) \cup \mathsf{Com}^*$	traces
st	$\in \mathsf{State} = (\,\Sigma \times \mathsf{Input} \times \mathsf{Com}^*\,) \uplus \mathsf{Inv}$	states
inv	$\in \mathsf{Inv} = (\,\{\mathtt{stopped}, \mathtt{invalid}\} \times \mathsf{Input} \times \mathsf{Com}^*\,) \uplus \mathsf{Com}^\omega$	invalid states
in	$\in \mathsf{Input}$	input streams

On State, an ordering $\sqsubseteq$ is defined that makes it a cpo. Basically, this ordering extends the prefix ordering on Com^ω onto State.

The denotational semantics of $\mathrm{PL}_0^{\mathrm{R}}$ processes is a state transformation, i. e. a function from $[\mathsf{State} \longrightarrow \mathsf{State}]$ that satisfies the extra condition that it does not change the communication history of processes as recorded in their Com^* or Com^ω components. The set of these functions is denoted by StatTr. So we have the semantic function

$$\mathcal{C} : \mathit{SeqProc} \longrightarrow \mathsf{DenEnv} \longrightarrow \mathsf{StatTr}$$

The (single) clause defining the semantics of loops is the following (again, the condition is **true**):

$$\mathcal{C}[\![\mathtt{WHILE}(exp, sproc)]\!]\, \mathsf{env} \overset{\text{def}}{=} \mu\Phi_{exp,sproc,\mathsf{env}} \quad \text{where}$$

$$\Phi_{exp,sproc,\mathsf{env}} : \mathsf{StatTr} \longrightarrow \mathsf{StatTr}$$

is a transformation of state transformations defined by

$$\Phi_{exp,sproc,\mathsf{env}}(Tr)st \overset{\text{def}}{=} st \quad \text{for all } st \in \mathsf{Inv}$$

and for all $st =< \sigma, in, tr >\in \mathsf{State} \setminus \mathsf{Inv}$

$$\Phi_{exp,sproc,\mathsf{env}}(Tr)st \overset{\text{def}}{=} \begin{cases} st & \text{, if } \mathcal{E}[\![exp]\!]\, \mathsf{env}\, \sigma = \mathsf{ff} \\ < \mathtt{invalid}, in, tr > & \text{, if } \mathcal{E}[\![exp]\!]\, \mathsf{env}\, \sigma \notin \mathsf{Bool} \\ (Tr \circ \mathcal{C}[\![sproc]\!]\, \mathsf{env})\, st & \text{, otherwise} \end{cases}$$

Of course, it has to be proved that $\Phi_{exp,sproc,\mathsf{env}}$ is a continuous function (i. e. an element of $[\mathsf{StatTr} \longrightarrow \mathsf{StatTr}]$).

Chapter 7

Modelling of SOS Definitions

This chapter formally describes the transformation of SOS deduction rules of the *ptp/t* format introduced in Def. 5.3 into term rewriting rules. It is one of the core chapters of this work, showing how to combine the $\lambda\sigma$-calculus modelling of **let** terms (cf. Section 3.1.3) with the new concept of contexts (cf. Section 3.3) in order to overcome the differences in restrictions on variables between the *ptp/t* format and the usual rewrite rule format of Def. 3.18.

The first section of this chapter gives an informal introduction to the idea behind the transformation of SOS definitions into rewrite rules; the full formal definition of the the transformation algorithm follows in the second section. The rewrite systems resulting from the algorithm enjoy rather pleasant simulation properties; these are listed in the third section, and a formal proof is contained in Appendix B. Finally, the *ptp/t* algorithm is compared with an algorithm that has been published before for the transformation of SOS systems in another format (GSOS) into rewrite rules.

The terms occurring in this chapter are written in standard mathematical notation rather than using the representations of Chapter 4. In this way, readability is enhanced and the terms become smaller. In implementations, however, the representations have to be used; so is sometimes necessary to refer to Chapter 4 in order to describe syntactic peculiarities.

7.1 The Basic Idea

The first section provides motivation and an informal description of the transformation method; it also shows why some other seemingly "obvious" methods do not work. The exact definition of this transformation follows in the next section.

7.1.1 The Example Language Definition

As an example (artificial, but not overly simple) consider an extract from an imperative language L. In L, there is a syntactic class of statements, denoting state transformations, and the usual operator ";" for sequential composition:

$$\mathit{Stmt} \ni \mathit{stmt} \ ::= \ \mathit{stmt}_1 ; \mathit{stmt}_2 \mid \ldots$$

On the semantic side, there is a set of states Σ that statements can be applied to. The internal structure of states $\sigma \in \Sigma$ is not important here. A configuration can either consist of a statement to be executed together with an initial state for this execution, or it can be the final state of an execution:

$$\begin{aligned} \Gamma_{Stmt} &= (\, Stmt \times \Sigma \,) \cup T_{Stmt} \\ T_{Stmt} &= \Sigma \end{aligned}$$

The execution of a statement list proceeds from left to right. After one computation step, the first statement in a list may have terminated, resulting in a final state, or there may still be a rest of the statement waiting for execution. For these two possibilities, there are the following two inference rule schemata:

$$\frac{\vdash \langle\, stmt_1, \sigma \,\rangle \rightarrow_{Stmt} \sigma_1}{\vdash \langle\, stmt_1; stmt_2, \sigma \,\rangle \rightarrow_{Stmt} \langle\, stmt_2, \sigma_1 \,\rangle} \tag{7.1}$$

for the case of termination of $stmt_1$ and

$$\frac{\vdash \langle\, stmt_1, \sigma \,\rangle \rightarrow_{Stmt} \langle\, stmt'_1, \sigma_1 \,\rangle}{\vdash \langle\, stmt_1; stmt_2, \sigma \,\rangle \rightarrow_{Stmt} \langle\, stmt'_1; stmt_2, \sigma_1 \,\rangle} \tag{7.2}$$

for the other case. $stmt'_1$ is the remainder of $stmt_1$ after one computation step.

7.1.2 Transformation Into Rewrite Rules

The simplest possible approach to the problem of transforming a rule

$$\frac{\vdash hypo}{\vdash \gamma \rightarrow \gamma'}$$

into a rewrite rule is to simulate the rule's semantics ("if *hypo* holds, then the step from γ to γ' is possible") in form of a simple conditional rewrite rule like

$$hypo \Rightarrow \gamma \rightarrow \gamma'$$

or, if only unconditional rules are allowed,

$$\gamma \rightarrow \textsf{if } hypo \textsf{ then } \gamma' \textsf{ else } \gamma''$$

where γ'' has to be defined appropriately. But with the format of Def. 3.18, this is only possible if there are no extra variables in *hypo*. This, however, is an exceptional case, since the extra variables in SOS rules are used as names for the results of intermediate computations; e. g. σ_1 in (7.1) and $stmt'_1$ in (7.2) both are extra variables.

So one has to be a little bit more inventive and has to find a way of disposing of the extra variables. Consider rule (7.1). The extra variable σ_1 stands for a terminal configuration that is related to $\langle\, stmt_1, \sigma \,\rangle$ by the transition relation. Viewing this relation more operationally, one can rephrase this as σ_1 standing for a possible (one-step) *result* of evaluating $\langle\, stmt_1, \sigma \,\rangle$.[1] The name σ_1 itself is irrelevant; it is only important that it denotes a terminal configuration.

[1]There may be more than one possible result if the language is non-deterministic.

The configuration $\langle\, stmt_1, \sigma \,\rangle$ does not contain extra variables; so it may safely occur on the right-hand side of a rewrite. Since only its result is important, it is enclosed by an additional operator *eval* that is intended to yield the result of evaluating its argument. By using let terms, i. e. λ-abstraction, this result can be named, and this name can again be σ_1. So one arrives at the rewrite rule

$$\langle\, stmt_1; stmt_2, \sigma \,\rangle \twoheadrightarrow \mathsf{let}\ \sigma_1 = eval\,(\,\langle\, stmt_1, \sigma \,\rangle\,)\ \mathsf{in}\ \langle\, stmt_2, \sigma_1 \,\rangle \quad . \tag{7.3}$$

Note that σ_1, although not appearing on the left-hand side, is not an extra variable. It is a bound variable of λ-calculus and, as much as term rewriting is concerned, it is just a constant of type $T_{Stmt}{}'$. In Section 3.2, such bound variables were eliminated completely by using de Bruijn indices. It will always be assumed that let terms are evaluated in applicative order (in *call by value* fashion).

So far, this looks like the kind of rule that was desired. But there still remains a problem. There is the other rule (7.2), and when the transformation procedure from above is applied to this rule, this results in the rewrite rule

$$\langle\, stmt_1; stmt_2, \sigma \,\rangle \twoheadrightarrow \mathsf{let}\ cf = eval\,(\,\langle\, stmt_1, \sigma \,\rangle\,)\ \mathsf{in}\ \langle\, cf \downarrow 1; stmt_2, cf \downarrow 2 \,\rangle \tag{7.4}$$

where cf is a variable of type $Stmt \times \Sigma$ and $\downarrow 1$ and $\downarrow 2$ are the projections to the first and second component of a configuration tuple, respectively. The problem is that the left-hand sides of (7.3) and (7.4) are identical, and so each of the two rules can be applied in any case where the other could be applied as well. In (7.1) and (7.2), the decision which rule to apply is made in the hypothesis by means of a type check.

It is specified by means of a term pattern where extra variables may be replaced by other terms of their type, and all other parts are considered as constants. This also includes the variables that are not extra because they are already instantiated to some constants by matching the left-hand side of the conclusion (see (1) in Def. 5.5).

In order to get correct rewrite rules, this kind of check must be added as well. So it must be tested whether the result of evaluating $\langle\, stmt_1, \sigma \,\rangle$ is terminal or not. And for the case that the result is non-terminal even though the rule derived from (7.1) was chosen, a way back must be provided giving a result that still allows application of the other rule. Speaking about the rewriting process in terms of traffic: choosing the wrong rule must be a "detour" rather than a "cul-de-sac".

Implementing the type check is simple: For (7.1), it amounts to having a rule of the form

$$\begin{array}{l} \langle\, stmt_1; stmt_2, \sigma \,\rangle \twoheadrightarrow \\ \quad \mathsf{let}\ \sigma_1 = eval\,(\,\langle\, stmt_1, \sigma \,\rangle\,)\ \mathsf{in}\ \mathsf{if}\ type\,(\sigma_1) = T_{Stmt}\ \mathsf{then}\ \langle\, stmt_2, \sigma_1 \,\rangle\ \mathsf{else}\ \ldots \end{array} \tag{7.5}$$

and for (7.2) to

$$\begin{array}{l} \langle\, stmt_1; stmt_2, \sigma \,\rangle \twoheadrightarrow \\ \quad \mathsf{let}\ cf = eval\,(\,\langle\, stmt_1, \sigma \,\rangle\,)\ \mathsf{in} \\ \quad\quad \mathsf{if}\ type\,(cf) = Stmt \times \Sigma\ \mathsf{then}\ \langle\, cf \downarrow 1; stmt_2, cf \downarrow 2 \,\rangle\ \mathsf{else}\ \ldots \end{array} \tag{7.6}$$

Of course this requires e. g. $\mathit{Stmt} \times T$ to be a term of the term language $T(\Sigma, V)$. As the basic rewrite system $\mathcal{B}$ is required to decide all conditions (see Section 4.3), one may assume that there are rules that rewrite these type checks to true or to false.

More problematic is the "way back" that must be placed in the else parts of (7.5) and (7.6). Intuitively, one would demand that in these cases, the original configuration $\langle\, \mathit{stmt}_1; \mathit{stmt}_2, \sigma \,\rangle$ should remain unchanged. But one cannot simply put this into the else parts since it would render the rewrite system non-terminating: If the type check failed, the same rule could be applied over and over again.

A straightforward solution for this problem is to provide a flag for each of the rules generated from the SOS rules that can be raised when the else part is selected. This flag then indicates that a rewrite rule has been tried in vain (i. e. its type check has been rewritten to false). This, however, is exactly the kind of situation that the concept of contexts has been defined for (see Section 3.3).

In this small example, there are only two rules. Hence it suffices to introduce contexts as elements of

$$\{0, 1, *\} \times \{\mathsf{on}, \mathsf{off}\}^2$$

and the desired rewrite rules become (for the one-step case)

$$\begin{array}{l} \langle\, \mathit{stmt}_1; \mathit{stmt}_2, \sigma \,\rangle \,@\, \langle\, 1, \mathsf{on}, s \,\rangle \twoheadrightarrow \\ \quad \mathsf{let}\ \ \sigma_1 = \mathit{eval}\,(\,\langle\, \mathit{stmt}_1, \sigma \,\rangle \,@\, \langle\, 1, \mathsf{on}, \mathsf{on} \,\rangle\,) \\ \quad \mathsf{in\ if}\ \ \mathit{type}\,(\mathit{cf}) = T_{\mathit{Stmt}} \\ \qquad\quad \mathsf{then}\ \ \langle\, \mathit{stmt}_2, \sigma_1 \,\rangle \,@\, \langle\, 1, \mathsf{on}, \mathsf{on} \,\rangle \\ \qquad\quad \mathsf{else}\ \ \langle\, \mathit{stmt}_1; \mathit{stmt}_2, \sigma \,\rangle \,@\, \langle\, 1, \mathsf{off}, s \,\rangle \end{array} \tag{7.7}$$

$$\begin{array}{l} \langle\, \mathit{stmt}_1; \mathit{stmt}_2, \sigma \,\rangle \,@\, \langle\, 1, s, \mathsf{on} \,\rangle \twoheadrightarrow \\ \quad \mathsf{let}\ \ \mathit{cf} = \mathit{eval}\,(\,\langle\, \mathit{stmt}_1, \sigma \,\rangle \,@\, \langle\, 1, \mathsf{on}, \mathsf{on} \,\rangle\,) \\ \quad \mathsf{in\ if}\ \ \mathit{type}\,(\mathit{cf}) = \mathit{Stmt} \times \Sigma \\ \qquad\quad \mathsf{then}\ \ \langle\, \mathit{cf} \downarrow 1; \mathit{stmt}_2, \mathit{cf} \downarrow 2 \,\rangle \,@\, \langle\, 1, \mathsf{on}, \mathsf{on} \,\rangle \\ \qquad\quad \mathsf{else}\ \ \langle\, \mathit{stmt}_1; \mathit{stmt}_2, \sigma \,\rangle \,@\, \langle\, 1, s, \mathsf{off} \,\rangle \end{array} \tag{7.8}$$

The rules that define the operator *eval* guarantee that its argument is evaluated appropriately (see Section 3.3.1); so $\langle\, \mathit{stmt}_1, \sigma \,\rangle$ is evaluated in one step only. For this evaluation, all rules may be used since no attempt to evaluate $\langle\, \mathit{stmt}_1, \sigma \,\rangle$ has been made so far. Furthermore, one can easily see that the else parts are smaller than the left-hand sides (under the well-founded ordering $\mathsf{off} < \mathsf{on}$); there is no termination problem when the type check fails. So (7.7) and (7.8) are really rules of a suitable kind. In the following, they will be called **SOS-derived rules**.

In this simple example, the two rules could be merged into one since the starting configurations in the hypotheses of (7.1) and (7.2) are the same. This results in identical *eval* expressions in the respective let terms, and so it would be possible to combine the two rules; this also means that only one flag in the context is needed:

$$\begin{array}{l}
\langle\, stmt_1; stmt_2, \sigma \,\rangle \,@\, \langle\, 1, \mathsf{on} \,\rangle \twoheadrightarrow \\
\quad \mathsf{let}\ \ cf = eval\,(\,\langle\, stmt_1, \sigma \,\rangle \,@\, \langle\, 1, \mathsf{on} \rangle\,) \\
\quad \mathsf{in\ if}\ \ type\,(cf) = T_{Stmt} \\
\qquad\quad \mathsf{then}\ \ \langle\, stmt_2, cf \,\rangle \,@\, \langle\, 1, \mathsf{on} \rangle \\
\qquad\quad \mathsf{else}\ \ \mathsf{if}\ \ type\,(cf) = Stmt \times \Sigma \\
\qquad\qquad\qquad \mathsf{then}\ \ \langle\, cf \downarrow 1; stmt_2, cf \downarrow 2 \,\rangle \,@\, \langle\, 1, \mathsf{on} \rangle \\
\qquad\qquad\qquad \mathsf{else}\ \ \langle\, stmt_1; stmt_2, \sigma \,\rangle \,@\, \langle\, 1, \mathsf{off} \,\rangle
\end{array} \tag{7.9}$$

In general, however, this merging is not possible, since the transitions in the hypotheses are not necessarily the same. Therefore the procedure that is described in the next section only considers one SOS rule at a time without trying this kind of optimization.[2]

What remains is to mention how to handle hypotheses containing simple Boolean conditions. The preconditions b_i can safely be used as the conditions of an **if** term surrounding the whole right-hand side since they do not contain extra variables. The **then** part of this **if** is the old right-hand side, and the **else** part is the left-hand side where the switch for this rule is set to **off**. The conditions B_k restricting the intermediate and final configurations must also become part of the type check. The extra variables must be replaced by suitable selection expressions in the style that has been used in rule (7.8). And finally, multiple transitions in the hypothesis are translated into iterated **let** expressions.

7.2 The Rewrite Rules Generated From an SOS System

In the following, assume a *ptp/t* SOS deduction system consisting of N rules for some $N \in \mathbb{N}$. Assume these rules are numbered $R_1, \ldots, R_N$, and consider R_l for some $l \in [N]$:

$$R_l : \quad \frac{\vdash \bigwedge_{i=1}^{p} b_i \ \wedge\ \bigwedge_{j=1}^{n} \gamma_j \xrightarrow{L_j} \gamma_j' \ \wedge\ \bigwedge_{k=1}^{q} B_k}{\vdash \overline{\gamma} \twoheadrightarrow \overline{\gamma}\,'}$$

In this case, contexts are $(N+1)$-tuples from the set

$$\{0, 1, *\} \times Flag$$

where $Flag =_{df} \{\mathsf{on}, \mathsf{off}\}^N$ is the set of rule switches.

[2]The improvement that is gained by merging the rules lies in the reduction of the number of rules and of the length of contexts. The terms occurring in the combined rules, however, become considerably larger.

Let $s_1, \ldots, s_N$ be fixed variables for flags.[3] The following abbreviations for special contexts (or rather, special context schemata) will be used[4]:

$$\begin{aligned}
Kv_1^{(l)} &=_{df} \ulcorner\langle 1, s_1, \ldots, s_{l-1}, \mathsf{on}, s_{l+1}, \ldots, s_N \rangle\urcorner \\
\overline{Kv}_1^{(l)} &=_{df} \ulcorner\langle 1, s_1, \ldots, s_{l-1}, \mathsf{off}, s_{l+1}, \ldots, s_N \rangle\urcorner \\
Kv^{(l)} &=_{df} \ulcorner\langle *, s_1, \ldots, s_{l-1}, \mathsf{on}, s_{l+1}, \ldots, s_N \rangle\urcorner \\
\overline{Kv}^{(l)} &=_{df} \ulcorner\langle *, s_1, \ldots, s_{l-1}, \mathsf{off}, s_{l+1}, \ldots, s_N \rangle\urcorner \\
K_f &=_{df} \ulcorner\langle *, \mathsf{on}, \ldots, \mathsf{on} \rangle\urcorner \\
K_{0f} &=_{df} \ulcorner\langle 0, \mathsf{on}, \ldots, \mathsf{on} \rangle\urcorner \\
K_{1f} &=_{df} \ulcorner\langle 1, \mathsf{on}, \ldots, \mathsf{on} \rangle\urcorner
\end{aligned} \tag{7.10}$$

Two rules will be generated for R_l, one for a 1-context $Kv_1^{(l)}$ and one for an $*$-context $Kv^{(l)}$.

7.2.1 First Case: No Transitions In the Premise

Definition 7.1

If $n = 0$, then also $q = 0$, and there are no extra variables in the rule; hence the derived rules can be rather simple:

$$\overline{\gamma} @ Kv_1^{(l)} \twoheadrightarrow \mathsf{if} \bigwedge_{i=1}^{p} b_i \mathsf{\ then\ } \overline{\gamma}' @ K_{0f} \mathsf{\ else\ } \overline{\gamma} @ \overline{Kv}_1^{(l)} \tag{7.11}$$

$$\overline{\gamma} @ Kv^{(l)} \twoheadrightarrow \mathsf{if} \bigwedge_{i=1}^{p} b_i \mathsf{\ then\ } \overline{\gamma}' @ K_f \mathsf{\ else\ } \overline{\gamma} @ \overline{Kv}^{(l)} \tag{7.12}$$

7.2.2 Second Case: At Least One Transition In the Premise

In this case, there may be extra variables in the SOS rule that have to taken care of. As in the example of Section 7.1, the following items must be considered:

(1) The type check of a configuration against a term pattern must be implemented.

(2) Extra variables must be substituted by appropriate terms.

(3) For extra variables occurring multiply, the transformation must maintain this correspondence for the terms that substitute the extra variables.

7.2.2.1 The Type Check

Reconsider the rule (7.2) from the example above:

$$\frac{\vdash \langle\, stmt_1, \sigma \,\rangle \twoheadrightarrow_{Stmt} \langle\, stmt_1', \sigma_1 \,\rangle}{\vdash \langle\, stmt_1; stmt_2, \sigma \,\rangle \twoheadrightarrow_{Stmt} \langle\, stmt_1'; stmt_2, \sigma_1 \,\rangle}$$

[3]The s_i are *meta-level variables* for elements of V_{Flag}, i. e. for *object-level variables* of sort *Flag*.

[4]If an abbreviation contains a "v" (like $Kv_1^{(l)}$), this indicates that the corresponding term contains variables (here the s_i).

In order to be able to describe what it means that $\ulcorner\langle\, stmt_1, \sigma \,\rangle\urcorner$ really evaluates to a configuration γ which is "of the form $\ulcorner\langle\, stmt_1', \sigma_1 \,\rangle\urcorner$", first the structure of configurations has to be made more precise. In the example, they were defined by

$$\begin{aligned} \Gamma_{Stmt} &= Stmt \times \Sigma \cup T_{Stmt} \\ T_{Stmt} &= \Sigma \end{aligned}$$

With the representation technique of Section 4.2.1, this means that configurations can either be pairs of the form $\ulcorner\langle\, stmt, \sigma \,\rangle\urcorner$, or just states that have to be injected into the configuration domain, however, i. e. terms of the form $\ulcorner\langle\, \sigma \,\rangle\urcorner$ (for sake of brevity, let $\langle \cdot \rangle$ denote this injection). So in both cases the structure is given by

- the top-level type "configuration";
- the number of sub-components; these numbers and the types of these components vary in the two cases.

This means that it must be checked that

- γ is of the sort of $\ulcorner\langle\, stmt_1', \sigma_1 \,\rangle\urcorner$, i. e. of the sort "configuration",
- γ and $\ulcorner\langle\, stmt_1', \sigma_1 \,\rangle\urcorner$ have the same number of sub-components, and if so, that
- the corresponding components match:
  (1) $\gamma \downarrow 1$ and $stmt_1'$ are of the same sort, and
  (2) $\gamma \downarrow 2$ and σ_1 are of the same sort.

If the term pattern were deeper structured than $\ulcorner\langle\, stmt_1', \sigma_1 \,\rangle\urcorner$, then the deeper subterms would have to be checked in the same way. So in general the type check is described by a recursively defined function yielding a predicate term. Note, however, that no recursion is needed if the term pattern does not contain extra variables. Only these define a true pattern that can be replaced by some other term; terms without extra variables just define the trivial pattern that may only be replaced by itself. So type checking against such a term can simply be expressed by checking equality to the term.

By Def. 5.3, there are no other possibilities for the patterns than variables, constants, and tuple constructor terms (with some mk- operator outermost). Therefore the example can be generalized in the following way:

Definition 7.2 (*type-ok*)
The function ***type-ok*** $: T(\Sigma, V) \times T(\Sigma, V) \times \mathcal{P}(V) \to T(\Sigma, V)$ is defined recursively by

$$type\text{-}ok(x, t, W) =_{df}$$

$$\begin{cases} \ulcorner x = t \urcorner & \text{, if } var(t) \subseteq W \\ \ulcorner \mathsf{type}(x) = {}_{\llcorner} type(t) {}_{\lrcorner} \urcorner & \text{, otherwise, if } t \text{ is not a tuple term} \\ \lceil \mathsf{type}(x) = {}_{\llcorner} type(t) {}_{\lrcorner} \wedge \#\mathsf{comp}(x) = {}_{\llcorner}\#comp(t){}_{\lrcorner} \wedge & \\ \quad \displaystyle\bigwedge_{i=1}^{{}_{\llcorner}\#comp(t){}_{\lrcorner}} {}_{\llcorner} type\text{-}ok(\ulcorner x \downarrow i \urcorner, t/i, W) {}_{\lrcorner} \rceil & \\ & \text{, otherwise} \end{cases}$$

The first term x in this definition is the term to be type-checked, the second term t is the pattern, and W is the set of variables that are not extra variables. Note how quasi-quotes and inverse quotes are used to separate terms that have to be evaluated on meta-level (e. g. t/i) from others that remain fixed (e. g. $x \downarrow i$). The big conjunction in the last clause is just an abbreviation, and not a term constructor.

Example 7.3

Let $v \in V_T, y \in V_U$ and $\langle \cdot, \cdot \rangle : T, U \to C$ be a tuple constructor. Then

$$\begin{aligned} & \mathit{type\text{-}ok}(x, \langle\, v, y \,\rangle, \{y\}) \\ & \quad = \ulcorner \mathsf{type}(x) = \mathsf{\_C} \ \wedge\ \#\mathsf{comp}(x) = 2 \wedge \\ & \qquad \llcorner \mathit{type\text{-}ok}(\ulcorner x \downarrow 1 \urcorner, v, \{y\}) \lrcorner \ \wedge\ \llcorner \mathit{type\text{-}ok}(\ulcorner x \downarrow 2 \urcorner, y, \{y\}) \lrcorner \urcorner \\ & \quad = \ulcorner \mathsf{type}(x) = \mathsf{\_C} \ \wedge\ \#\mathsf{comp}(x) = 2 \ \wedge\ \mathsf{type}(x \downarrow 1) = \mathsf{\_}\llcorner \mathit{type}\,(v) \lrcorner \ \wedge\ x \downarrow 2 = y \urcorner \\ & \quad = \ulcorner \mathsf{type}(x) = \mathsf{\_C} \ \wedge\ \#\mathsf{comp}(x) = 2 \ \wedge\ \mathsf{type}(x \downarrow 1) = \mathsf{\_T} \ \wedge\ x \downarrow 2 = y \urcorner \end{aligned}$$

□

7.2.2.2 Positions of Extra Variables

In the example of Section 7.1, the extra variables have been replaced by terms constructed from de Bruijn indices and applications of selector functions: $\ulcorner \langle\, \mathit{stmt}'_1; \mathit{stmt}_2, \sigma_1 \,\rangle \urcorner$ in (7.2), for example, becomes $\ulcorner \langle\, \mathit{cf} \downarrow 1; \mathit{stmt}_2, \mathit{cf} \downarrow 2 \,\rangle \urcorner$ in (7.8), where cf is the representation of a de Bruijn index bound in a let expression (see Section 3.2).

The sequences of selector applications that may be used can be deduced from the position of the extra variables in the γ'_j. For an extra variable v, let $\boldsymbol{j_v} \in [n]$ be the index of a transition in the hypothesis of R_l such that $v \in \mathit{var}\,(\gamma'_{j_v})$.[5] Since the γ'_j are constructed only from variables, constants, and tuple-forming operators (see condition (3) in Def. 5.3), v can be retrieved from γ'_{j_v} by subsequently selecting components. This path can be represented by a sequence $\boldsymbol{w_v}$ of projection operators $\downarrow i$ for appropriate $i \in \mathbb{N}$. So

$$w_v \in (\ \{\downarrow\}\mathbb{N}\ )^* \quad \text{with} \quad \ulcorner \gamma'_{j_v} w_v \urcorner =_E v \quad .$$

Note that there is a choice for w_v if an extra variable occurs more than once; in this case, one fixed occurrence is chosen. Of course, the correspondence between the different occurrences has to be checked (this is done with the predicate term $\mathit{EV\text{-}match}$; see below).

Example 7.4

If v is an extra variable of R_l with $j_v = 2$ and $\gamma'_2 = \ulcorner \langle\, x, \langle\, v, y \,\rangle \,\rangle \urcorner$, then $w_v = \ulcorner \downarrow 2 \downarrow 1 \urcorner$ because

$$\gamma'_2 w_v = \ulcorner \langle\, x, \langle\, v, y \,\rangle \,\rangle \downarrow 2 \downarrow 1 \urcorner =_E v \quad .$$

□

[5]Such an index always exists because all extra variables of SOS rules must appear in these configuration terms by condition (8) in Def. 5.3.

7.2.2.3 The Structure of the Derived Rules

Definition 7.5

The structure of the rules for the one-step and the multi-step case is the same; they only differ in the contexts used:

$$
\begin{array}{l}
\overline{\gamma} \mathbin{@} Kv_1^{(l)} \longrightarrow \\
\quad \textsf{if} \quad \bigwedge\limits_{i=1}^{p} b_i \\
\qquad \textsf{then} \;\; \textsf{let} \;\; x_1 = \mathit{eval}\,(\, \gamma_1 \mathbin{@} \langle\, L_1, \textsf{on}, \ldots, \textsf{on} \,\rangle\,), \\
\qquad\qquad\qquad \ldots, \\
\qquad\qquad\qquad x_n = \mathit{eval}\,(\, \gamma_n \mathbin{@} \langle\, L_n, \textsf{on}, \ldots, \textsf{on} \,\rangle\,) \\
\qquad\qquad \textsf{in if} \;\; \bigwedge\limits_{i=1}^{n} \llcorner \mathit{type\text{-}ok}\,(x_i, \gamma_i', \mathit{var}\,(\overline{\gamma}))\lrcorner \wedge \bigwedge\limits_{i=1}^{q} \overline{B_i} \wedge \llcorner \mathit{EV\text{-}match} \lrcorner \\
\qquad\qquad\quad \textsf{then} \;\; \tilde{\gamma}\,' \mathbin{@} K_{0f} \\
\qquad\qquad\quad \textsf{else} \;\; \overline{\gamma} \mathbin{@} \overline{Kv_1}^{(l)} \\
\qquad \textsf{else} \;\; \overline{\gamma} \mathbin{@} \overline{Kv_1}^{(l)}
\end{array}
\tag{7.13}
$$

$$
\begin{array}{l}
\overline{\gamma} \mathbin{@} Kv^{(l)} \longrightarrow \\
\quad \textsf{if} \quad \bigwedge\limits_{i=1}^{p} b_i \\
\qquad \textsf{then} \;\; \textsf{let} \;\; x_1 = \mathit{eval}\,(\, \gamma_1 \mathbin{@} \langle\, L_1, \textsf{on}, \ldots, \textsf{on} \,\rangle\,), \\
\qquad\qquad\qquad \ldots, \\
\qquad\qquad\qquad x_n = \mathit{eval}\,(\, \gamma_n \mathbin{@} \langle\, L_n, \textsf{on}, \ldots, \textsf{on} \,\rangle\,) \\
\qquad\qquad \textsf{in if} \;\; \bigwedge\limits_{i=1}^{n} \llcorner \mathit{type\text{-}ok}\,(x_i, \gamma_i', \mathit{var}\,(\overline{\gamma}))\lrcorner \wedge \bigwedge\limits_{i=1}^{q} \overline{B_i} \wedge \llcorner \mathit{EV\text{-}match} \lrcorner \\
\qquad\qquad\quad \textsf{then} \;\; \tilde{\gamma}\,' \mathbin{@} K_{f} \\
\qquad\qquad\quad \textsf{else} \;\; \overline{\gamma} \mathbin{@} \overline{Kv}^{(l)} \\
\qquad \textsf{else} \;\; \overline{\gamma} \mathbin{@} \overline{Kv}^{(l)}
\end{array}
\tag{7.14}
$$

where:

(1) The x_i are distinct new identifiers. x_i plays the role of the de Bruijn index i.

(2) For $i \in [n] : \overline{B_i} =_{\mathit{df}} B_i[(x_{j_v})w_v/v \mid v \text{ is an extra variable in } B_i]$

(3) The predicate $\mathit{EV\text{-}match}$ checks that the same instantiations for v are inserted in every possible place:

$$\mathit{EV\text{-}match} =_{\mathit{df}} \bigwedge\{\ulcorner x_k w_v' = x_{j_v} w_v \urcorner \mid v \text{ is an extra variable occurring more than once in the } \gamma_i'\text{, and } \gamma_k' w_v' \text{ is an occurrence of } v \text{ different from } \gamma_{j_v}' w_v\}$$

(4) $\tilde{\gamma}\,' =_{\mathit{df}} \begin{cases} \overline{\gamma}\,' & \text{, if } \overline{\gamma}\,' \text{ does not contain extra variables} \\ \overline{\gamma}\,'[x_{j_v} w_v / v \mid v \text{ is an extra variable in } \overline{\gamma}\,'] & \text{, otherwise} \end{cases}$

Remarks:

(1) If γ'_j is a universal pattern that matches any configuration, e. g. because it is just an extra variable, then the type check for x_j can be omitted because it is always true (and hence, by completeness of $\mathcal{B}$, rewrites to **true**). If this holds for all $j \in [n]$, and if the B_i and *EV-match* parts are not present, either, the whole if _ then _ else _ part can by replaced by its then part.

(2) There is also the possibility of moving the condition $\bigwedge_{i=1}^{p} b_i$ into the type check condition, which would give simpler structured rules. On the other hand, rewriting with these simpler versions of the rules is less efficient because leading let clauses are rewritten before $\bigwedge_{i=1}^{p} b_i$ is checked even in those cases where this condition (which is independent of the let variables) is rewritten to false.

7.2.3 Examples

As examples, consider the rules (EO1) and (EO6) that we already encountered in Section 5.2.

(EO1) is an axiom schema:

$$\text{(EO1)} \qquad \rho \vdash_{\delta} \langle\, int, \sigma \,\rangle \longrightarrow_{Expr} \alpha\,(int)$$

The first step in transforming this rule is to include its additional parameters ρ and δ into the configurations:

$$\text{(EO1')} \qquad \vdash \langle\, \rho, \delta, \langle\, int, \sigma \,\rangle \,\rangle \longrightarrow_{Expr} \langle\, \rho, \delta, \alpha\,(int) \,\rangle$$

Now the transformation procedure can be applied: here, $n, p, q = 0$, and thus the following two rules are generated, assuming (EO1) is the l-th rule out of N:

$$\langle\, \rho, \delta, \langle\, int, \sigma \,\rangle \,\rangle \mathbin{@} Kv_1^{(l)} \to \langle\, \rho, \delta, \alpha\,(int) \,\rangle \mathbin{@} K_{0f}$$
$$\langle\, \rho, \delta, \langle\, int, \sigma \,\rangle \,\rangle \mathbin{@} Kv^{(l)} \to \langle\, \rho, \delta, \alpha\,(int) \,\rangle \mathbin{@} K_{f}$$

More interesting is the translation of (EO6). After inclusion of the environments ρ and δ into the configurations, it becomes

$$\text{(EO6')} \qquad \frac{\vdash \langle\, \rho, \delta, \langle\, exp, \sigma \,\rangle \,\rangle \longrightarrow_{Expr} \langle\, \rho, \delta, val \rangle \wedge val \neq \texttt{error}}{\vdash \langle\, \rho, \delta, \langle\, mop\ exp, \sigma \,\rangle \,\rangle \longrightarrow_{Expr} \langle\, \rho, \delta, \alpha(\, mop\,)\,(val) \,\rangle}$$

Here, $p = 0, n = q = 1, L_1 = 1$, and the rules generated look as follows (only presenting the one-step rule and assuming that (EO6) is the j-th rule):

```
$\langle\, \rho, \delta, \langle\, mop\ exp, \sigma \,\rangle \,\rangle \mathbin{@} Kv_1^{(j)} \to$
  let  $x_1 = eval\,(\,\langle\, \rho, \delta, \langle\, exp, \sigma \,\rangle\, \rangle K_{1f} \mathbin{@}\ )$
  in if  $type\,(x_1) = \text{\_ExtTExpr} \ \wedge\ \#\text{comp}(x_1) = 2 \ \wedge\ x_1 \downarrow 1 = \rho \ \wedge\ x_1 \downarrow 2 = \delta \ \wedge$
         $type\,(x_1 \downarrow 3) = \text{\_TExpr} \ \wedge\ x_1 \downarrow 3 \neq \texttt{error}$
       then  $\langle\, \rho, \delta, \alpha(\, mop\,)\,(x_1 \downarrow 3) \,\rangle \mathbin{@} K_{0f}$
       else  $\langle\, \rho, \delta, \langle\, mop\ exp, \sigma \,\rangle \,\rangle \mathbin{@} \overline{Kv}_1^{(j)}$
```

Remarks:

- _TExpr is the type constant standing for the sort representing T_{Expr}.
- _ExtTExpr is the type constant standing for the sort representing T_{Expr} extended by additional environment parameters.
- Since *val* occurs only once, *EV-match* is empty. Because $val =_E \ulcorner \langle \rho, \delta, val \rangle \downarrow 3 \urcorner$, $w_{val} = \ulcorner \downarrow 3 \urcorner$. This explains the occurrences of $x_1 \downarrow 3$ in the translation of B_1 ($\ulcorner val \neq \texttt{error} \urcorner$) and in the **then** part ($\ulcorner \tilde{\gamma}' \urcorner$).

7.2.4 A Transformation Algorithm for *ptp/t* Systems

The structure of the SOS-derived rewrite rules is obtained from the SOS deduction rules in a very regular manner. Therefore it was not too complicated to implement an algorithm that transforms a complete SOS definition into a simulating rewrite system.

In principle, each of the SOS rules can be transformed into rewrite rules individually. One only has to know the relative position of the SOS rule in the list of all rules in order to generate the correct contexts. So the algorithm simply consists of performing the procedure of Sections 7.2.1 and 7.2.2 (depending on whether there are transitions in the premise or not) to each of the rules of an SOS definition.

A C program (developed with `lex` and `yacc`) that implements this algorithm for practical applications is described in Appendix A.2.

7.3 Properties of the Transformed System

7.3.1 Simulation

The transformation procedure has been devised in order to produce a rewriting system $\mathcal{R}$ that models an SOS semantics definition $\mathcal{S}$ as closely as possible, the characteristic feature being the set of possible transition sequences. Therefore the most interesting questions to ask about $\mathcal{R}$ are what rewriting sequences are possible and how they are related to the transition sequences of $\mathcal{S}$. It can be shown that the relation between $\mathcal{R}$ and $\mathcal{S}$ is indeed very close; if rewriting is considered modulo the equational theory $\mathcal{E}$ of the basic system $\mathcal{B}$, there is a 1-1 correspondence between rewriting sequences and flattened transition sequences (where steps performed in order to establish a premise also contribute to the visual steps).

In the proofs, the **general assumption** from Section 4.3 will be needed that $\mathcal{B}$ provides (in the logical sense) a correct and complete decision procedure for all conditions that do not depend on the semantics definition. This means that each term expressing such a condition has exactly one $\mathcal{B}$-normal form, viz. either **true** or **false**.

Let $\mathcal{S}$ be defined by the transition system $(\Gamma, T, \rightarrow_{\mathcal{S}})$, and let $\rightarrow_{\mathcal{S}}$ be given by a set of $N \in \mathbb{N}$ deduction rules. Let the rewrite system $\boldsymbol{\mathcal{R}'}$ be the system of SOS-derived rules, and let $\boldsymbol{\mathcal{R}}$ $=_{df} \mathcal{R}' \uplus \mathcal{B}$.

The following additional abbreviations for contexts will be useful, where $k \in [N]$ and $r_j \in \{\mathsf{on}, \mathsf{off}\}$ for $j \in [N]$:

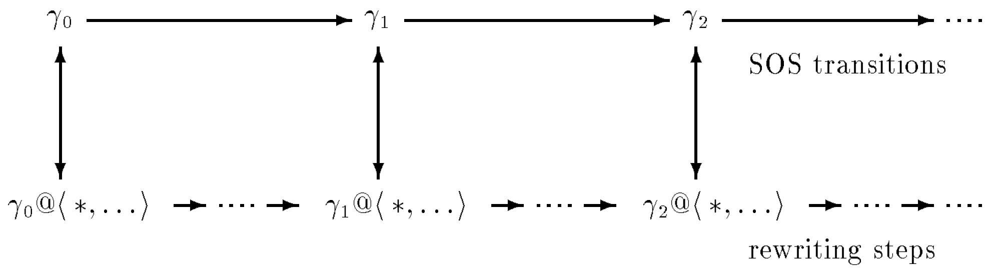
$$
\begin{array}{l}
K_1^{(k)} =_{df} \ulcorner\langle\, 1, r_1, \ldots, r_{k-1}, \mathsf{on}, r_{k+1}, \ldots, r_N \,\rangle\urcorner \\
\overline{K}_1^{(k)} =_{df} \ulcorner\langle\, 1, r_1, \ldots, r_{k-1}, \mathsf{off}, r_{k+1}, \ldots, r_N \,\rangle\urcorner \\
K^{(k)} =_{df} \ulcorner\langle\, *, r_1, \ldots, r_{k-1}, \mathsf{on}, r_{k+1}, \ldots, r_N \,\rangle\urcorner \\
\overline{K}^{(k)} =_{df} \ulcorner\langle\, *, r_1, \ldots, r_{k-1}, \mathsf{off}, r_{k+1}, \ldots, r_N \,\rangle\urcorner
\end{array}
\qquad (7.15)
$$

Note that these contexts are ground terms (i. e. contain no variables), whereas the abbreviations (7.10) from Section 7.2 contain variables; the r_j used here are meta variables for ground terms, not variables for the rewriting process.

7.3.1.1 Overview

The simulation of $\mathcal{S}$ by $\mathcal{R}$ can be described as in Fig. 7.1. The intermediate terms in the rewriting sequence result from applying rules from $\mathcal{R}$ to configuration terms. They need not themselves be configuration terms, but they are equal to such a term modulo $=_E$.

Figure 7.1: "1:n" simulation of transition sequences



$$\begin{tikzcd}
\gamma_0 \arrow[rr] \arrow[dd, leftrightarrow] & & \gamma_1 \arrow[rr] \arrow[dd, leftrightarrow] & & \gamma_2 \arrow[rr] \arrow[dd, leftrightarrow] & & \cdots\cdot \\
& & & & & \text{SOS transitions} & \\
\gamma_0@\langle\, *, \ldots\rangle \arrow[r] & \cdots\cdot \arrow[r] & \gamma_1@\langle\, *, \ldots\rangle \arrow[r] & \cdots\cdot \arrow[r] & \gamma_2@\langle\, *, \ldots\rangle \arrow[r] & \cdots\cdot \arrow[r] & \cdots\cdot \\
& & & & & \text{rewriting steps} &
\end{tikzcd}$$

Each transition step is modelled by a rewriting sequence in $\mathcal{R}$ which generally has more than just one step. In order to be allowed to perform one step $\gamma_1 \rightarrow_{\mathcal{S}} \gamma_2$ using the transition system, there usually have to be performed a number of transitions that correspond to the premises of transition rules. These "hidden" transitions only contribute indirectly to $\gamma_1 \rightarrow_{\mathcal{S}} \gamma_2$ by determining parts of γ_2. So the transition process is not organized in linear form; each transition is equipped with a tree of other transitions (a proof tree) that justifies it. The corresponding rewriting process, however, can only construct flat sequences of terms. Therefore all the hidden transitions become part of the simulating sequence $\gamma_1 @ K_f \xrightarrow{*}_{\mathcal{R}} \gamma_2 @ K_f$ as well. Furthermore, rewriting has to make explicit use of the rules in $\mathcal{B}$, while SOS transitions take place modulo $=_E$.

Simulation works in the other direction as well. If a rewriting sequence uses one SOS-derived rule, then this sequence corresponds to a transition sequence that is obtained via this particular SOS rule.

For the simulation of transitions, one has to bear in mind that in $\gamma \rightarrow_{\mathcal{S}} \gamma'$ both source and target may be freely modified as long as the resulting configurations γ_1 and γ_1' stay E-equal to γ resp. γ' (see Def. 5.5). In term rewriting, however, matching is done in an exact way and not modulo

$=_E$, and the steps from γ_1 to γ and from γ_1' to γ' may not be possible with $\rightarrow_{\mathcal{B}}$ because they go into the wrong direction (from simple to more complex terms). Therefore, only exact transitions $\gamma \rightarrow_{\mathcal{S}}^{=} \gamma'$ can be simulated (see Def. 5.8). But this is not a real restriction; by Lemma 5.9, a transition $\gamma \rightarrow_{\mathcal{S}} \gamma'$ can always be split into $\gamma =_E \tilde{\gamma} \rightarrow_{\mathcal{S}}^{=} \tilde{\gamma}' =_E \gamma'$ for some $\tilde{\gamma}, \tilde{\gamma}' \in \Gamma'$, and then the exact transition can be simulated.

The remainder of this section provides a formal description of this simulation.

7.3.1.2 One-step simulation

The basic building stone for the simulation in Fig. 7.1 is the simulation of one transition step by a rewriting sequence. It is expressed formally in the following way:

One-step completeness

$$\begin{array}{l}\forall \gamma, \gamma' \in \Gamma\ \forall k \in [N]\ \forall r_1, \ldots, r_{k-1}, r_{k+1}, \ldots, r_N \in \{\mathsf{on}, \mathsf{off}\} : \\ \quad \text{if } \gamma \rightarrow_{\mathcal{S}}^{=} \gamma' \text{ using rule } k \text{ then } \gamma\ @\ K_1^{(k)} \xrightarrow{+}_{\mathcal{R}} \gamma'\ @\ K_{0f}\end{array} \tag{7.16}$$

In the other direction, we have the following relation between rewriting sequences with just one application of an SOS-derived rule and a transition sequence:

One-step correctness

$$\begin{array}{l}\forall \gamma, \gamma' \in \Gamma\ \forall k \in [N]\ \forall r_1, \ldots, r_{k-1}, r_{k+1}, \ldots, r_N \in \{\mathsf{on}, \mathsf{off}\} : \\ \quad \text{if } \gamma\ @\ K_1^{(k)} \xrightarrow{+}_{\mathcal{R}} \gamma'\ @\ K_{0f} \text{ then } \gamma \rightarrow_{\mathcal{S}} \gamma'\end{array} \tag{7.17}$$

The names "correctness" and "completeness" are used in the logical sense: Any transition corresponds to a rewriting sequence (completeness), and any rewriting sequence that contains one outermost application of an SOS-derived rule corresponds to a transition step (correctness). Note how the 1-contexts $K_1^{(k)}$ restrict rewriting to just one transition-related step.

7.3.1.3 Normal-form simulation

Building up inductively from the one-step results, simulation properties for longer transition sequences may be obtained. One special case is of particular interest: sequences that end with a terminal configuration describe the complete evaluation of their initial configuration. Furthermore, expressions like $\gamma \xrightarrow{*}_{\mathcal{S}} t$ $(\gamma \in \Gamma, t \in T)$ may occur in the premises of SOS rules.

For transitions to normal forms, the following implications hold:

Normal form completeness and correctness

$$\forall \gamma \in \Gamma\ \forall t \in T : \gamma \xrightarrow{*}_{\mathcal{S}}^{=} t \quad \Rightarrow \quad \gamma\ @\ K_f \xrightarrow{*}_{\mathcal{R}} t\ @\ K_f \tag{7.18}$$

$$\forall \gamma \in \Gamma\ \forall t \in T : \gamma\ @\ K_f \xrightarrow{*}_{\mathcal{R}} t\ @\ K_f \quad \Rightarrow \quad \gamma \xrightarrow{*}_{\mathcal{S}} t \tag{7.19}$$

The proofs for these results can be found in Appendix B. Because one-step and normal form transitions are intertwined via transitions in the premises of rules, all results must be proved by one simultaneous induction (over the number of applications of SOS-derived rules).

7.3.1.4 Simulation of divergence

From one-step completeness, it can be immediately deduced that each infinite transition sequence corresponds to an infinite rewriting sequence. So non-termination is preserved by the rewriting system:

Divergence completeness

$$\begin{array}{l} \forall \{\gamma^{(i)}\} \in \Gamma^{\mathbb{N}} : \\ \quad (\forall i \in \mathbb{N} : \gamma^{(i)} \rightarrow^{=}_{\mathcal{S}} \gamma^{(i+1)}) \ \Rightarrow\ (\forall i \in \mathbb{N} : \gamma^{(i)} @ K_f \xrightarrow{*}_{\mathcal{R}} \gamma^{(i+1)} @ K_f) \end{array} \tag{7.20}$$

On the other hand, all infinite rewriting sequences correspond to infinite behaviours of the transition system. If the rewriting sequence keeps returning to configurations terms, i. e. each tail of the sequence contains a configuration-context pair, then one-step correctness yields the existence of a corresponding infinite transition sequence which can be found by taking the first components of a subsequence of the rewriting sequence:

Divergence correctness, preliminary version

$$\begin{array}{l} \forall \gamma \in \Gamma\ \forall \{t^{(i)}\} \in T(\Sigma, V)^{\mathbb{N}} : t^{(1)} = \gamma @ K_f \ \wedge\ (\forall i \in \mathbb{N} : t^{(i)} \rightarrow_{\mathcal{R}} t^{(i+1)}) \ \wedge \\ \quad (\forall i \in \mathbb{N}\ \exists j \in \mathbb{N}\ \exists \gamma' \in \Gamma : j > i \ \wedge\ t^{(j)} = \gamma' @ K_f) \qquad \Rightarrow \\ \exists \{\gamma^{(i)}\} \in \Gamma^{\mathbb{N}}\ \exists j : \mathbb{N} \rightarrow \mathbb{N} \text{ strictly monotonic} : \\ \quad (\forall i \in \mathbb{N} : \gamma^{(i)} = (t^{(j(i))} \downarrow 1) \ \wedge\ \gamma^{(i)} \rightarrow_{\mathcal{S}} \gamma^{(i+1)}) \end{array} \tag{7.21}$$

Now assume that there is a point in an infinite rewriting sequence from which on there are no more configuration-context terms. Then there also is a corresponding behaviour of the transition system, that is, a non-terminating attempt to prove a transition. Non-termination of this kind can only happen because there is some let term that is never β-reduced. But $\mathcal{B}$ is assumed to be terminating and the contexts prevent using the same rule unsuccessfully repeatedly; so this means that there is an infinite chain of attempts to evaluate configuration-contexts pairs using the SOS-derived rules. By construction of the rules, each such attempt corresponds to an attempt to prove a transition in the premise of a rule, and so there is the possibility of a non-terminating proof attempt with the transition system. But this violates the additional Requirement 5.10 in Section 5.2; therefore, such an infinite rewriting sequence cannot occur, and so the last result to may be simplified to:

Divergence correctness, final version

$$\begin{array}{l} \forall \gamma \in \Gamma\ \forall \{t^{(i)}\} \in T(\Sigma, V)^{\mathbb{N}} : t^{(1)} = \gamma @ K_f \ \wedge\ (\forall i \in \mathbb{N} : t^{(i)} \rightarrow_{\mathcal{R}} t^{(i+1)}) \ \Rightarrow \\ \exists \{\gamma^{(i)}\} \in \Gamma^{\mathbb{N}}\ \exists j : \mathbb{N} \rightarrow \mathbb{N} \text{ strictly monotonic} : \\ \quad (\forall i \in \mathbb{N} : \gamma^{(i)} = (t^{(j(i))} \downarrow 1) \ \wedge\ \gamma^{(i)} \rightarrow_{\mathcal{S}} \gamma^{(i+1)}) \end{array} \tag{7.22}$$

7.3.2 Properties Related to Term Rewriting

The system $\mathcal{R}$ consists of two parts: the basic system $\mathcal{B}$ and the system $\mathcal{R}'$ containing the SOS-derived rules. As already mentioned in Section 4.3, $\mathcal{B}$ is assumed to be TRS-complete, i. e. confluent and terminating, so there are no problems with this part. But for $\mathcal{R}'$, the situation is different because these properties are completely determined by the semantics of the language L.

As seen in the previous section, every rewriting sequence in $\mathcal{R}$ has a direct counterpart in $\mathcal{S}$ and vice versa. This has direct consequences for confluence and termination. Assume $\mathcal{R}$ is terminating. This means that there is no configuration-context pair that is the initial point for an infinite rewriting sequence. Therefore there is also no configuration that starts an infinite transition sequence in $\mathcal{S}$. Obviously, this property is equivalent to L being a language that only contains terminating programs.

For confluence, the situation is very similar. Consider rewriting modulo the equational theory $=_E$ generated by the basic system $\mathcal{B}$. Then the only rewrite rules needed are those derived from the SOS system. Confluence of this rewrite system means that every configuration has at most one normal form (modulo $=_E$). As a consequence, for each initial state and each program starting in this state, there is at most one final state, and hence the programming language must be deterministic.[6]

So typically $\mathcal{R}$ is not complete. In most cases, it will be non-terminating, and therefore normalization of configuration terms must be handled with care. Languages in the tradition of CSP (cf. Hoare [68]) and Occam ([75]) do not even lead to confluent systems since they contain non-deterministic choice operators. This might seem a serious drawback of the method, but it only reflects the desire to have a rewrite system that models the semantics as closely as possible. And the problem is very well known: Interpreters for functional languages, say, usually do not terminate (disregarding restrictions like finite stack size) when interpreting programs that are (semantically) "non-terminating".

There is also no point in completing the system $\mathcal{R}$, e. g. by applying the Knuth-Bendix procedure (cf. [81]). All completion algorithms assume that a rewrite rule $l \longrightarrow r$ is a "directed equation" and hence the interpretation of l and r are the same. But the rules from $\mathcal{R}'$ are different: if L is non-deterministic, there may be two rules in $\mathcal{R}'$ that apply to a given configuration and yield distinct results, exactly as in the SOS system. So there may be the situation

$$\gamma_1 \mathbin{@} K_1 \;{}_{\mathcal{R}}\!\xleftarrow{*} \gamma \mathbin{@} K \xrightarrow{*}_{\mathcal{R}} \gamma_2 \mathbin{@} K_2 \tag{7.23}$$

where γ_1 and γ_2 are (semantically) different configurations, and so $\gamma_1 = \gamma_2$ should not hold (neither should $\gamma_1 =_E \gamma_2$).

A natural interpretation of configurations is the set of evaluation sequences (or results thereof) starting in them. In (7.23), these interpretations of left- and right-hand sides are not the same; here the interpretation of the left-hand side is the union of all the interpretations of possible right-hand sides.

So $\rightarrow_{\mathcal{R}'}$ is not the directed version of an equality relation[7], and therefore completion does not make sense. Essentially, what it would do is to generate a rewrite system where all non-determinism has been artificially removed by declaring different possible results of programs as equal, and this system is certainly not consistent with the original SOS system. Moller [94] shows that to reach an equational description of systems incorporating non-determinism, additional operators have to be introduced in the language; for an example of such operators, see the next section.

[6]This requirement can be slightly weakened; e. g. the evaluation order of parameters for function calls is unimportant as long as this evaluation has no side effects.

[7]Quite in contrast to $\rightarrow_{\mathcal{B}}$!

7.4 Another Approach: Turning GSOS Rules into Equations

For the GSOS format of Def. 5.16, there exists a different approach to generate an equational theory from an SOS system. In [2], Aceto, Bloom and Vaandrager present an algorithm to transform a special class of GSOS systems into equations. Their work originated from the area of process algebra (cf. Baeten and Weijland [6] or Milner [93]), so they only consider languages of processes. All of these languages include the language FINTREE, containing the following process constructors:

- $\mathbf{0}$, the empty process;
- $\boldsymbol{a\,p}$, denoting action prefixing;
- $\boldsymbol{p + q}$, denoting non-deterministic choice

The algorithm takes a GSOS system fulfilling certain "sanity conditions" and produces a set of equations that can be used as rewrite rules. With these, any ground process term can be normalized to **head normal form**, i. e. to a sum of action prefixed terms.

Since neither the *ptp/t* format nor the GSOS format are generalizations of each other, an GSOS system without negative premises will be used as an example that can be handled by both the transformation algorithm of Aceto, Bloom and Vaandrager, and the algorithm of Section 7.2.

7.4.1 The Example Language

The example language $L_{||}$ extends FINTREE with one additional operator, viz. synchronous parallel composition $\cdot \,||\, \cdot$. The abstract syntax or such processes is given by:

$$Proc \ni p ::= \mathbf{0} \mid a\,p \mid p_1 + p_2 \mid p_1 \,||\, p_2$$

The set of actions includes an element c serving for synchronisation of parallel processes:

$$Act \ni a ::= c \mid \ldots$$

Ignoring the problem of termination of processes like $\mathbf{0} \,||\, \mathbf{0}$, the GSOS system defining the semantics of $L_{||}$ is the following:

$$\vdash a\,p \xrightarrow{a} p \tag{7.24}$$

$$\frac{\vdash p_1 \xrightarrow{a} p_1'}{\vdash p_1 + p_2 \xrightarrow{a} p_1'} \tag{7.25}$$

$$\frac{\vdash p_2 \xrightarrow{a} p_2'}{\vdash p_1 + p_2 \xrightarrow{a} p_2'} \tag{7.26}$$

$$\frac{\vdash p_1 \xrightarrow{a} p_1'}{\vdash p_1 \,||\, p_2 \xrightarrow{a} p_1' \,||\, p_2} \quad a \neq c \tag{7.27}$$

$$\frac{\vdash p_2 \xrightarrow{a} p_2'}{\vdash p_1 \,\|\, p_2 \xrightarrow{a} p_1 \,\|\, p_2'} \quad a \neq c \tag{7.28}$$

$$\frac{\vdash p_1 \xrightarrow{c} p_1', p_2 \xrightarrow{c} p_2'}{\vdash p_1 \,\|\, p_2 \xrightarrow{c} p_1' \,\|\, p_2'} \tag{7.29}$$

(7.24) to (7.26) form the definition of FINTREE, and hence these rules are always included in the GSOS systems considered in [2].

7.4.2 Equations for Generating Head Normal Forms

The system of equations generated according to [2] includes an equational theory T_{FINTREE} essentially defining properties of $\cdot + \cdot$:

$$\begin{aligned} p_1 + p_2 &= p_2 + p_1 && (7.30)\\ (p_1 + p_2) + p_3 &= p_1 + (p_2 + p_3) && (7.31)\\ p + p &= p && (7.32)\\ p + \mathbf{0} &= p && (7.33) \end{aligned}$$

Since any process from FINTREE is already in head normal form, there is no need to introduce other equations.

Rules (7.27) to (7.29) from above define the semantics of $\cdot \,\|\, \cdot$. Each of these rules tests the arguments of $p_1 \,\|\, p_2$ in a different way: (7.27) and (7.28) check whether a transition with an action different from c is possible in one of the arguments without considering the other, and (7.29) whether both are ready for a c transition. In order to group rules that perform these checks in the same way, new operators $\lfloor\!\lfloor$, $\rfloor\!\rfloor$, and $\mid$ are introduced for each of the ways that replace $\|$ in (7.27) to (7.29). The resulting rules are

$$\frac{\vdash p_1 \xrightarrow{a} p_1'}{\vdash p_1 \lfloor\!\lfloor p_2 \xrightarrow{a} p_1' \,\|\, p_2} \quad a \neq c \tag{7.34}$$

$$\frac{\vdash p_2 \xrightarrow{a} p_2'}{\vdash p_1 \rfloor\!\rfloor p_2 \xrightarrow{a} p_1 \,\|\, p_2'} \quad a \neq c \tag{7.35}$$

$$\frac{\vdash p_1 \xrightarrow{c} p_1', p_2 \xrightarrow{c} p_2'}{\vdash p_1 \mid p_2 \xrightarrow{c} p_1' \,\|\, p_2'} \tag{7.36}$$

and the new operators are connected by the equation

$$p_1 \,\|\, p_2 = (p_1 \lfloor\!\lfloor p_2) + (p_1 \rfloor\!\rfloor p_2) + (p_1 \mid p_2) \tag{7.37}$$

For each of the new operators, three types of equations are generated:

(1) **distributivity laws** with respect to $+$:

$$(p_1 + p_2) \;\square\; p_3 = (p_1 \;\square\; p_3) + (p_2 \;\square\; p_3)$$

where $\square \in \{\lfloor\!\lfloor, \rfloor\!\rfloor, |\}$.

(2) **action laws**, describing when a process can take an action:

$$\begin{array}{lll} (a\,p_1) \lfloor\!\lfloor p_2 = a\,(p_1 \,\|\, p_2) & ,a \neq c & \text{(from (7.34))} \\ (a\,p_1) \rfloor\!\rfloor p_2 = a\,(p_1 \,\|\, p_2) & ,a \neq c & \text{(from (7.35))} \\ (c\,p_1) \mid c\,p_2 = c\,(p_1 \,\|\, p_2) & & \text{(from (7.36))} \end{array}$$

(3) **inaction laws**, describing when a process cannot take an action, i. e. is equal to $\mathbf{0}$:

$$\left.\begin{array}{l} \mathbf{0} \lfloor\!\lfloor p = \mathbf{0} \\ (c\,p_1) \lfloor\!\lfloor p_2 = \mathbf{0} \end{array}\right\} \text{(from (7.34))}$$

$$\left.\begin{array}{l} p \rfloor\!\rfloor \mathbf{0} = \mathbf{0} \\ p_1 \rfloor\!\rfloor (c\,p_2) = \mathbf{0} \end{array}\right\} \text{(from (7.35))}$$

$$\left.\begin{array}{l} \mathbf{0} \mid p = \mathbf{0} \\ p \mid \mathbf{0} = \mathbf{0} \\ (a\,p_1) \mid (b\,p_2) = \mathbf{0} \quad a \neq c \text{ or } b \neq c \end{array}\right\} \text{(from (7.36))}$$

In the general case when there are negative premises to consider, some more equations are generated. In our example, however, we do not need those.

The equational theory T generated by the algorithm has two important properties:

(1) **correctness**: for all $\ulcorner p = q \urcorner \in T$ and all ground substitutions σ, $p\sigma$ and $q\sigma$ bisimulate each other, i. e. there exists a relation $\sim$ on processes such that for all $a \in Act$, the following two conditions hold:

- $\forall p' : p\sigma \xrightarrow{a} p' \;\Rightarrow\; (\; \exists q' : q\sigma \xrightarrow{a} q' \;\wedge\; p' \sim q' \;)$
- $\forall q' : q\sigma \xrightarrow{a} q' \;\Rightarrow\; (\; \exists p' : p\sigma \xrightarrow{a} p' \;\wedge\; p' \sim q' \;)$

(2) **completeness**: any process term $p \in Proc$ can be transformed into head normal form using the equations from T.

Note that the equations from T cannot simply be turned into rewrite rules as the axiomatization of $+$ in T_{FINTREE} leads to problems with termination.

7.4.3 Rewrite Rules According to Section 7.2

In order to be able to apply the algorithm of Section 7.2 to the language from Section 7.4.1, first the SOS system has to be transformed into an unlabelled one (cf. Lemma 5.2). This results in configurations $\langle\, p, h \,\rangle$ that consist of a process and a trace $h \in Act^*$:

$$\begin{array}{l} \Gamma = Proc \times Act^* \\ T = \{\mathbf{0}\} \times Act^* \end{array}$$

The rules are transformed in the following way where $\cdot\hat{}\cdot : Act^*, Act \to Act^*$ is the concatenation operation on traces:

$$\vdash \langle\, a\,p, h \,\rangle \longrightarrow \langle\, p, h\hat{}a \,\rangle \tag{7.38}$$

$$\frac{\vdash \langle p_1, \varepsilon \rangle \longrightarrow \langle p_1', a \rangle}{\vdash \langle p_1 + p_2, h \rangle \longrightarrow \langle p_1', h\hat{\ }a \rangle} \tag{7.39}$$

$$\frac{\vdash \langle p_2, \varepsilon \rangle \longrightarrow \langle p_2', a \rangle}{\vdash \langle p_1 + p_2, h \rangle \longrightarrow \langle p_2', h\hat{\ }a \rangle} \tag{7.40}$$

$$\frac{\vdash \langle p_1, \varepsilon \rangle \longrightarrow \langle p_1', a \rangle \ \wedge\ a \neq c}{\vdash \langle p_1 \,\|\, p_2, h \rangle \longrightarrow \langle p_1' \,\|\, p_2, h\hat{\ }a \rangle} \tag{7.41}$$

$$\frac{\vdash \langle p_2, \varepsilon \rangle \longrightarrow \langle p_2', a \rangle \ \wedge\ a \neq c}{\vdash \langle p_1 \,\|\, p_2, h \rangle \longrightarrow \langle p_1 \,\|\, p_2', h\hat{\ }a \rangle} \tag{7.42}$$

$$\frac{\vdash \langle p_1, \varepsilon \rangle \longrightarrow \langle p_1', c \rangle, \langle p_2, \varepsilon \rangle \longrightarrow \langle p_2', c \rangle}{\vdash \langle p_1 \,\|\, p_2, h \rangle \longrightarrow \langle p_1' \,\|\, p_2', h\hat{\ }a \rangle} \tag{7.43}$$

From these rules, the algorithm generates the following rewrite rules (only the form for 1-contexts is presented; for the abbreviations, see Section 7.2):

$$\langle a\,p, h \rangle \mathbin{@} Kv_1^{(1)} \longrightarrow \langle p, h\hat{\ }a \rangle \mathbin{@} K_{0f} \tag{7.44}$$

$$\begin{aligned} \langle p_1 + p_2, h \rangle \mathbin{@} Kv_1^{(2)} \longrightarrow\ & \textsf{let}\ \ \gamma = \mathit{eval}(\langle p_1, \varepsilon \rangle \mathbin{@} K_{1f}) \\ & \textsf{in}\ \langle \gamma \downarrow 1, h\hat{\ }(\gamma \downarrow 2) \rangle \mathbin{@} K_{0f} \end{aligned} \tag{7.45}$$

$$\begin{aligned} \langle p_1 + p_2, h \rangle \mathbin{@} Kv_1^{(3)} \longrightarrow\ & \textsf{let}\ \ \gamma = \mathit{eval}(\langle p_2, \varepsilon \rangle \mathbin{@} K_{1f}) \\ & \textsf{in}\ \langle \gamma \downarrow 1, h\hat{\ }(\gamma \downarrow 2) \rangle \mathbin{@} K_{0f} \end{aligned} \tag{7.46}$$

$$\begin{aligned} \langle p_1 \,\|\, p_2, h \rangle \mathbin{@} Kv_1^{(4)} \longrightarrow\ & \textsf{let}\ \ \gamma = \mathit{eval}(\langle p_1, \varepsilon \rangle \mathbin{@} K_{1f}) \\ & \textsf{in if}\ \ hd(\gamma \downarrow 2) \neq c \\ & \qquad \textsf{then}\ \ \langle \gamma \downarrow 1 \,\|\, p_2, h\hat{\ }(\gamma \downarrow 2) \rangle \mathbin{@} K_{0f} \\ & \qquad \textsf{else}\ \ \langle p_1 \,\|\, p_2, h \rangle \mathbin{@} \overline{Kv}_1^{(4)} \end{aligned} \tag{7.47}$$

$$\begin{aligned} \langle p_1 \,\|\, p_2, h \rangle \mathbin{@} Kv_1^{(5)} \longrightarrow\ & \textsf{let}\ \ \gamma = \mathit{eval}(\langle p_2, \varepsilon \rangle \mathbin{@} K_{1f}) \\ & \textsf{in if}\ \ hd(\gamma \downarrow 2) \neq c \\ & \qquad \textsf{then}\ \ \langle p_1 \,\|\, \gamma \downarrow 1, h\hat{\ }(\gamma \downarrow 2) \rangle \mathbin{@} K_{0f} \\ & \qquad \textsf{else}\ \ \langle p_1 \,\|\, p_2, h \rangle \mathbin{@} \overline{Kv}_1^{(5)} \end{aligned} \tag{7.48}$$

$$\begin{aligned} \langle p_1 \,\|\, p_2, h \rangle \mathbin{@} Kv_1^{(6)} \longrightarrow\ & \textsf{let}\ \ \gamma_1 = \mathit{eval}(\langle p_1, \varepsilon \rangle \mathbin{@} K_{1f}), \\ & \qquad\ \gamma_2 = \mathit{eval}(\langle p_2, \varepsilon \rangle \mathbin{@} K_{1f}) \\ & \textsf{in if}\ \ hd(\gamma_1 \downarrow 2) = c\ \wedge\ hd(\gamma_2 \downarrow 2) = c \\ & \qquad \textsf{then}\ \ \langle \gamma_1 \downarrow 1 \,\|\, \gamma_2 \downarrow 1, h\hat{\ }c \rangle \mathbin{@} K_{0f} \\ & \qquad \textsf{else}\ \ \langle p_1 \,\|\, p_2, h \rangle \mathbin{@} \overline{Kv}_1^{(6)} \end{aligned} \tag{7.49}$$

Obviously, these rules are more complex than those in the previous section. But this is not surprising; the algorithm of Aceto, Bloom, and Vaandrager is tailored to this process algebra situation, whereas the algorithm of Section 7.2 is designed to handle a more general class of problems. The main differences in the two approaches are that the algorithm from [2] generates an equational theory and introduces new operators, whereas the *ptp/t* algorithm uses only the operators of the original system and generates a “non-equational” rewrite system (cf. Section 7.3.2): rewriting a configuration produces one of the possible outcomes at a time (and not the collection of all of them). To get all possible results, one has to control the rewriting process from outside or add new operators in the way it is done in [2], since there is no other way to construct a complete equational theory (cf. Moller [94]).

Chapter 8

Modelling of Denotational Semantics Definitions

The task of transforming a denotational semantics definition $\mathcal{D}$ in clausal form into a rewrite system representation is twofold. First, rewrite rules have to be derived from the defining equations, and second, the mathematical domains that form the basis of the definition have to be represented. This chapter shows how both tasks can be solved in a way that is consistent with the approach presented in the previous chapter for modelling SOS definitions.

Since the clausal form equations are already given in an appropriate format with a natural orientation from left to right, their conversion into rewrite rules is rather straightforward and much less difficult than the transformation of SOS rules. In order to be able to cope with bound variables in let terms and λ-abstractions occurring on the right-hand side of equations, again the $\lambda\sigma$-calculus of Section 3.2 is used. In accordance with Section 4.2.5.1 it is also applied to model quantified formulas that arise from the defining axioms of the mathematical domains. The second section show how this approach is used to model cpo structures. An additional problem occurring in this example is the need for suitable representations for monotonic and continuous functions that yield the fixed points used for the definition of recursive structures like while loops or procedure calls. Section 8.2.1 shows some of the problems concerning fixed points in the cpo-setting of the examples.

8.1 Turning Clausal Form Definitions into Rewrite Rules

Let C be a syntactic category defined by the context-free productions

$$C \ni p ::= p_1 \mid \ldots \mid p_n \qquad , n \geq 1$$

and let the semantic function $[\![\cdot]\!] : C \to D$ be given in clausal form according to Def. 6.12. Then there are n clauses of the form

$$[\![c_i]\!]\ a_{i1} \ldots a_{ip_i} = \begin{cases} result_{i1} & \text{, if } condition_{i1} \\ \vdots & \\ result_{im_1} & \text{, if } condition_{im_1} \end{cases} \tag{8.1}$$

An obvious translation of such a clause uses the conditional operator $\mathsf{if}\ \_\ \mathsf{then}\ \_\ \mathsf{else}\ \_$ and derives a single rewrite rule directly from the clause's meaning as defined in Section 6.3:

$$\begin{array}{ll} [\![c_i]\!]\ a_{i1} \ldots a_{ip_i} \longrightarrow \mathsf{if} & \mathit{condition}_{i1} \\ \quad\quad\quad\quad\quad\quad\quad\quad \mathsf{then} & \mathit{result}_{i1} \\ \quad\quad\quad\quad\quad\quad\quad\quad \mathsf{else} & \ldots\ \mathsf{if}\ \mathit{condition}_{im_i} \\ & \quad\quad \mathsf{then}\ \mathit{result}_{im_i} \\ & \quad\quad \mathsf{else}\ \underline{\mathsf{undefined}}_D \end{array} \tag{8.2}$$

This rule suffices because there is exactly one clause for each $i \in [n]$; there is no need to provide a "way back" as in the SOS-derived rules. In most cases, $\mathit{condition}_{im_i}$ will be **true**, which means that the innermost $\mathsf{if}\ \_\ \mathsf{then}\ \_\ \mathsf{else}\ \_$ can be replaced by result_{im_i}. Otherwise, there may be cases when $[\![c_i]\!]$ is not defined because none of the $\mathit{condition}_{ij}$ is satisfied. The purpose of the special term $\underline{\mathsf{undefined}}_D \in \Sigma_{\varepsilon,D'}$ is to indicate such cases; it does not denote an element of D', but rather that none of these elements may take its place.

Note that the $\mathit{condition}_{ij}$ and result_{ij} may contain let subterms. These can be dealt with in the style introduced in Section 3.2 and already used in Section 7.2, i. e. by applying the $\lambda\sigma$-calculus.

Dealing with Explicit Abstractions

Denotational function definitions often make use of explicit λ-abstractions in order to construct unnamed functions. There are two main reasons for this:

(1) A semantic function $[\![\cdot]\!] : C \to D$ is defined in curried form, i. e. its result domain has the functional structure $D = E \to F$ for some domains E and F.

Example (notationally adapted from Stoy [113], p. 196):

Consider a language containing conditional expressions:

$$\mathit{Expr} \ni e ::= \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 \mid \ldots$$

The semantics of expressions is a truth value $b \in \mathbb{B}$, depending on states $\sigma \in \Sigma$:

$$[\![\cdot]\!] : \mathit{Expr} \to \Sigma \to \mathbb{B}$$

Then the semantics of conditional expressions can be defined in the one-result clause:

$$[\![\mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3]\!] = \lambda\sigma : \Sigma\ .\ \mathsf{if}\ [\![e_1]\!]\,\sigma\ \mathsf{then}\ [\![e_2]\!]\,\sigma\ \mathsf{else}\ [\![e_3]\!]\,\sigma \tag{8.3}$$

(Note that the conditional expression on the right-hand side belongs to the meta-level.)

(2) Some function occurring in the result_{ij} requires functional parameters and these are explicitly constructed. A typical case where this happens very often are semantics definitions involving **continuations** (cf. Stoy [113], p. 251 ff. or Gordon [58], p. 52 ff.). The continuation of a piece of program represents the remainder of the program to be executed after the evaluation of that piece; it takes the result of that evaluation as its argument and produces a new result from it.

Example (notationally adapted from Gordon [58], p. 58):

Consider another language containing expressions denoting truth values and natural numbers ($\mathsf{Val} = \mathbb{B} \uplus \mathbb{N}$):

$$\mathit{Expr} \ni e ::= 0 \mid e_1 = e_2 \mid \mathsf{not}\ e_1 \mid \ldots$$

Here, the semantics of an expression not only depends on the current state $\sigma \in \Sigma$, but also on its "expression continuation" $\kappa \in \mathit{ECont} = \mathsf{Val} \rightarrow \Sigma \rightarrow (\Sigma \uplus \{\mathsf{error}\})$:

$$[\![\cdot]\!] : \mathit{Expr} \rightarrow \mathit{ECont} \rightarrow \Sigma \rightarrow (\Sigma \uplus \{\mathsf{error}\}$$

The defining clause for the not operator is the following:

$$[\![\mathsf{not}\ e]\!]\, \kappa\sigma = [\![e]\!]\, (\lambda v : \mathsf{Val}, \sigma' : \Sigma\ .\ \mathsf{if}\ v \in \mathbb{B}\ \mathsf{then}\ \kappa(\neg v)\sigma\ \mathsf{else}\ \mathsf{error})\ \sigma \tag{8.4}$$

Of course these abstractions are not only used to define the semantics, but they are also applied to given arguments when the semantics of concrete expressions is to be evaluated. Therefore, β-reduction must also be simulated if function definitions with λ-abstractions are modelled.

In the setting of this work, this is not a fundamental problem since techniques such as the $\lambda\sigma$-calculus (cf. Section 3.2) can be used to implement β-reduction and substitution handling. But one should note that application of the rewrite rules defining the $\lambda\sigma$-calculus (cf. Def. 3.51) can be quite time-consuming, in particular when substitutions are applied to large terms as they often occur in denotational definitions. So it is worthwhile to try to remove the explicit abstractions beforehand by performing abstract β-reductions.

This is hardly possible for abstractions of the form of the second example above, because in equations such as (8.4) it is not known yet what the arguments might be that are passed to the continuation. Usually, the information about this parameter passing is only contained in the definition of the semantics of the most primitive language elements. In the example (2) from above, e. g., the semantics of the truth values is defined by equations such as $[\![\mathsf{true}]\!]\, \kappa\ \sigma = \kappa\ (\mathsf{tt})$.

But in the other case, the removal of abstractions is fairly easy. In an equation such as (8.3), it simply amounts to adding the bound variables occurring outermost on the right-hand side as additional arguments to the left-hand side, and removing the λ-abstraction. This exactly produces the effect of β-reduction, and for the example it results in:

$$[\![\mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3]\!]\ \sigma = \mathsf{if}\ [\![e_1]\!]\ \sigma\ \mathsf{then}\ [\![e_2]\!]\ \sigma\ \mathsf{else}\ [\![e_3]\!]\ \sigma \tag{8.5}$$

This method also works for clauses with more than one possible result, because by condition (6) in Def. 6.12 all of these results are abstractions, or none of them is. Clauses of the form

$$[\![c_i]\!]\ a_{i1} \ldots a_{ip_i} = \begin{cases} \lambda x : X\ .\ \mathit{result}_{i1} & \text{, if } \mathit{condition}_{i1} \\ \vdots & \\ \lambda x : X\ .\ \mathit{result}_{im_i} & \text{, if } \mathit{condition}_{im_i} \end{cases}$$

can also be treated in the same way by adding x to the arguments on the left-hand side, and removing all of the "$\lambda x : X\ .$" on the left-hand side.

Uncurrying of semantic functions Working with curried functions can lead to rather complicated terms: A curried function application $[\![c]\!]\ a_1 a_2$ must be modelled as $\mathsf{apply}(\mathsf{apply}([\![c]\!], a_1), a_2)$, where the overloaded apply operator has the functionalities $\mathsf{apply} : C, A_1 \rightarrow (A_2 \rightarrow A)$ and $\mathsf{apply} : (A_2 \rightarrow A), A_2 \rightarrow A$.[1]

[1] In a first-order formalism such as the one used here, only simple sorts are allowed for the result type of operators. Therefore, a new sort $D = A_2 \rightarrow A$ must be introduced for the definition of the apply operators.

These complications can be avoided if the above transformations for abstraction removal are performed in such a way that every occurrence of a semantic function $[\![\cdot]\!]$ in the whole definition is supplied with exactly the same number m and the same kind of arguments from domains $A_1, \ldots, A_m$ (assuming $D = A_1 \rightarrow \cdots \rightarrow A_m \rightarrow A$):

$$[\![c]\!]\, a_1 \ldots a_m \qquad (a_j \in A_j \text{ for } j \in [m], c \in C)$$

If this is the case, then instead of the curried version

$$[\![\cdot]\!] : C \rightarrow A_1 \rightarrow \cdots \rightarrow A_m \rightarrow A \tag{8.6}$$

the uncurried version can be used:

$$\mathit{sem} : C, A_1, \ldots, A_m \rightarrow A \tag{8.7}$$

Using the uncurried version in the example from above, $[\![c]\!]\, a_1 a_2$ can be modelled in the form $\mathit{sem}\ (c, a_1, a_2)$ which is much more readable and easier to deal with than the solution using apply operators.[2]

The semantics definitions used as examples in the proof experiments described in Chapter 10 only contained outermost abstractions of the form (8.3), if any. Therefore, they could be removed as described above; due to the special structure of these definitions, the uncurrying step from (8.6) to (8.7) could also be performed, resulting in smaller terms and hence more efficient rewriting and more readable proofs.

8.2 Modelling Cpo's with Rewrite Systems

The basic mathematical structure for the denotational value domains of all the examples is that of a cpo. In this section, rewrite rules are devised that capture the properties of such a structure. Most of the rules of this section consist only of one predicate term p. Of course, they must be read as abbreviations for the rewrite rule $p \longrightarrow \mathsf{true}$.

Let $\langle D, \sqsubseteq \rangle$ be a cpo. The first step is to define a sort D with variables $\mathsf{d}, \mathsf{d}_1, \mathsf{d}_2, \ldots \in V_{\mathsf{D}}$ and an operator $\sqsubseteq : \mathsf{D}, \mathsf{D} \rightarrow \mathsf{Bool}$ that represents the **partial ordering**. This leads to the rules[3]

$$\begin{array}{l} \mathsf{d} \sqsubseteq \mathsf{d} \\ \mathsf{d}_1 \sqsubseteq \mathsf{d}_2 \ \wedge\ \mathsf{d}_2 \sqsubseteq \mathsf{d}_1 \ \Rightarrow\ \mathsf{d}_1 = \mathsf{d}_2 \\ \mathsf{d}_1 \sqsubseteq \mathsf{d}_2 \ \wedge\ \mathsf{d}_2 \sqsubseteq \mathsf{d}_3 \ \Rightarrow\ \mathsf{d}_1 \sqsubseteq \mathsf{d}_3 \end{array} \tag{8.8}$$

The **bottom element** of D is represented by $\bot : \rightarrow \mathsf{D}$; it is defined by the rule

$$\bot \sqsubseteq \mathsf{d} \tag{8.9}$$

[2]The notation $[\![c, a_1, a_2]\!]$ is not adopted because the double brackets $[\![\cdot]\!]$ usually serve to separate the syntactic argument (inside the brackets) from the semantic arguments.

[3]Note that all of these are rewrite rules according to Def. 3.18; $\Rightarrow$ is just a Boolean operator and has nothing to do with conditional term rewriting as defined e. g. by Kaplan [78].

In order to be able to define that a partial order constitutes a cpo, the concept of **chains** is required. According to Def. 6.2, these are a special form of sequences, and so a sort seqD is introduced into $\mathcal{S}$ that is intended to model the set of sequences $D^{\mathbb{N}}$. To this end, an application operator

$$. : \mathsf{seqD}, \mathsf{Nat} \rightarrow \mathsf{D}$$

is defined; $\mathsf{ds}\ .\ \mathsf{n}$ represents the $n-$th element of the sequence $ds \in D^{\mathbb{N}}$.

That a sequence $ds \in D^{\mathbb{N}}$ is a chain is expressed by the equivalence

$$ds \text{ is a chain} \iff \forall n \in \mathbb{N} : ds_n \sqsubseteq ds_{n+1} \tag{8.10}$$

An equivalence can be read as an equation on the sort Bool, and by applying the the method of Section 4.2.5.1, it can be turned into a rewrite rule that defines the recognizer operator $\mathsf{chain} : \mathsf{seqD} \rightarrow \mathsf{Bool}$:

$$\mathsf{chain}(\mathsf{ds}) \longrightarrow \mathsf{forall}(\_\mathsf{Nat}, \mathsf{ds}.\_\mathsf{n} \sqsubseteq \mathsf{ds}.(\_\mathsf{n}+1)) \tag{8.11}$$

(assuming that the addition operator $+ : \mathsf{Nat}, \mathsf{Nat} \rightarrow \mathsf{Nat}$ and $1 : \rightarrow \mathsf{Nat}$ have already been appropriately defined and $\_\mathsf{n}$ is the first de Bruijn index of sort Nat). Note, however, that this method of using quantifiers in terms has its fundamental weaknesses as already mentioned in Section 4.2.5.1. These rules may only be used if nested quantifiers ranging over the same sort do never occur.

In cpo's, chains have a **least upper bound**. So an operator $\mathsf{lub} : \mathsf{seqD} \rightarrow \mathsf{D}$ is introduced and defined by the rules

$$\begin{array}{l} \mathsf{chain}(\mathsf{ds}) \Rightarrow \mathsf{ds.n} \sqsubseteq \mathsf{lub}(\mathsf{ds}) \\ \mathsf{chain}(\mathsf{ds}) \Rightarrow (\ \mathsf{forall}(\_\mathsf{Nat}, \mathsf{ds}.\_\mathsf{n} \sqsubseteq \mathsf{d}) \Rightarrow \mathsf{lub}(\mathsf{ds}) \sqsubseteq \mathsf{d}\ ) \end{array} \tag{8.12}$$

For sequences d that are not chains, $\mathsf{lub}(\mathsf{d})$ is still a well-formed term; it is not defined by any rewrite rule, however.

When dealing with chains that only have a finite number of different elements, e. g. because they are representations of finite chains (with the last element repeated), Lemma 6.4 turns out to be very useful for calculating the lub of these. So another rule is added (assume that n is a variable of sort Nat):

$$\mathsf{chain}(\mathsf{ds}) \Rightarrow (\ \mathsf{forall}(\_\mathsf{Nat}, \_\mathsf{n} \geq \mathsf{n} \Rightarrow \mathsf{ds}.\_\mathsf{n} = \mathsf{ds.n}\ ) \Rightarrow (\ \mathsf{lub}(\mathsf{ds}) = \mathsf{ds.n}\ )\ ) \tag{8.13}$$

In Appendix C, an example is presented where these rules are used to prove that the domain of functions between two cpo's again forms a cpo, and this proof is compared to a proof performed not just by rewriting, but also by other methods.

8.2.1 Handling of Fixed Points

For some applications, a representation of the least fixed point operator μf of Def. 6.7, or a way to deal with fixed points in general will be needed. In particular, the μ operator may occur in the clauses of denotational function definitions of the form introduced in Def. 6.12. It is easy to define a fixed point x of a function $f : X \to X$ by an equation:

$$x = f(x) \tag{8.14}$$

But the problem with this equation is that it is difficult to use it as a rewrite rule. The direction

$$\mathsf{f(x)} \longrightarrow \mathsf{x} \tag{8.15}$$

is mostly useless; in the applications at hand, x stands for the semantics of some kind of recursive program part, and it is given without the surrounding $f(\ldots)$. This means that (8.15) cannot be applied. On the other hand,

$$\mathsf{x} \longrightarrow \mathsf{f(x)} \tag{8.16}$$

can be applied to unfold the semantics; but this rule is obviously not terminating. So one has to make sure that (8.16) is applied in a very controlled way; in proofs, this usually means "just once" because after one step a point is reached where some induction hypothesis can be exploited and so a complete evaluation of the term containing the fixed point (for examples of such proofs, see Chapter 10) is not necessary. Usually, the operator f does not have the regular properties required to apply the control mechanism of Section 3.3 that uses contexts, and so some other technique must be used. The example proofs of Chapter 10 rely on control features of the proof tool that is used; here, it suffices to be able to mark the rule (8.16) as "only to be used on explicit command". Without such features, a proof tool in general cannot be used to implement reasoning about fixed points in the setting of this chapter.

An example for reasoning about fixed points with the help of a proof tool can be found in Broy [17]. In this report, several experiments with the Larch Prover are described, and one of them consists of a correctness proof for code generation for a functional language whose semantics is defined in denotational form.

Chapter 9

The Larch Prover as a Tool for Implementation

The development of the simulation methods of Chapters 7 and 8 was motivated by the wish to be able to use a proof tool in reasoning about SOS and denotational semantics definitions. As an example for such a tool, the Larch Prover has been chosen for the implementation of the methods and for performing some example proofs. This chapter will provide a short introduction into the prover and describe those features that are needed to understand the examples. In particular, term rewriting and general theorem proving aspects will be mentioned.

9.1 Introduction

The Larch Prover (LP for short, see Garland and Guttag [51, 52]) is a proof support system for a subset of many-sorted first-order logic[1]. Originally, it has been developed for analysing specifications written in Larch (cf. Guttag and Horning [62, 63, 64]), but it has also been applied in other areas like software and hardware verification and reasoning about algorithms. An overview of recent activities can be found in Martin and Wing [88].

Since LP has been developed from the REVE rewrite rule laboratory (cf. Forgaard and Guttag [44] and Lescanne [84]), its main strength lies in the area of term rewriting. But LP also provides features that make it a useful tool for proof development. It is not, however, designed to be an automatic theorem prover, but rather an interactive proof checker or, as Garland and Guttag [52] put it, a "proof debugger". In general, it only makes sense to start working with LP if a not too rough idea of the proof to be developed exists beforehand.[2]

LP is written in CLU and runs under UNIX on several different types of hardware, including SUN workstations on which the example proofs of this work have been implemented. Although it is possible to work with LP even on smaller machines if the problems are not too big, it is advisable to use fast machines with a large memory for realistic applications. Only then does working with LP become really interactive.

[1]The version of LP used in the examples and described here is Release 2.4. Not all the items mentioned in this chapter may pertain to other versions.

[2]In fact, this is true for all automated proof assistants.

Notation: Input for and output of LP will be written in `typewriter font` in order to distinguish it from the notation for abstract terms of the previous chapters which were written in sans serif font.

9.2 Specification and Rewriting Features of Special Interest

All proofs with LP have to start with the definition of an equational specification using the many-sorted approach described in Chapter 3. Only the sort Bool modelling the truth values has a predefined representation in LP (`Bool`); all other sorts have to be introduced explicitly.

Only a subset of first-order logic is supported. In particular, there is (almost) no way to use quantifiers in `Bool` sorted terms. (A very restricted form of universal quantification is allowed in deduction rules; see below.) LP provides the possibility to declare binary infix operators (but no mixfix operators), and to declare operators as commutative or associative and commutative. The problem with the definition of commutativity by non-terminating rewrite rules of the form $a * b \rightarrow b * a$ does not exist here, since the special operator properties are already respected in matching, substitution, and rewriting.

The semantic approach taken by LP is that of **loose specification**; so any term-generated algebra that satisfies all equations may be taken as a model of the specification.

From the equational specification, LP generates a term rewriting system by turning the equations into rewrite rules. Methods are provided that guarantee termination and confluence of rewrite systems (by construction of a termination ordering, see Def. 3.40, or by calculation of critical pairs, see Def. 3.46, respectively); application of these methods is not mandatory, however.

Example 9.1

The running example of this section is related to Chapter 4; an axiomatization for VDM-style domains is to be developed. So assume that the following domain equation is given:

$$\mathsf{Val} = \mathsf{Bool} \mid \mathsf{Nat} \times \mathsf{Nat} \mid \underline{\mathsf{error}}$$

i. e. elements of the domain Val can either be Booleans, pairs of natural numbers or the special element error.

The first step to model this domain in LP is to introduce the sorts:

```
declare sorts Val, Nat
```

(as said before, `Bool` is predefined). Next, the variables for the different sorts are introduced.

```
declare variables
  b, b1, b2 : Bool
  n, n1, n2 : Nat
  v, v1, v2 : Val
..
```

(The final "`..`" line terminates a declaration that exceeds a single line.)

Now operators are declared in the style of Section 4.2.1. For each of the subdomains of Val, there is a constructor mapping from the subdomain into Val, and for each subdomain with a name, an injector into that subdomain and a recognizer. For tuple domains, additionally selector (projection) operators are desired. Tokens are taken as constants. This leads to the following set of rules describing the relation between the standard operators:

```
declare operators
  % constructors:
  mk_Val : Bool -> Val
  mk_Val : Nat, Nat -> Val
  _ERROR : -> Val
  % selectors for Nat x Nat:
  s_1 : Val -> Nat
  s_2 : Val -> Nat
  % injector into the subdomain Bool:
  to_Bool : Val -> Bool
  % recognizer for the subdomain Bool:
  is_Bool : Val -> Bool
..
```

There are some peculiarities with this declaration. First note that overloading of operators is allowed in LP. There is a type checker built in that guarantees that all terms that occur are well-typed; it may become necessary to decorate terms with a type to make this checking possible. And second, there are two different forms of using the sort `Bool` in this declaration. In `to_Bool`, it serves as the name of a subdomain of Val, whereas in the declaration of `is_Bool`, `Bool` represents the "meta sort" of Booleans.

□

After the signature is fixed, equations can be entered. There are two ways to do so: either by separating left- and right-hand side by a `->`, or by a `==`. In the first case, the equation is considered as directed, the direction of the rule being determined by the arrow, whereas there is no direction in the second case.

In order to express that some Bool-sorted term equals **true**, it suffices just to write the term without "`-> true`" or "`== true`"; this is added by LP automatically. Rules of the form `not(t)` are converted automatically into the form `t -> false`.

Example 9.2

Some of the laws valid in the example can be stated as follows:

```
set name-prefix Val
assert
  s_1(mk_Val(n1, n2)) -> n1     % Constructor-selector rules
  s_2(mk_Val(n1, n2)) -> n2
  is_Bool(mk_Val(b))            % Recognizer rules
  not(is_Bool(mk_Val(n1, n2))
  not(is_Bool(_ERROR))
..
```

In LP, each rule has a unique name. Setting the name prefix to `Val` here has the effect that the rules can be addressed as `Val.1`, ..., `Val.5`. The command `assert` begins the definition of axioms, in this example the definition of a set of equations.

□

The equations cannot be used for rewriting until they have been turned into rewrite rules. This is also true for equations `l -> r`; here the arrow only indicates that the direction "right to left" is not allowed. LP transforms equations into rewrite rules by "ordering" them, i. e. by trying to find a termination ordering such that the left-hand side of each resulting rule is strictly larger than the right-hand side (see Theorem 3.41). This process is started either automatically after the rules have been entered (if the switch `automatic-ordering` is set to `on`), or by entering the command `order`.

The usual ordering process is based on weights assigned to the operators of the signature; several strategies for the construction of the ordering are available. Especially when there are many operators to consider, this can be very time-consuming; it might also happen that LP does not succeed in ordering all rules on its own. In these cases, there are two possibilities. The first is to instruct LP to order equations in the direction they were entered by typing

```
set ordering-method left-to-right
```

in which case termination cannot be guaranteed anymore. The second way to proceed is to order the rules interactively by proposing suitable operator weights .

After the rules have been ordered, they can be used to rewrite terms. The default case is that this is done using all rules; if certain rules shall be excluded (e. g. because they are known to be non-terminating or because they define an abbreviation that is to be unfolded only on special occasions), this can be done by writing

`make` *name* `passive`

where *name* subsumes all names that have *name* as their name-prefix. It is also possible to use UNIX-like wildcards in *name*; e. g. `*hyp` subsumes all name-prefix ends with `hyp`. The inverse action is:

`make` *name* `active`

Passive rules can still be used; but they must be addressed explicitly by

`rewrite` *term* `with` *name*

(if *term* is to be rewritten just once using rules with name prefix *name*) or

`normalize` *term* `with` *name*

(if a normal form of *term* with respect to the set of rules with name prefix *name* is to be generated). In both cases, *term* will be normalized with the set of all active rules.

LP maintains the invariant that the rules of the system are normalized with respect to each other ("internormalized"). Rules can be protected from this process by writing

`make immune` *name* .

This can be useful if all the terms in a rule R are known to be normal forms and the rewriting system is large. In this case, the time saved by not trying to apply in vain all the rules to the rule R can be quite considerable.

Critical pairs between rules can be computed by writing

`critical-pairs` $name_1$ `with` $name_2$

All pairs found during this process are ordered into rewrite rules as described above and added to the system. An attempt to compute all critical pairs of a system is started by

```
complete  .
```

If this procedure terminates successfully, the resulting system is TRS-complete. Note, however, that it does not necessarily terminate (cf. Section 3.1.3).

Normally, rewriting sequences are not fully documented. This can be changed by

```
set trace-level n
```

where a larger number n causes LP to print more information during rewriting, e. g. attempts to apply rules and matching substitutions.

9.3 LP as a Theorem Prover

There is a variety of proof methods available in LP, both in forward and in backward direction. *Forward inference* (also called *goal-directed inference*) uses the given facts and hypotheses to deduce additional information, whereas *backward inference* (or *subgoal-directed inference*) starts from the goal to be proved and reduces it to (hopefully simpler) subgoals.

As already said before, however, LP has not been designed to be used as an automatic theorem prover. So there are only very restricted possibilities to define strategies or tactics as e. g. in HOL [59], KIV [65] or PVS [104]. There is no way to "program" proofs by taking proof steps only under certain conditions or by repeating them until some condition becomes true. The decisions needed for building complex proof structures always have to be taken by the user. They can, however, be stored in a file (a "proof script"), so proofs can at least be repeated.

In the following, the main proof methods will be presented.

Normalization by term rewriting is the fundamental method since LP is based on rewrite systems. It is used in a forward way by rewriting facts and hypotheses, but also in the other direction by rewriting the current goal.

Computation of critical pairs is sometimes a very useful way of forward inference. Since the process stops when some goal has been proved by the newly generated rewrite rules, it can be used to finish a proof when all the required information has been collected, but a direct rewrite proof is not possible.

Example: Consider the following situation. The rewrite system R is based on the signature

```
declare sort S
declare variables x, y: S
declare operators
  a, b : -> S
  f    : S -> Bool
  g    : S, S -> Bool
..
```

and contains the rules

```
f(x, y) -> g(x)
f(a, b) -> true
```

The goal that is to prove is `g(a)`.

It is easy to see that `g(a)` equals `true` in the theory generated by R, but no rewrite proof is possible since there is no `g`-rule (R is not TRS-complete).[3] But by computing critical pairs,

[3]Note that the first rule cannot be ordered in the other direction because
$var(\ulcorner \texttt{g(x)} \urcorner) = \{x\} \not\supseteq \{x, y\} = var(\ulcorner \texttt{f(x,y)} \urcorner)$.

the goal can be proved; in this simple case, the goal even *is* the only critical pair of the two rules: `g(a) = true`.

Application of deduction rules: Besides rewrite rules, LP also provides a restricted form of deduction rules, written in the form

```
assert
  when premise
  yield conclusion
..
```

where both *premise* and *conclusion* are `Bool`-sorted terms. When *premise* can be deduced from the system, *conclusion* is added to it. A particularity of deduction rules is that the premise may contain universal quantifiers whereas normally quantifiers are not part of LP's object language.[4] As an example for such a rule, consider the modelling of least upper bounds in some cpo D (cf. Def. 6.1 and rule (8.12)). Assume that `ch` is a variable for chains in D, and that ⌜`ch.n`⌝ represents the n-th element of a chain `ch`. Let `lub` be the operator mapping a chain to its least upper bound, let `<=` model the ordering on D, and let `d` be a variable for elements of D. Then the property that the lub of a chain is smaller than all upper bounds can be expressed by the following deduction rule:

```
assert
  when (forall n) ch . n <= d
  yield lub(ch) <= d
..
```

By using deduction rules instead of those rewrite rules of Section 8.2 that explicitly mention quantifiers, quantifiers can be completely avoided. Therefore one does not have to provide rules that model predicate calculus; since LP already includes rules for propositional logic, this means that one does not have to set up logical rules at all. See Appendix C.3 for an example proof with deduction rules.

Induction is one of the most important methods of backward inference in LP. The special form available is called "generator induction". In the following, this concept will be explained using a simple example. Consider the signature

```
declare sorts S, T
declare variables s : S, t : T
declare operators
  b : T -> S
  r : S, T -> S
..
```

where `S` is the representation of some set S. If all elements of S can be represented by terms whose subterms of sort `S` are constructed by `b` or `r`, then `S` is said to be generated by `b` and `r`. In this case, a predicate term `p[s]` depending on the variable `s` can be proved using structural

[4]This is one of the major disadvantages of LP as it sometimes makes direct formulation of properties impossible.

induction:

```
      prove p[s] by induction on s
```

In the induction basis, `p[b(t)]` is considered, and in the induction step `p[r(sc, tv)]` where `sc` is a newly generated (induction) constant of sort `S` and `tv` a newly generated variable of sort `T`. The induction hypothesis for the new constant is `p[sc]`.

The induction rule behind this procedure is the following:

$$\frac{\forall t \in T : b(t) \ \wedge \ (\forall s_1 \in S : p[s_1] \ \Rightarrow \ p[r(s_1, t)]\,)}{\forall s \in S : p[s]}$$

where the symbols in italics are the semantic counterparts of the symbols in typewriter font consisting of the same characters.

In order to enable this induction rule, one has to enter

```
      assert S generated by b, r  .
```

In LP's terminology, this rule is also called an "induction rule".

The example situation is extended in the obvious way if there is more than one base case or more than one case in the induction step (caused by more operators mapping into `S` without or with `S` as an argument sort).

Induction was particularly important for the proofs using the simulation methods. The semantics definitions considered all consist of an inductive definition, based on the recursive structure of language constructs given by a context-free grammar. And so proofs generally start with an induction on this recursive structure.

Other methods A frequently used proof method is case distinction; in LP, this is initiated by

```
      prove p by cases t1,...,tn
```

where $\texttt{p},\texttt{t}_1,\ldots,\texttt{t}_n$ are `Bool`-sorted terms. The result of this proof command is that LP generates $n+1$ new subgoals: one for proving `p` using each of the $\texttt{t}_i$ in turn as additional hypothesis, and one for proving that the case distinction is exhaustive (i. e. $\texttt{t}_1$ `OR` ...`OR` $\texttt{t}_n$ `= true`).

Proof commands that exploit the logical structure of goals are the methods for implications and conjunctions:

```
      prove p => q by =>-method
```

causes LP to add `p` to the hypothesis in an attempt to prove `q`. In order to retain consistency, all variables occurring in `p` have to be turned into constants during this proof (both in `p` and in `q`). The names for these constants are generated by LP; typically, a "`c`" (for constant) is appended to the original name of the variable.

Conjunctions `p & q` can be proved by typing

```
      prove p & q by &-method  .
```

This start two separate proofs for `p` and `q` (this results in less complex conjectures). Finally,

```
      prove p by contradiction
```

causes LP to add `not(p)` to the hypotheses and to generate the subgoal `true = false`.

Resuming proofs If a proof method does not succeed in finishing a proof completely, the proof can be continued using another method M by entering:

```
    resume by M
```

Testing for the end of a proof LP provides a special command for checking whether all conjectures (top-level and lemmas) have been proved:

```
    qed
```

If there still is an unfinished proof, the command results in an error message indicating which proof still has to be finished. If this happens when the command is part of a proof script file, further execution of that file is prevented. So this command can be used inside a proof script to check that all lemmas have been proved before the proof of a theorem is begun.

Chapter 10

Application - a Semantics Equivalence Proof

In this chapter, it will be demonstrated how rewrite systems that are derived from SOS and denotational definitions using the techniques of Chapters 7 and 8 can be applied in computer-aided verification. The problem addressed here has already been solved without mechanical assistance by Lakhneche [82]. It is particularly useful for demonstrating the simulating rewrite systems since it deals with several different semantics definitions; the aim is to prove the equivalence of two different semantics definitions (one SOS and one denotational) for a fixed programming language. Moreover, there already exists a complete hand proof, and hence the peculiarities of the automated proof can be pointed out more clearly.

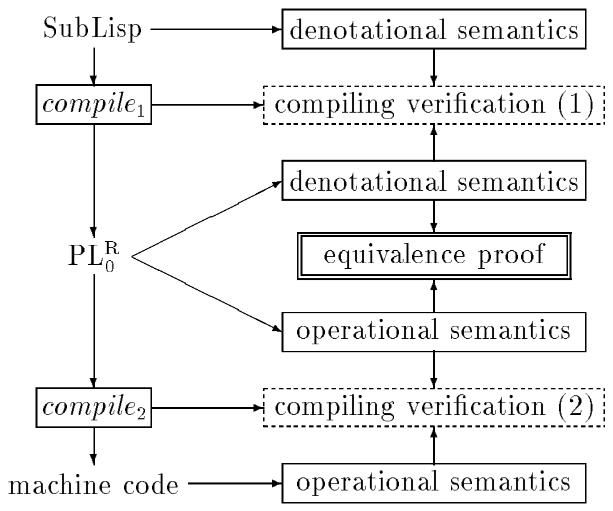
10.1 The Problem

The main objective of the ProCoS project [11, 14] is to demonstrate a method for obtaining a fully verified computer system. An important part of such a system is a compiler for a high-level programming language. It has to generate code correctly, i. e. source and target program have to be semantically "equivalent" (more precisely, the target program has to refine the source program) in a sense that has to be defined suitably to meet the requirements of the actual application.

In order to implement compiling specifications written in a functional style (see e. g. Fränzle [45, 46]), a language called SubLisp is used (cf. Müller-Olm [96]), which is a purely functional subset of Common Lisp (cf. Steele [111]) with simple data. In order to be able to generate compiler programs that are executable on the transputer microprocessor, a compiler for SubLisp was implemented. Aspects of the verification of this compiler will be the topic of this section.

The semantics definitions for SubLisp and for transputer machine code are very different in style: SubLisp is defined denotationally [96], whereas the machine code is defined operationally in SOS style [15]. The gap between the two languages is bridged by using a language called $\mathrm{PL}_0^{\mathrm{R}}$ as intermediate language for the compilation. This language is essentially the language of while programs enriched by input and output features, one-dimensional integer arrays, and parameterless procedures. Since it contains arrays and procedures, it is a suitable target for the compilation of a functional language, and since it is imperative, it is much closer to machine code than SubLisp.

Figure 10.1: Compiling verification for SubLisp



For verification against SubLisp, $\mathrm{PL}_0^{\mathrm{R}}$ has a denotational definition, and for verification against machine code, it has an SOS definition (cf. Fig. 10.1). In Sections 5.2.1 and 6.3.1, parts of these definitions have already been introduced.

The idea is to deduce the correctness of the compilation from SubLisp to machine code from the correctness of the compilations from SubLisp to $\mathrm{PL}_0^{\mathrm{R}}$ ($compile_1$) and from $\mathrm{PL}_0^{\mathrm{R}}$ to machine code ($compile_2$). But since the proof for $compile_1$ is based on denotational semantics and the proof for $compile_2$ on operational semantics, this deduction is only possible if the equivalence of the two semantics definitions for $\mathrm{PL}_0^{\mathrm{R}}$ has been proved. Lakhneche [82] has given such a proof, and the aim is to use LP to produce a mechanically checked version of it.

In Section 10.2, the description of the proof starts with an examination of the subproblem of proving semantics equivalence for $\mathrm{PL}_0^{\mathrm{R}}$ expressions. This section also includes an account of the general inductive proof method used to solve the problems. Section 10.3 then deals with the verification problem for sequential processes.

10.2 Semantics Equivalence for $\mathrm{PL}_0^{\mathrm{R}}$ Expressions

Although expressions form but a small sub-language of $\mathrm{PL}_0^{\mathrm{R}}$, many general proof techniques can already be demonstrated on this subproblem. This section starts with a sufficiently extensive account of its features, introducing all those parts that are necessary to understand the equivalence theorem. The form of exposition is in large parts taken directly from Lakhneche [82]. After that, the general proof method will be explained, and the automated proof will be compared with the hand proof in [82].

10.2.1 Abstract Syntax

$\mathrm{PL}_0^{\mathrm{R}}$ provides the usual set of arithmetical and logical operators (monadic and dyadic):

$$\begin{array}{lcl} mop & \in & MonadicOp \\ dop & \in & DyadicOp \\ mop & ::= & - \mid \texttt{NOT} \\ dop & ::= & + \mid \; = \; \mid \texttt{AND} \mid \ldots \end{array}$$

Variables can be simple identifiers or array components:

$$\begin{array}{lcl} name & \in & Name \\ var & \in & Variable \\ exp & \in & Expr \\ var & ::= & name \mid name[exp] \end{array}$$

Expressions can be integers, truth values, variables or applications of operators:

$$\begin{array}{lcl} int & \in & Int \\ exp & ::= & int \mid \texttt{TRUE} \mid \texttt{FALSE} \mid var \mid mop \;\; exp \mid exp_1 \;\; dop \;\; exp_2 \end{array}$$

An equivalent definition that makes the recursive structure more obvious is the following:

$$exp \; ::= \; int \mid \texttt{TRUE} \mid \texttt{FALSE} \mid name \mid name[exp] \mid mop \;\; exp \mid exp_1 \;\; dop \;\; exp_2$$

The latter form was used because it eases application of LP's built-in induction proof method. Consider the proof of a conjecture $p(exp)$ by induction on the syntactic structure of the expression exp. The case $exp = name[exp_1]$ should belong to the induction step. With the original definition of the syntactic domains, however, the case $exp = var$ is considered by LP as belonging to the induction basis because var does not contain an expression subterm, and so no induction hypothesis is generated.

As an alternative to the merging of the syntactic domains for exp and var, one could add the missing hypothesis as an additional axiom in the case $exp = name[exp_1]$ (or, more exactly, using the term constructors of Chapter 4, $\mathsf{exp} = \mathsf{mk\text{-}Expr}(\mathsf{mk\text{-}Var}(\mathsf{name}, \mathsf{exp}))$). In this small example, merging is the more elegant solution, but in Section 10.3.1 an example of a similar case will be seen where the addition of axioms is the better solution.

10.2.2 Static Semantics

In this simple example, static well-formedness is only a question of correct typing. Identifiers occurring in expressions can either denote integer or array variables. This is recorded in static environments:

$$\begin{array}{l} \delta \in \mathsf{Dict} \;=_{df}\; Name \to \mathsf{Type} \\ \qquad \mathsf{Type} =_{df} \{\texttt{VarInt}\} \uplus \{(\texttt{ArrayInt}, n) \mid n \in \mathbb{N}_0\} \uplus \{\bot\} \end{array}$$

where $\delta(name) = \bot$ iff $name$ has not been declared as an identifier. The type of expressions can be either "Boolean" or "integer":

$$tp \in Tp =_{df} \{\texttt{Bool}, \texttt{integ}\}$$

Formulas $\delta \vdash_{Variable} var$ and $\delta \vdash_{Expr} exp : tp$ are used to express well-formedness of variables and expressions, respectively, with respect to static environments δ. They are defined by a deduction system; the format of the rules can be seen in the following examples:

$$\text{(ER1)} \quad \frac{\delta \longmapsto_{Variable}\ var}{\delta \longmapsto_{Expr}\ var : \mathtt{integ}}$$

$$\text{(ER2)} \quad \delta \longmapsto_{Expr}\ int : \mathtt{integ}$$

$$\text{(ER6)} \quad \frac{\delta \longmapsto_{Expr}\ exp : \mathtt{Bool}}{\delta \longmapsto_{Expr}\ \mathtt{NOT}\ exp : \mathtt{Bool}}$$

As usual, the intended meaning of this system is that $\mathrm{PL}_0^{\mathrm{R}}$ constructs are statically well-formed if and only if this property can be proved using the rules.

In order to model the static semantics by rewrite rules, we first introduce a "well-formed" operator:

```
wf_expr:  Expr, Dict, Type_ -> Bool
```

where `Type_` contains `_INTEG_` and `_BOOL_`, modelling `integ` and `Bool`, the possible types of expression. The corresponding operator for variables is not needed since the definition of the domain *Var* has been expanded into the definition of *Expr*.

The deduction rules are always used in a backwards way to prove static correctness of given complex expressions from the static correctness of simpler components and not in the other (forward) direction. Moreover, there are no extra variables in the premise of the deduction rules. Therefore a set of inference rules

$$\frac{premise_1}{\delta \longmapsto_{Expr}\ exp : tp} \quad \cdots \quad \frac{premise_n}{\delta \longmapsto_{Expr}\ exp : tp}$$

that describes the several possibilities[1] for an expression *exp* to be well-typed with type *tp* can be modelled by the rewrite rule

```
wf_expr(exp, delta, tp_) -> premise_1 | ...| premise_n.
```

where `premise`$_i$ is the representation of $premise_i$.

For the inference rules from above, this results in the following rules ('.' is the application operator; `idf` and `i` are the typical variables for *Name* and *Int*, respectively):

```
wf_expr(mk_Expr(idf), delta, _INTEG_) -> ((delta . idf) = VarInt)
wf_expr(mk_Expr(i), delta, _INTEG_)
wf_expr(mk_Expr(exp1, _PLUS, exp2), delta, _INTEG_) ->
   wf_expr(exp1, delta, _INTEG_) & wf_expr(exp2, delta, _INTEG_)
```

The first rule results from expansion of *var* in the *exp* productions. In order to capture the intended equivalence, some more rules are needed. For otherwise, with only the rules from above, a term like

```
wf_expr(mk_expr(42), delta, _BOOL_)
```

is irreducible, even though it should reduce to `false`. Therefore, all the cases concerning the "wrong types" have also to be added, e. g.

[1]For most kinds of expressions, there is exactly one such rule.

```
wf_expr(mk_Expr(idf), delta, _BOOL_) -> false
wf_expr(mk_Expr(i), delta, _BOOL_) -> false
wf_expr(mk_Expr(exp1, _PLUS, exp2), delta, _BOOL_) -> false
```

10.2.3 Operational Semantics

The operational semantics of an expression depends on the actual operational environment, recording the memory locations corresponding to the identifiers, and on the machine state, containing the information which values are stored at the locations. Values may be integers, Booleans, or the special element `error` denoting faulty evaluation:

$$\begin{array}{lll} \mathit{val} \in \mathsf{Val} & =_{\mathit{df}} & \mathsf{Integer} \uplus \mathsf{Bool} \uplus \{\mathtt{error}\} \\ \sigma \in \Sigma & =_{\mathit{df}} & \mathsf{Loc} \rightarrow \mathsf{Integer} \\ \rho \in \mathsf{OpEnv} & =_{\mathit{df}} & \mathit{Name} \rightarrow (\mathsf{Loc} \uplus \mathsf{Loc}^*) \end{array}$$

The transition system for the semantics of expressions is defined relative to a given static environment δ:

$$\begin{array}{l} \Gamma^{\delta}_{\mathit{Expr}} =_{\mathit{df}} \\ \quad \{\langle\, \mathit{exp}, \sigma \,\rangle \in \mathit{Expr} \times \Sigma \mid \exists \mathit{tp} \in \mathit{Tp} : \delta \vdash_{\mathit{Expr}} \mathit{exp} : \mathit{tp}\} \\ \quad \cup\ \mathsf{T}_{\mathit{Expr}} \\ \mathsf{T}_{\mathit{Expr}} =_{\mathit{df}} \mathsf{Val}. \end{array}$$

The semantics of expressions, however, does not depend on the static environment. It is only used to restrict the set of configurations to those containing well-typed expressions. The transition relation is based on the interpretation α that assigns meanings to the language primitives. The defining rules depend on an operational environment ρ; some examples are:[2]

(EO1) $\rho \vdash_{\delta} \langle\, \mathit{int}, \sigma \,\rangle \longrightarrow_{\mathit{Expr}} \alpha(\mathit{int})$

(EO4) $\rho \vdash_{\delta} \langle\, \mathit{name}, \sigma \,\rangle \longrightarrow_{\mathit{Expr}} \sigma(\rho(\mathit{name}))$

(EO6) $$\frac{\rho \vdash_{\delta} \langle\, \mathit{exp}, \sigma \,\rangle \longrightarrow_{\mathit{Expr}} \mathit{val} \wedge \mathit{val} \in \mathsf{Val} \setminus \{\mathtt{error}\} \mid \mathtt{error}}{\rho \vdash_{\delta} \langle\, \mathit{mop}\ \mathit{exp}, \sigma \,\rangle \longrightarrow_{\mathit{Expr}} \alpha(\,\mathit{mop}\,)(\mathit{val}) \mid \mathtt{error}}$$

The last rule is shorthand for two rules, each with identical left-hand sides in the transitions. The various cases for the right-hand sides are separated by '|', the n-th case in the conclusion corresponding to the n-th case in the hypothesis.

10.2.4 Denotational Semantics

Denotational environments and values differ from their operational counterparts by being possibly undefined:

$$\begin{array}{rll} & \mathsf{DenVal} & =_{\mathit{df}}\ \mathsf{Val} \uplus \{\bot_{\mathrm{DenVal}}\} \\ \mathsf{env} \in & \mathsf{DenEnv} & =_{\mathit{df}}\ \mathit{Name} \rightarrow (\mathsf{Loc} \uplus \mathsf{Loc}^* \uplus \{\bot_{\mathrm{DenEnv}}\}) \end{array}$$

[2]As already said before (cf. Section 5.2.1), these rules fulfil the restrictions of the *ptp/t* format since the environments ρ and δ can be integrated in the configurations (see Section 10.2.7 below).

The semantic function $\mathcal{E}$ for expressions is defined in the standard way. Example clauses are:

$$\mathcal{E} : \mathit{Expr} \longrightarrow \mathsf{DenEnv} \longrightarrow \Sigma \longrightarrow \mathsf{DenVal}$$

$$\mathcal{E}[\![\mathit{int}]\!]\,\mathtt{env}\,\sigma =_{df} \alpha(\mathit{int})$$

$$\mathcal{E}[\![\mathit{name}]\!]\,\mathtt{env}\,\sigma =_{df} \begin{cases} \sigma(\mathtt{env}(\mathit{name}))\,, & \text{if } \mathtt{env}(\mathit{name}) \neq \bot \\ \mathtt{error}\,, & \text{otherwise} \end{cases}$$

$$\mathcal{E}[\![\,\mathit{mop}\;\mathit{exp}]\!]\,\mathtt{env}\,\sigma =_{df} \begin{cases} \alpha(\,\mathit{mop}\,)\,(\mathcal{E}[\![\mathit{exp}]\!]\,\mathtt{env}\,\sigma)\,, & \\ \qquad\qquad \text{if } \mathcal{E}[\![\mathit{exp}]\!]\,\mathtt{env}\,\sigma \in \mathsf{Val} \setminus \{\mathtt{error}\} \\ \mathtt{error}\,, \quad \text{otherwise} \end{cases}$$

10.2.5 The Equivalence Theorem

Before defining what is meant by equivalence of the two semantics definitions, first a compatibility relation on environments has to be introduced: Let $\rho \in \mathsf{OpEnv}, \delta \in \mathsf{Dict}$. Then

$$\rho : \delta \;\overset{\text{def}}{\Longleftrightarrow}\; \forall\, \mathit{name} \in \mathit{Name} \;: (\rho(\mathit{name}) \in \mathsf{Loc} \Leftrightarrow \delta(\mathit{name}) = \mathtt{VarInt}) \wedge$$
$$(\rho(\mathit{name}) \in \mathsf{Loc}^* \Leftrightarrow \delta(\mathit{name}) = (\mathtt{ArrayInt}, \#\rho(\mathit{name})))$$

where $\#l$ for a list l is the length of l.

Now the equivalence theorem for expressions can be stated [3]:

Theorem 5.2 from [82] *(expression equivalence)*
For all static environments $\delta \in \mathsf{Dict}$, expressions $exp \in Expr$, operational environments $\rho \in \mathsf{OpEnv}$, states $\sigma \in \Sigma$ such that

$$\rho : \delta \;\wedge\; \exists\; tp \in Tp \;:\; \delta \vdash_{Expr} exp : tp$$

the following two conditions are satisfied:

(1) $\rho \vdash_{\delta} \langle\, exp, \sigma \,\rangle \longrightarrow_{Expr} t$ *iff* $\mathcal{E}[\![exp]\!]\,\tilde{\rho}\,\sigma = t$,
for all $t \in \mathsf{T}_{Expr}$ *and*

(2) $\neg(\exists t \in \mathsf{T}_{Expr} \,.\, \rho \vdash_{\delta} \langle\, exp, \sigma \,\rangle \longrightarrow_{Expr} t)$ *iff*
$\mathcal{E}[\![exp]\!]\,\tilde{\rho}\,\sigma = \bot$.

where $\tilde{\rho}(\mathit{name})$ is $\rho(\mathit{name})$ if this is in $\mathsf{Loc} \cup \mathsf{Loc}^*$ and $\bot$ otherwise. In words: provided the environments ρ and δ are compatible and the expression *exp* is well-typed, the denotational semantics of *exp* is $\bot$ iff the operational semantics does not produce a result (given as a terminal configuration), and it is different from $\bot$ iff this value is also produced operationally.

A sufficient condition for the above theorem is:

$$\begin{aligned} &\forall \delta \in \mathsf{Dict}\; \forall exp \in Expr\; \forall \rho \in \mathsf{OpEnv}\; \forall \sigma \in \Sigma \;: \\ &\;\rho : \delta \;\wedge\; \exists tp \in Tp \;:\; \delta \vdash_{Expr} exp : tp \;\Rightarrow \\ &\;(\mathcal{E}[\![exp]\!]\,\tilde{\rho}\,\sigma \neq \bot \;\;\wedge\;\; \rho \vdash_{\delta} \langle\, exp, \sigma \,\rangle \longrightarrow_{Expr} \mathcal{E}[\![exp]\!]\,\tilde{\rho}\,\sigma) \end{aligned} \tag{10.1}$$

In words: If *exp* is statically well-formed, then operational and denotational semantics yield the same value which is not "undefined", provided the static and dynamic environments are compatible.

[3] Verbatim quotes from [82] will be printed in *italic font* in order to distinguish them from the rest of the text.

In [82], (10.1) is proved by structural induction on *exp*.

10.2.6 The Proof Method

The expression *exp* in (10.1) whose semantics is to be determined is not given explicitly; application of the semantic rules, however, requires information about the semantics of its subexpressions. Therefore the automated proof must be based on a case distinction considering the different possibilities for the structure of *exp*; information about subexpressions must be gained by an inductive argument, just as in the hand proof.

To illustrate how this induction works in connection with the simulating rewrite systems, consider one case of the induction step, viz. that case where *exp* is of the form *mop exp'* for some arbitrary but fixed $exp' \in Expr$ and $mop \in MonadicOp$. If the aspects of compatibility of environments and static semantics in (10.1) are neglected, it remains to prove:

$$\forall \rho \in \mathsf{OpEnv}, \delta \in \mathsf{Dict}, \sigma \in \Sigma : \rho \vdash_\delta \langle\, mop\;\; exp', \sigma \,\rangle \longrightarrow_{Expr} \mathcal{E}[\![mop\; exp']\!]\, \tilde{\rho}\, \sigma \tag{10.2}$$

The first step is to include the environments in the configurations because this gathers related information in a way that is better accessible by rewriting. So the new configuration sets are

$$\begin{aligned}
&\overline{\Gamma}_{Expr} =_{df} \\
&\quad \{\langle\, \rho, \delta, exp, \sigma \,\rangle \in \mathsf{OpEnv} \times \mathsf{Dict} \times Expr \times \Sigma \mid \delta \vdash_{Expr} exp : tp\} \\
&\quad \cup\ \mathsf{T}_{Expr} \\
&\mathsf{T}_{Expr} =_{df} \mathsf{Val}
\end{aligned}$$

which leads to the reformulated goal

$$\forall \rho \in \mathsf{OpEnv}, \delta \in \mathsf{Dict}, \sigma \in \Sigma : \langle\, \rho, \delta, mop\;\; exp', \sigma \,\rangle \longrightarrow_{Expr} \mathcal{E}[\![mop\; exp']\!]\, \tilde{\rho}\, \sigma \quad . \tag{10.3}$$

The induction hypothesis is

$$\forall \rho \in \mathsf{OpEnv}, \delta \in \mathsf{Dict}, \sigma \in \Sigma : \langle\, \rho, \delta, exp', \sigma \,\rangle \longrightarrow_{Expr} \mathcal{E}[\![exp']\!]\, \tilde{\rho}\, \sigma \quad . \tag{10.4}$$

In order to use this hypothesis in a rewrite proof, a corresponding rewrite rule must be found. An obvious choice is

$$\langle\, \rho, \delta, exp', \sigma \,\rangle \;@\; K_{1f} \longrightarrow \langle\, \mathcal{E}[\![exp']\!]\, \tilde{\rho}\, \sigma \,\rangle \;@\; K_{0f} \tag{10.5}$$

where $K_{1f} = \langle\, 1, \mathsf{on}, \ldots, \mathsf{on} \,\rangle$ and $K_{0f} = \langle\, 0, \mathsf{on}, \ldots, \mathsf{on} \,\rangle$ as in Section 7.2. This rule is correct with respect to (10.4), since it only enables transitions that are allowed by the induction hypothesis (note that *exp'* is a constant, not a variable), and hence it may be added to the SOS-derived rules in $\mathcal{R}'$ without destroying its correctness properties.[4] The rewriting proof of (10.3) proceeds in three steps:

[4]The completeness properties are not destroyed, either. This fact will not be needed here, however.

Step 1 $\mathcal{E}[\![exp']\!]\,\tilde{\rho}\,\sigma$ is evaluated using the rules from $\mathcal{B}$ and the denotational rules. This results in some term t_1.

Step 2 Next, the configuration/context term $eval\,(\,\langle\,\rho, \delta, exp', \sigma\,\rangle\ @\ K_{1f}\,)$ is evaluated using all rules in $\mathcal{R}$ including the induction hypothesis (10.5). This results in some term t_2. Note that the 1-context K_{1f} restricts this evaluation to one outermost application of an SOS-derived rule.

Step 3 Finally, equality of t_1 and t_2 is proved with the rules of $\mathcal{B}$.

If the last step succeeds, it has been proved

$$eval\,(\,\langle\,\rho, \delta, exp, \sigma\,\rangle\ @\ K_{1f}\,) \xrightarrow{*}_{\mathcal{R}} t_1 =_E t_2 \ {}_{\mathcal{B}}\!\xleftarrow{*} \mathcal{E}[\![exp']\!]\,\tilde{\rho}\,\sigma \quad .$$

The *eval* operator is only used in connection with SOS-derived rules. Hence, t_2 cannot contain this operator, and there must be a point in the above sequence where the outermost *eval* is removed. Due to the special context elimination rules (see Section 3.3.1), this means that there is some term t_3 such that

$$eval\,(\,\langle\,\rho, \delta, exp, \sigma\,\rangle\ @\ K_{1f}\,) \xrightarrow{*}_{\mathcal{R}} eval\,(\,t_3\ @\ K_{0f}\,) \rightarrow_{\mathcal{B}} t_3 \xrightarrow{*}_{\mathcal{B}} t_1$$

and therefore $\langle\,\rho, \delta, exp, \sigma\,\rangle\ @\ K_{1f} \xrightarrow{*}_{\mathcal{R}} t_3$. By one-step correctness (7.17) and correctness of (10.5), it follows that

$$\langle\,\rho, \delta, exp, \sigma\,\rangle \rightarrow_{\mathcal{S}} t_3$$

and since $\xleftrightarrow{*}_{\mathcal{B}}\ \subseteq\ =_E$ (see Requirement 4.6) and SOS rules are applied modulo $=_E$, finally

$$\langle\,\rho, \delta, exp, \sigma\,\rangle \rightarrow_{\mathcal{S}} \mathcal{E}[\![exp']\!]\,\tilde{\rho}\,\sigma \quad .$$

10.2.7 Comparison of Automated Proof and Hand Proof

The first step in automating the proof of (10.1) is to reformulate the goal in the term representation used. Starting from (10.3) and introducing the *eval* operator needed to model the SOS transition relation (cf. Sections 3.3.1 and 7.2), the goal can be stated as

```
prove
  (compatible(rho, delta) & wf_expr(exp, delta, tp_))
  =>
  (is_Val(dsem(exp, retrieve(rho), sigma)) &
   ( eval(mk_ExtCf(rho, delta, mk_Config(exp, sigma)) @ K1f)
      =
      mk_ExtCf(rho, delta,
               mk_Config(to_Val(dsem(exp, retrieve(rho), sigma)))) ))
  by induction on exp
..
```

where `K1f` is an abbreviation for a constant context (altogether there are 12 SOS rules):

```
set name context
assert
  K1f -> mk_Context(_ONE, _ON, _ON, _ON, _ON, _ON, _ON,
                         _ON, _ON, _ON, _ON, _ON, _ON)
..
```

`compatible(rho, delta)` models $\rho : \delta$ and `wf_expr(exp, delta, tp_)` stands for $\delta \vdash_{Expr} exp : tp$. Note that the existential quantifier in (10.1) can be moved to the outside and turned into a universal quantifier since *tp* only occurs in the hypothesis. In the term representation, the environments are included in the non-terminal configurations. Using the input syntax for the program **gensig** that generates most of the basic rewrite system (see Appendix A.1), the domain equations for configurations can be written as follows (the domain constructor $\times$ is written as `*`):

```
ExtCf   = OpEnv * Dict * Config | Val
Config  = Expr * Store
```

The inductive proof of Lakhneche starts with

Case exp is a constant symbol:
This is the case when

$$exp = int \lor exp = \texttt{TRUE} \lor exp = \texttt{FALSE}.$$

In this case, rule (EO1), (EO2), (EO3) and the definition of $\mathcal{E}$ yield

$$\rho \vdash_\delta \; < exp, \sigma > \longrightarrow_{Expr} \alpha(exp) \text{ and}$$

$$\mathcal{E}[\![exp]\!]\,\tilde{\rho}\,\sigma = \alpha(exp).$$

So it merely indicates what definitions are to be applied to calculate the desired result. With LP, this step is slightly simpler since rewrite rules are applied automatically unless indicated otherwise. The SOS-derived rules, however, are kept passive in order to speed up the proof (see below for the effect of this optimization); so they have to be made active again at the proper places. The same holds for the rule defining the abbreviation `K1f`.

```
% ==================================================
% case exp = mk_Expr(i) [INT]

make active context
make active EO1

% ==================================================
% case exp = mk_Expr(b) [Bool]

make active context
make active EO2*
```

The third case of the induction basis is similarly simple in [82]:

Case $exp = name$:
This implies by rule (EO4) that

$$\rho \vdash_\delta \; < exp, \sigma > \longrightarrow_{Expr} \sigma(\rho(name))$$

$\rho(name) \in \mathsf{Loc}$ holds since $\rho : \delta$ and $\delta \vdash_{Expr} exp : tp$. This implies

$$\begin{array}{rcll} \mathcal{E}[\![exp]\!]\,\tilde{\rho}\,\sigma & = & \sigma(\tilde{\rho}(name)) & [\textit{Definition of } \mathcal{E}] \\ & = & \sigma(\rho(name)). & [\rho : \delta \textit{ and } 4.8] \end{array}$$

The LP proof for this case is more complicated because it requires some guidance. The proof script containing the proof commands looks as follows:

```
% ===================================================
% case exp = mk_Expr(i13) [Idf]

resume by induction on tp_
```

Since the sort `Type_` is declared to be generated by the constants `_INTEG_` and `_BOOL_`, induction on this sort is nothing else than a complete case distinction with the cases `tp_ = _INTEG_` and `_BOOL_`.

```
% ---------------------------------------------------
% Case tp_ = _INTEG_:

resume by =>
% new constants: deltac (delta), rhoc (rho), i13c (i13)
```

If the conjecture is an implication, it is mostly advisable to proceed by the `=>-method`. This should not be done, however, if the hypothesis of the implication contains a variable that is subsequently to be used for induction. The `=>-method` transforms all variables occurring in the hypothesis into constants, and there is no direct way in LP to prove a conjecture by induction on a non-variable term.

```
set immunity ancestor
instantiate idf by i13c in theorem5_2*Hyp
set immunity off
```

After extracting the hypothesis from the goal, the compatibility precondition is instantiated for `i13c`.[5] Here, the instantiated formula is made "ancestor-immune", which means that it can be rewritten by all other rules except for the rule from which it descended (without being immune, this rule would immediately normalize it to `true`).

Next, some lemmas have to be proved that are needed to decide the conditions that are part of the semantics definitions. In the hand proof, these lemmas are hidden in the phrases *"$\rho(name) \in \mathsf{Loc}$ holds since $\rho : \delta$"* and *"this implies ...[definition of $\mathcal{E}$]"*. (The first lemma is declared immune because it would otherwise be normalized to `true`.)

```
% ---------------------------------------------------
% Lemmas for case Idf:

set name lemma7
set immunity on
prove
```

[5]The names for the components of *exp* are generated automatically by LP. When variables are turned into constants, e. g. as the result of applying the `=>-method`, LP usually appends a `c` to the names of variables.

```
  is_Loc(retrieve(rhoc) . i13c)
..
set immunity off

[] % lemma 7

% ----------------------------------------
set name lemma8
prove
  not((retrieve(rhoc) . i13c) = _BOTTOM_LOC)
by contradiction
..
% This lemma follows from lemma 7; but since lemma7 is immune, the required
% rewriting steps must be stated explicitly:
normalize lemma7 with lemma*hyp
normalize lemma7 with GLoc1

[] % lemma 8
% ----------------------------------------
```

After the lemmas are proved, it suffices to activate the SOS-derived rule for this case to calculate that both semantics definitions yield the same result. (The second case is proved automatically; `wf_expr(mk_Expr(i13), delta, _BOOL_)` is rewritten to `false` by the static semantics rules).

```
make active context
make active E04

% case _BOOL_ is automatic

[] % case exp = mk_Expr(i13) [Idf]
```

As an example for the induction step, consider the case $exp = \, mop\, exp_1$. The hand proof reads as follows:

Case $exp = mop\ exp_1$:

By induction hypothesis a value val exists such that

$$\rho \vdash_\delta \ < exp_1, \sigma > \ \longrightarrow_{Expr} val \ \textit{and}$$

$$\mathcal{E}[\![exp_1]\!]\,\tilde{\rho}\,\sigma \ = \ val.$$

If $val = \texttt{error}$ *then by rule (EO6) and by the definition of* $\mathcal{E}[\![\ ]\!]$,

$$\rho \vdash_\delta \ < exp, \sigma > \ \longrightarrow_{Expr} \texttt{error} \ \textit{and}$$

$$\mathcal{E}[\![exp]\!]\,\tilde{\rho}\,\sigma \ = \ \texttt{error}.$$

If $val \neq \texttt{error}$ *then rule (EO6) and the denotational semantics yield*

$$\rho \vdash_\delta \ < exp, \sigma > \ \longrightarrow_{Expr} \alpha(\ mop\ )\, val \ \textit{and}$$

$$\begin{aligned} \mathcal{E}[\![exp]\!]\,\tilde{\rho}\,\sigma \ &= \ \alpha(\ mop\ )\,\mathcal{E}[\![exp_1]\!]\,\tilde{\rho}\,\sigma \\ &= \ \alpha(\ mop\ )\, val. \end{aligned}$$

The structure of the automated proof is quite similar:

```
% ==================================================
% case exp = mk_Expr(m, expc)  [Mop * Expr]

resume by induction on tp_

% --------------------------------------------------
% case tp_ = _INTEG_

instantiate tp_ by _INTEG_ in wf_lemma

resume by =>
% new constants: rhoc, deltac, mc

% enable induction hypothesis
instantiate rho by rhoc, delta by deltac, tp_ by _INTEG_ in *hyp

make active context
make active E06

resume by case to_Val(dsem(expc, retrieve(rhoc), sigma)) = _ERROR

[] % case tp_ = _INTEG_

% --------------------------------------------------
% case tp_ = _BOOL_

instantiate tp_ by _BOOL_ in wf_lemma

resume by =>

% enable induction hypothesis
instantiate rho by rhoc, delta by deltac, tp_ by _BOOL_ in *hyp

make active context
make active E06

resume by case
  to_Val(dsem(expc, retrieve(rhoc), sigma)) = _ERROR
..
```

The only additional complication is the need to give explicit instantiations for the induction hypothesis. This is not done automatically because this hypothesis is an implication. The rest of the proof exhibits the same level of detail as the hand proof.

Note that there was no need to refer to the rather complicated structure of the SOS-derived rules. All case distinctions have counterparts in the hand proof; the only time that contexts occur explicitly is in the statement of the goal itself. The SOS rewriting system works completely in the background, and hence the user is not disturbed or confused by the large intermediate terms developing during semantics calculations.[6]

[6]This is only true, however, if the basic rewriting system is capable of deciding all the if conditions occurring in the intermediate terms, see Requirement 4.6.

The LP proof script could be made even simpler by letting the SOS-derived rules be active throughout the proof, allowing LP to apply them automatically whenever this is possible. The disadvantage of this simplification of the proof script is that the proof itself as performed by LP becomes more time consuming as can be seen in the respective results of the `statistics` command. If the SOS-derived rules are activated manually, it produces the following output (on a Sun SparcStation 10/40):

```
Recent          Success            Failure          Total
------       Count     Time     Count     Time       Time
Ordering       200     0.05         0     0.00       0.05
Rewriting     1930    19.29     12805    17.11      36.40
Deductions     155     2.57      1784     0.80       3.37
Unification     14     0.04         8     0.02       0.06
Prover                                            1:34.69
GC's             9
Total time                                        3:22.31

Heap size  =     733,395 words
```

and without manual activation

```
Recent          Success            Failure          Total
------       Count     Time     Count     Time       Time
Ordering       176     0.06         0     0.00       0.06
Rewriting     2973    47.73      9412    23.17    1:10.90
Deductions     155     4.92      1580     0.61       5.53
Unification     14     0.03         8     0.01       0.04
Prover                                            1:37.67
GC's            10
Total time                                        4:07.66

Heap size  =     743,323 words
```

(All times are in measured in seconds; a "word" consists of four bytes.) If the SOS-derived rules are always active, LP applies them as early as possible, thus generating large intermediate terms early. These terms contain subterms that are again redices but would be eliminated by application of selector operators later. This explains the increase of successful rewritings by 50 %. Furthermore, it takes more time to handle these large terms.

So the general strategy should be to apply the SOS-derived rules as lately as possible.

Both the above proofs started from a "frozen" basic rewriting system, i. e. a system that had already been processed by LP and written to a file. To set up the basic system, LP needs additional `1:36.02` minutes.

The proof scripts show that the structure of hand proof and automated proof are very similar. The main advantage of a proof tool is its ability to perform simple calculations. This can be seen in those parts that merely consist of applications of definitions: here the automated proofs are simpler than the hand proof.

LP is not able of checking proofs "by handwaving", however. So those parts of the hand proofs that are just a sketch, consisting only of a collection of the relevant facts without a detailed description of their connection, have to be spelled out completely (e. g. in the two lemmas of the script above). This can often lead to proof scripts that are a lot more complicated than the hand proof, in particular containing a lot of instantiations.

10.3 Semantics Equivalence for Sequential Processes

The language of $\mathrm{PL}_0^{\mathrm{R}}$ sequential processes is larger than the expression language presented in the previous section. In particular, expressions themselves may occur inside processes, and hence the equivalence proof for processes relies on that for expressions. Sequential processes are generated according to the following grammar:

$$sproc \in SeqProc$$

$$\begin{aligned} sproc \quad ::= \quad & \mathtt{SKIP} \mid \mathtt{STOP} \mid var \ := exp \mid \mathtt{INPUT?}name \mid \mathtt{OUTPUT!}exp \mid \\ & \mathtt{SEQ}\,[sproc_1, \ldots, sproc_n] \mid \mathtt{IF}\,[gc_1, \ldots, gc_n] \mid \mathtt{WHILE}(exp, sproc) \mid \\ & \mathtt{CALL}(name) \end{aligned}$$

where $n \in \mathbb{N}$ and gc is a "guarded command":

$$gc \in Guarded$$

$$gc \ ::= exp \rightarrow sproc$$

These guarded commands are not to be mistaken for those of Dijkstra [34] often written with the same syntax. Dijkstra defines a nondeterministic semantics for his IF, whereas the semantics of the IF defined here is deterministic (in accordance with the Occam [75] IF construct; see below).

The static semantics of processes is defined in the same style as that of expressions (cf. Section 10.2.2) by defining a predicate $\delta \vdash_{SeqProc} sproc$.

10.3.1 Operational and Denotational Semantics

The operational semantics OS of sequential processes is given by the labelled transition system

$$(\Gamma^{\delta}_{SeqProc}, \mathsf{T}^{\delta}_{SeqProc}, \mathsf{A}, \rightarrow_{SeqProc})$$

where

$$\Gamma^{\delta}_{SeqProc} =_{df} \{< sproc, \sigma > \mid \delta \vdash_{SeqProc} sproc \ \wedge \ \sigma \in \Sigma\} \cup \mathsf{T}^{\delta}_{SeqProc} \quad ,$$

$$\mathsf{T}^{\delta}_{SeqProc} =_{df} \{\mathtt{terminated}, \mathtt{stopped}, \mathtt{invalid}\} \times \Sigma \quad ,$$

$$\mathsf{A} \stackrel{\mathrm{def}}{=} \mathsf{Com} \cup \{\tau\} \quad \text{, and}$$

$$\mathsf{Com} \overset{\text{def}}{=} \{\mathtt{input} \cdot k,\ \mathtt{output} \cdot k \mid k \in \mathsf{Integer}\}.$$

Terminal configurations contain a state and the information how this state has been reached: regularly terminating (`terminated`), irregularly terminating (`stopped`) or as the result of a run-time error (`invalid`). Labels can be either communications with the environment, or the silent move τ.

Some of the rules defining the transition relation $\rightarrow_{SeqProc}$ are the following (we always assume that static environment $\delta \in \mathsf{Dict}$ and dynamic environment $\rho \in \mathsf{OpEnv}$ are compatible):

Rule (OP1)

$$\rho \vdash_\delta \ < \mathtt{SKIP}, \sigma > \ \xrightarrow{\tau}_{SeqProc} \ < \mathtt{terminated}, \sigma >$$

Rule (OP10) For $n \geq 1$:

$$\frac{\rho \vdash_\delta \ < exp_1, \sigma > \ \longrightarrow_{Expr} \mathsf{tt} \mid \mathsf{ff} \mid val \notin \mathsf{Bool}}{\rho \vdash_\delta \ < \mathtt{IF}[exp_1 \rightarrow sproc_1, gc_2, \ldots, gc_n], \sigma > \ \xrightarrow{\tau}_{SeqProc} \ \begin{array}{l} < sproc_1, \sigma > \ \mid \\ < \mathtt{IF}[gc_2, \ldots, gc_n], \sigma > \ \mid \\ < \mathtt{invalid}, \sigma > \end{array}}$$

Rule (OP12)

$$\frac{\rho \vdash_\delta \ < exp, \sigma > \ \longrightarrow_{Expr} \mathsf{tt} \mid \mathsf{ff} \mid val \notin \mathsf{Bool}}{\rho \vdash_\delta \ < \mathtt{WHILE}(exp, sproc), \sigma > \ \xrightarrow{\tau}_{SeqProc} \ \begin{array}{l} < \mathtt{SEQ}[sproc, \mathtt{WHILE}(exp, sproc)], \sigma > \ \mid \\ < \mathtt{terminated}, \sigma > \ \mid \\ < \mathtt{invalid}, \sigma > \end{array}}$$

As already mentioned in Section 6.3.1, the denotational semantics $\mathcal{C}[\![sproc]\!]$ is a state transformation also depending on the denotational environment. (For the definition of the states, see also Section 6.3.1, page 74.) The function $\mathcal{C}$ is defined as

$$\mathcal{C} : SeqProc \longrightarrow \mathsf{DenEnv} \longrightarrow \mathsf{StatTr} \quad .$$

The clauses corresponding to the operational rules from above are

$$\mathcal{C}[\![\mathtt{SKIP}]\!]\,\mathsf{env}\, st =_{df} st$$

$$\mathcal{C}[\![\mathtt{IF}[exp_1 \rightarrow sproc_1, gc_2, \ldots, gc_n]]\!]\,\mathsf{env}\, st$$

$$=_{df} \begin{cases} \mathcal{C}[\![sproc_1]\!]\,\mathsf{env}\, st & , \text{if } \mathcal{E}[\![exp_1]\!]\,\mathsf{env}\,\sigma = \mathsf{tt} \\ \mathcal{C}[\![\mathtt{IF}[gc_2, \ldots, gc_n]]\!]\,\mathsf{env}\, st & , \text{if } \mathcal{E}[\![exp_1]\!]\,\mathsf{env}\,\sigma = \mathsf{ff} \\ < \mathtt{invalid}, in, tr > & , \text{if } \mathcal{E}[\![exp_1]\!]\,\mathsf{env}\,\sigma \notin \mathsf{Bool} \end{cases}$$

So the guarded commands in an `IF` are treated as a list: the first command whose condition is true is executed. In Dijkstra's language, on the other hand, any command whose condition is fulfilled can be selected for execution (not just the first one).

$$\mathcal{C}[\![\mathtt{WHILE}(exp, sproc)]\!]\,\mathsf{env} =_{df} \mu\Phi_{exp, sproc, \mathsf{env}} \quad \text{where}$$

$$\Phi_{exp,sproc,\mathsf{env}} : \mathsf{StatTr} \longrightarrow \mathsf{StatTr}$$

$$\Phi_{exp,sproc,\mathsf{env}}(Tr)st =_{df} st \quad \text{for all } st \in \mathsf{Inv}$$

and for all $st =< \sigma, in, tr > \in \mathsf{State} \setminus \mathsf{Inv}$

$$\Phi_{exp,sproc,\mathsf{env}}(Tr)st =_{df} \begin{cases} st & \text{, if } \mathcal{E}[\![exp_1]\!]\,\mathsf{env}\,\sigma = \mathsf{ff} \\ < \mathtt{invalid}, in, tr > & \text{, if } \mathcal{E}[\![exp_1]\!]\,\mathsf{env}\,\sigma \notin \mathsf{Bool} \\ (Tr \circ \mathcal{C}[\![sproc]\!]\,\mathsf{env})\,st & \text{, otherwise} \end{cases}$$

10.3.2 The Structure of the Equivalence Proof in [82]

The structure of the manual equivalence proof is rather complex; some "intermediate semantics" are defined in order to close the gap between OS and $\mathcal{C}$. Instead of repeating the full definitions here, rather the ideas behind the several steps of the proof will be explained. In particular, the equivalence predicate itself will not be defined in a fully formal way; for details, see [82].

Step 0 Definition of the equivalence predicate $\equiv$.

OS and $\mathcal{C}$ are equivalent iff the following holds: If a process *sproc* terminates after having produced some input/output actions, then the trace component of its denotational semantics $\mathcal{C}[\![sproc]\!]$ also includes exactly these actions in the same order. If the termination was not regular, then this is also recorded in $\mathcal{C}[\![sproc]\!]$. If *sproc* does not terminate, then $\mathcal{C}[\![sproc]\!]$ is an infinite trace containing the input/output actions produced operationally.

Step 1 Definition of an unlabelled SOS system *TrS* defining an operational semantics *TrOS* equivalent to OS.

This step uses the same idea as Lemma 5.2: actions are recorded in an additional trace component of the configurations.

The main problem in the proof of $\equiv$ is the occurrence of mutually recursive procedures which prevents application of ordinary structural induction. Lakhneche overcomes this problem by introducing upper limits for the number of executions of the bodies of loops (denoted by i) and for the number of successive procedure calls (denoted by j): instead of *sproc*, the "indexed sequential process" $sproc^{(i,j)}$ is used.

Step 2 Definition of an SOS system ω-*TrS* defining an operational semantics ω-*TrOS* for indexed processes.

TrOS and ω-*TrOS* are proved to be equivalent: if *TrS* yields a terminal configuration for some process *sproc*, then there are some finite limits for the numbers of executions of loops and procedure calls. These limits can be used as indices for *sproc*, and ω-*TrOS* applied to this indexed process also terminates with the same result.

ω-*TrOS* always produces finite transition sequences, and it is monotonic with respect to indices: if the limits are increased, the resulting traces only become longer.

Step 3 Definition of a denotational semantics $\omega\text{-}\mathcal{C}_\rho$ for indexed processes (depending on an operational environment ρ).

Indexed processes only have finite executions. Therefore, $\omega\text{-}\mathcal{C}_\rho$ can be defined without the use of fixed points; loops and procedure calls are unfolded as often as allowed by the indices.

Step 4 $\omega\text{-}\mathit{TrS}$ is proved to be correct with respect to $\omega\text{-}\mathcal{C}_\rho$.

"Correct" here means that a transition in $\omega\text{-}\mathit{TrS}$ does not change the $\omega\text{-}\mathcal{C}$ semantics of the configurations, i. e. for all $\omega\text{-}\mathit{TrS}$ configurations $\omega\text{-}\gamma$ and $\omega\text{-}\gamma'$ must hold that if $\omega\text{-}\gamma \rightarrow_{\omega\text{-}\mathit{TrS}} \omega\text{-}\gamma'$, then also $\omega\text{-}\mathcal{C}_\rho[\![\omega\text{-}\gamma]\!] = \omega\text{-}\mathcal{C}_\rho[\![\omega\text{-}\gamma']\!]$.

From steps 2 and 4 follows that $\omega\text{-}\mathcal{C}$ is also monotonic.

Step 5 Proof that $\mathcal{C}[\![\mathit{sproc}]\!]$ is the least upper bound of $\omega\text{-}\mathcal{C}_\rho[\![\langle\, \mathit{sproc}^{(i,j)}, \ldots \rangle]\!]$ (with some additional parameters).

With step 5 completed, the proof is finished. In step 4, the correspondence of the appropriate operational and denotational semantics is proved, and the other steps are needed to transform the given definitions into the special indexed versions for which step 4 can succeed.

10.3.3 Automating the Proof

In steps 2 to 5 above, several different kinds of proofs have to be performed. There are steps that require a large amount of formalization and use ingenuous ideas, but not much simple calculation. On the other hand, there are proofs that mainly consist of case distinctions, inductive proofs and straightforward calculation using the semantics definitions, and there are proofs whose complexity lies in between the extremes.

One of the most important purposes of a proof support tool is to assist in proofs of the second kind. When done by hand, these are often considered as routine work and not spelled out completely. So there is a danger of careless mistakes, e. g. when an analogy that is assumed does not exist in the required form. With a proof tool, however, it can easily be checked if a subproof is analogous to another by simply repeating it the for the case in question (possibly with small modifications). Moreover, these proofs are more appropriate for automating than those of the first kind because they usually require less complex formalizations.

Among the steps from above, there are two that contain longish proofs of the kind just described, mainly based on an induction on the structure of processes.

10.3.3.1 Automating a Part of Step 4

In Step 4, Lakhneche states the following lemma:

Lemma 6.33 (from [82])
Let $\delta \in \mathsf{Dict}$ *be a static environment,* ω-γ, ω-$\gamma' \in \Gamma^{\delta}_{\omega\text{-}TrS}$ *be configurations, and* $\rho \in \mathsf{OpEnv}$ *an operational environment such that*

$$\rho : \delta \quad \textit{and} \quad \delta \vdash_{SeqProc} \omega\text{-}\gamma .$$

Then

$$\rho \vdash_{\delta} \omega\text{-}\gamma \rightarrow_{\omega\text{-}TrS} \omega\text{-}\gamma' \quad \textit{implies} \quad \omega\text{-}\mathcal{C}_{\rho}[\![\omega\text{-}\gamma]\!] = \omega\text{-}\mathcal{C}_{\rho}[\![\omega\text{-}\gamma']\!] .$$

The proof of this lemma extends over five pages of mathematical text. But one should note that many auxiliary definitions and lemmas also contribute to the size of this proof; in particular, the additional semantics definitions have to be mentioned.

The first step in formalizing Lemma 6.33 for LP is to model these semantics definitions. The language of indexed sequential processes is built on top of that of sequential processes already defined above:

$$\omega\text{-}sproc \ \in \omega\text{-}SeqProc$$

$$\begin{aligned} \omega\text{-}sproc \quad ::= \quad & \mathtt{SKIP}^q \mid \mathtt{STOP}^q \mid (var\ := exp)^q \mid (\mathtt{INPUT?}name)^q \mid (\mathtt{OUTPUT!}exp)^q \mid \\ & \mathtt{SEQ}\,[sproc_1, \ldots, sproc_n]^q \mid \mathtt{SEQ}\,[\omega\text{-}sproc, sproc_2, \ldots, sproc_n]^q \mid \\ & \mathtt{IF}\,[gc_1, \ldots, gc_n]^q \mid \mathtt{WHILE}(exp, sproc)^q \mid \mathtt{CALL}(name)^q \end{aligned}$$

where $q \in \mathbb{N}^2$, $name \in Name$, $sproc$, $sproc_1, \ldots, sproc_n \in SeqProc$, $exp \in Expr$, $gc_1, \ldots, gc_n \in Guarded$. An equivalent definition is

$$\omega\text{-}sproc \quad ::= \quad sproc^q \mid \mathtt{SEQ}\,[\omega\text{-}sproc, sproc_2, \ldots, sproc_n]^q \quad .$$

This language is much larger than the expression language of Section 10.2 and its semantics definitions require additional domains not needed for the expression case. Moreover, the SOS and denotational definitions are larger and more involved. As a result of this increase in complexity, the rewrite system required for formalizing and proving the lemma is more than twice as large as that for the proof in the previous section (about 1200 rules compared to about 500).

Formalizing the lemma is fairly simple:

```
prove
  ( compatible(rho, delta) & wf_ocf(ocf1, delta) &
    trans(rho, delta, ocf1, ocf2) )
  =>
  ( sdsem(ocf1, rho) = sdsem(ocf2, rho) )
by induction on ocf1
..
```

where `ocf1` and `ocf2` are variables for the sort of configurations `oExtCf` used in ω-TrS, and the operator `trans` is used to abbreviate assertions about the transition relation:

```
declare operator
  trans : OpEnv, Dict, oConfig, oConfig -> Bool
..

set name trans
assert
  trans(rho, delta, ocf1, ocf2) ->
    (eval(mk_oExtCf(rho, delta, ocf1) @ oK1f):oExtCf
     = mk_oExtCf(rho, delta, ocf2))
..
```

The context `ok1f` corresponds to `K1f` that was used earlier (see Section 10.2.7); these contexts differ in their length since the SOS systems have different numbers of rules. The `eval` term has to be qualified with its sort `oExtCf` in order to pass LP's type checker.

The automated proof has been developed in a way similar to the proof described in Section 10.2. The main difference is that, due to the larger size of the problem and the more complicated semantics definitions, more explicit guidance is needed. This guidance, however, could in most cases be easily developed from the original hand proof. The proof shall not be described in detail here, because this would require the explanation of many auxiliary definitions. Instead, some specific aspects shall be mentioned.

Passive rules No attempt is made to complete the rewriting system, because for a system consisting of more than 1200 rewrite rules this would have taken far too much time.[7] As a consequence, there are situations during the proof where certain rules are not allowed to be applied because they would have lead either to subgoals not provable with the current rewrite system or only at too great a cost. In order to deal with this problem, some rules have to be declared as "passive" so that they can only be applied by explicit request.

Other rules are kept passive because they defined an abbreviation, e. g. the definition of the `trans` operator above. As a general rule, such rules should be applied as late as possible in order to keep terms smaller and more comprehensible.[8]

Ordering newly generated equations LP does not support conditional rewriting. Therefore, an implication

$$b \;\Rightarrow\; l = r \tag{10.6}$$

is only represented as a rewrite rule in the form

$$\texttt{(b => l = r) -> true} \tag{10.7}$$

In particular, `l = r` is not considered as a *directed* equation, and there is no way to indicate the direction since `->` may be used only once in a rewrite rule.[9] This may lead to a problem, if for

[7]Note that the fundamental problem concerning completion described in Section 7.3.2 does not exist here because $\mathrm{PL}_0^{\mathrm{R}}$ is deterministic.

[8]Thus by using the `passive` feature of LP, a weak form of strategies can be implemented.

[9]The situation described is true for the current version 2.4 of LP. In future versions, it may provide ways to state that equations shall only be directed in one way.

an implication of the form (10.7) the precondition `b` can be rewritten to `true`. Then normally the resulting equation is ordered in such a way that LP still can guarantee termination of the rewriting system. If it is important to have the rule `l -> r` resulting from this situation, an ordering method must be used that calculates the appropriate direction of the rule from weights that have been assigned to the operators included.[10] As a rule of thumb, it suffices to assign the highest weight `top` to the outermost operator of `l` or the lowest weight `bottom` to that of `r` to reach the desired direction. This procedure has to be applied rather often during the proofs with LP when implications are used in formulas (in particular in inductive goals).

Repetition of proofs Just as the proof of the previous section, the proof of Lemma 6.33 contains many parts that follow exactly the same pattern. If the first proof in such a row has been completed with the help of LP, the other similar proofs can very often simply be copied from the first one (with some obvious modifications). Thus proofs that are claimed to be "analogous" in the hand proof can easily be verified explicitly.

Splitting the proof Due to the definition of ω-*SeqProc*, the automated proof of Lemma 6.33 can be split into two parts. The first one deals with the cases for

$$\omega\text{-}\mathit{sproc} ::= \mathit{sproc}^q$$

and the second one with those for

$$\omega\text{-}\mathit{sproc} ::= \mathtt{SEQ}\,[\omega\text{-}\mathit{sproc}, \mathit{sproc}_2, \ldots, \mathit{sproc}_n]^q \quad .$$

In the first part, only the semantics definitions for indexed ordinary sequential processes are needed, and in the second only those for the other kind. Each of the cases requires only a part of the whole semantics definition, in particular only a part of the SOS-derived rules, and hence the length of the context tuples can be reduced. Since the size of terms not only affects their readability, but also the performance of LP's rewriting mechanisms, this splitting results in a speedup of the development of the proof.

Statistics The total length of the proof scripts for the proof of Lemma 6.33 is about 1100 lines for the first part and about 1400 lines for the second. Note, however, that both scripts are heavily commented, so it cannot be deduced just from the difference in length that the first proof is simpler. For the execution time of the proofs on a Sun SparcStation 10/40, the `statistics` command reports a total time of 17:52.99 minutes for the first part and 3:31.61 minutes for the second part. The timings for the generation of the basic system are 11:50.17 minutes for the first case and 47.81 seconds for the second.

10.3.3.2 Automating a Part of Step 5

Step 5 contains the following main theorem:

[10]Several of such ordering methods are implemented in LP; see the manual [52] for details.

Theorem 6.41 (from [82])
Let $\delta \in \mathsf{Dict}$, *sproc* $\in$ *SeqProc*, $\rho \in \mathsf{OpEnv}$, $\sigma \in \Sigma$, *in* $\in \mathsf{Input}$, *and* $tr \in \mathsf{Com}^*$ *such that*

$$\rho : \delta \quad and \quad \delta \vdash_{SeqProc} sproc.$$

Then for all $j \geq 0$

$$\mathcal{C}[\![sproc]\!]\,(\vartheta \oplus \Psi^j_{\vartheta,\theta}(\bot)) < \sigma, in, tr > = \bigsqcup_{i \geq 0} \omega\text{-}\mathcal{C}_\rho[\![< sproc^{(i,j)}, \sigma, in, tr >]\!]$$

with ϑ *and* θ *defined as follows*

$$\vartheta(name) \overset{\text{def}}{=} \begin{cases} \rho(name) & , \text{ if } \rho(name) \in \mathsf{Loc} \uplus \mathsf{Loc}^* \\ \bot & , \text{ otherwise} \end{cases}$$

and

$$\theta(name) \overset{\text{def}}{=} \begin{cases} \rho(name) & , \text{ if } \rho(name) \in SeqProc \\ \bot & , \text{ otherwise .} \end{cases}$$

where $\Psi_{\vartheta,\theta}$ *is the functional defined by* $\Psi_{\vartheta,\theta} : \mathsf{DenEnv} \to \mathsf{DenEnv}$

$$\Psi_{\vartheta,\theta}(\mathtt{env}) =_{df} \vartheta[name \mapsto \mathcal{C}[\![\theta(name)]\!]\,\mathtt{env} \mid name \in \operatorname{dom}\theta] \quad .$$

$\oplus$ is the "function overwrite" operator: $f \oplus g =_{df} f[x \mapsto g(x) \mid x \in \operatorname{dom} g]$.

This theorem is proved on seven pages in [82]. One should note, however, that only a part of the resulting cases are treated explicitly: about a third of the cases are (correctly) declared as being analogous to some of the other cases.

Unlike Lemma 6.33, Theorem 6.41 only deals with denotational semantics definitions. Therefore, the problems encountered during the automated proof were partly of a different nature; in particular, reasoning about cpo's and the partially ordered set $\langle\, \mathbb{N}, \leq \,\rangle$ was required.

Formalizing the theorem starts with a slight simplification: instead of the more complicated definition of the functional $\Psi_{\vartheta,\theta}$ above, one can use an equivalent, but simpler definition (the equivalence is easily proved by a case distinction over the different possibilities for $\rho(name)$):

$$\Psi_\rho : \mathsf{DenEnv} \to \mathsf{DenEnv}$$

$$\Psi_\rho(\mathtt{env}) =_{df} \rho[name \mapsto \mathcal{C}[\![\rho(name)]\!]\,\mathtt{env} \mid \rho(name) \in SeqProc]$$

Using this simplification, the goal can be stated as:

```
prove
  ( compatible(rho, delta) & wf_sproc(sproc, delta) )
  =>
  ( dsem(sproc, env_j(rho, n2))
    . mk_State(sigma, in, tr)
     = lub(ch(sproc, n2, sigma, in, tr, rho)) )
  by induction on sproc
..
```

where the operator `env_j` is defined below, and `ch` is used to abbreviate the chain of $\omega\text{-}\mathcal{C}_\rho[\![\ldots]\!]$ results:

```
declare operator
  ch : SeqProc, Nat, Store, Input, Trace, OpEnv -> seqState
..
% This definition has to be passive to keep terms small:
set activity off
assert
  ch(sproc, n2, sigma, in, tr, rho) . n
    -> sdsem(mk_oConfig(mk_oSeqProc(sproc, mk_Index(n, n2)),
                        sigma, in, tr), rho)
..
```

`sdsem` is the representation of the semantic function $\omega\text{-}\mathcal{C}_\rho[\![]\!]$, and `mk_Index` constructs a pair of natural numbers (resp. a representation thereof).

Again, the proof shall not be described in full detail. Most of the remarks about Lemma 6.33 also apply to this proof; but there are also some important points that are special to it.

Passive rules As in the other proof, passive rules are used to deal with non-confluence and to define abbreviations in order to make the proof more comprehensible. An example is `ch` operator from above or the indexed environment `env_j`, defined by

$$\mathtt{env}_j =_{df} \vartheta \oplus \Psi_\rho^j(\bot) \quad .$$

Through most parts of the proof, it is much more convenient to work with the unexpanded form, and therefore the rule

```
set name env_j
assert
  env_j(rho, n) -> theta(rho) + ((psi(rho) ^ n) . bot_de')
..
```

is declared passive and terms `env_j(rho, n)` are only expanded when special properties are needed.[11]

Other rules that have to be passive concern the least fixed point operator. Consider e. g. the "retrieve" function $\tilde{\cdot} : OpEnv \rightarrow \mathsf{DenEnv}$ mapping operational to corresponding denotational environments. In [82], this function is defined by

$$\tilde{\rho} =_{df} \mu\Psi_\rho \quad .$$

The representation of this function within LP reads as follows (`DenEnvTr` is the sort representing transformations of denotational environments):

[11] ϑ is modelled as a function taking an argument from OpEnv in order to reflect its dependence on ρ.

```
declare operators
  psi : OpEnv -> DenEnvTr
  fix : DenEnvTr -> DenEnv
..

set name retrieve
assert
  (psi(rho) . env) . idf ->
     if(is_SeqProc(rho . idf),
        mk_GLoc1(dsem(to_SeqProc(rho . idf), env)),
        to_GLoc1(rho . idf))
..

% "retrieve" is an abbreviation:
set activity off
assert
  retrieve(rho) -> fix(psi(rho))
..

% The fixed point properties must also be passive.
set name fixedpoint
assert
  % basic fixedpoint property of fix(detr)
  fix(detr) -> detr . fix(detr)
..

assert
  % minimality property of fix(detr)
  when
    detr . env = env
  yield
    (fix(detr) . idf) <= (env . idf)
..
set activity on
```

Of course, the rule defining the fixed point property of the operator `fix` must be passive because it is non-terminating, and it may only be used to rewrite terms one step at a time.

Reasoning about orderings and least upper bounds Theorem 6.41 is a statement about the least upper bound of a special chain, and during its proof some other chains have to be considered. In particular, chains are constructed from others by removing a finite number of elements from the beginning. These operations not only entail a lot of calculations within the respective cpo, but also within the set $\mathbb{N}$ of natural numbers, as many inequations $n_1 \leq n_2$ have to be proved or are part of hypotheses.

Since the method for representing cpo's with rewrite rules described in Sections 8.2 (and 4.2.5) is only applicable if none of the occurring formulas contain nested quantifiers, cpo's are modelled with the help of LP's deduction rule feature. As a result, most calculations concerning lub's are not performed automatically, but have to be started by proper instantiation of the deduction rules (see Appendix C for simple examples for such proofs and a comparison of the two methods to represent cpo's).

A set of rewrite rules that requires a lot of explicit instantiations is the definition of the ordering $\leq$ on $\mathbb{N}$ which is represented by the following specification:

```
declare operator <= : Nat, Nat -> Bool

set name Nat_le_refl
assert
  n <= n
..
set name Nat_le_trans
assert
  (n1 <= n2 & n2 <= n3) => (n1 <= n3)
..
set name Nat_le_antisym
assert
  (n1 <= n2 & n2 <= n1) => (n1 = n2)
..
set name Nat_le_total
assert
  (n1 <= n2) | (n2 <= n1)
..
```

The reason for this need of explicit guidance is again that implications are treated in a different way than rewrite rules. So, if `x <= y & y <= z` is known and transitivity shall be exploited to deduce `x <= z`, the corresponding rule must be properly instantiated. Using it directly as a rewrite rule is only possible if a part of the term to be rewritten matches the whole implication.

Alternatively, these implications could be transformed into deduction rules , e. g.

```
when n1 <= n2, n2 <= n3 yield n1 <= n3   .
```

The disadvantage of such a representation is that this rule would be partially applied whenever a fact `t1 <= t2` for some `Nat` sorted terms `t1` and `t2` is known, producing another deduction rule

```
when t2 <= n3 yield t1 <= n3
```

In the proof of Theorem 6.41, there are many such hypotheses, and hence, as the price for less explicit instantiations, one would have to accept to lot of additional temporary deduction rules that would decrease LP's performance.

Developing the proof The greatest part of the automated proof could be developed very much along the lines of the hand proof. Apart from the complications mentioned in the previous paragraph, the level of detail is quite similar. But two major exceptions have to be mentioned that result from limitations of LP:

The first one concerns the proof for the case of conditional processes. Here the syntactic structure is

$$\begin{array}{rcl} \mathit{sproc} & ::= & \ldots \mid \texttt{IF}\,[gc_1, \ldots, gc_n] \mid \ldots \\ gc & ::= & \mathit{exp} \rightarrow \mathit{sproc} \end{array}$$

For LP, the list construction above has to be made explicit; this leads to the following domain equations (again in **gensig** input syntax, see Appendix A.1):

```
SeqProc  = ... | _IF * GList | ...
GList    = _EMPTY_GL | Expr * SeqProc * GList
```

As a result of this modelling, proofs by induction on the structure of sequential processes regard the case `IF` $[gc_1, \ldots, gc_n]$ as a part of the induction basis, since the processes that are contained in the guarded clauses gc_i occur one level too deep. In order to have the induction hypotheses for these processes as well, the appropriate instance has to be stated explicitly as an additional axiom. Due to the more complicated structure, the clauses for `SeqProc` and for `GList` cannot be merged in the way presented in the expression example (compare Section 10.2.1).

The other case where the proof development requires some kind of trick concerns the case of `WHILE` processes. The hand proof for this case is developed in four different branches, the conditions leading to the cases being rather involved, in particular containing existential quantifiers, e. g.

$$\textit{there exist} \quad n \in \mathbb{N} \quad \textit{and} \quad \sigma_i \in \Sigma,\, in_i \in \mathsf{Input},\, tr_i \in \mathsf{Com}^* \quad \textit{for all} \quad i \le n \quad \textit{such that}$$

$$\forall i \le n : \;\; (\mathcal{C}[\![sproc_1]\!]\, \mathtt{env}_j)^i \;< \sigma, in, tr > = < \sigma_i, in_i, tr_i > \quad \textit{and}$$

$$\forall i < n : \;\; \mathcal{E}[\![exp]\!]\, \mathtt{env}_j\, \sigma_i = \mathsf{tt} \quad \textit{and} \quad \mathcal{E}[\![exp]\!]\, \mathtt{env}_j\, \sigma_n = \mathsf{ff} \quad .$$

Due to the limited complexity of Boolean terms in LP, such formulas cannot be expressed, and so a simple case distinction is not possible. The method used to solve this problem is a "forced case distinction". A new sort of `Cases` is introduced that is generated by four constants:

```
declare sort Cases

declare variable case : Cases

declare operators
  case_a, case_b, case_c, case_d : -> Cases
  case_predicate : Cases -> Bool
..

set name Cases
assert Cases generated by case_a, case_b, case_c, case_d

% "case_predicate" is just a way to introduce a variable of type "Cases"
% into the current subgoal:
assert case_predicate(case)
```

The reason for this construction is the wish to construct four branches in the automated proof. If the original goal is

```
prove p
```

(for some Bool sorted term `p`) the desired branching behaviour can be achieved by considering instead the goal

```
prove
  case_predicate(case) => p
  by induction on case
..
```

This induction results in four cases forming the induction basis, and in each of these cases, the artificial hypothesis `case_predicate(case)` disappears automatically by definition, and the required existentially quantified entities can be introduced by defining appropriate constants. For the case from above, this is done as follows:

```
declare operators
  n' : -> Nat
  states : -> seqState
..
set name theorem6_41CaseHyp
% ... this is the name used by LP for the case hypotheses!
assert
  (n <= n') =>
     ( ( states . n =
       ( (dsem(sprocc, env_j(rhoc, n2)) ^ n) . mk_State(sigma, in, tr) ) )
       & not(is_Inv(states . n)) )

  (s(n) <= n') =>
    ( dsem(ec5, env_j(rhoc, n2), s_1(states . n)) =
        mk_DenVal(mk_Val(true)) )

  dsem(ec5, env_j(rhoc, n2), s_1(states . n')) =
    mk_DenVal(mk_Val(false))
..
```

Of course, the correctness of this proof method relies on a meta-level proof that the case conditions defined by the constants are exclusive and exhaustive.

Statistics The total length of the proof scripts for the proof of Theorem 6.41 is about 5000 lines, again containing mostly comments. The execution time of the proof on a Sun SparcStation 10/40 have been recorded as 33:01.94 minutes, starting from a basic system that took 3:53:49 to be built.

Chapter 11

Conclusion

The aim of this work was to present new techniques for the representation of structured operational and denotational semantics definitions in form of term rewriting systems. These systems were intended to be used within automated proofs about semantics definitions, and the approaches for representation were to be consistent to allow both kinds of semantics to occur in the same proof. In order to become independent of specific proof tools, moreover the formalism to be used was to be simple and general, i. e. it should be based on pure unconditional term rewriting without too many additional features.

In Chapters 7 and 8, representation techniques have been described that fulfil the above requirements. From semantics definitions stated in a fairly general format, simulating term rewriting systems are derived. In both cases the axiomatic basis is the same, consisting of a basic rewriting system $\mathcal{B}$, and so they can be used simultaneously in proofs performed with an automated proof tool.

In order to simulate SOS definitions, one essentially has to deal with two problems. The first one stems from different regulations for the usage of variables in SOS rules and rewrite rules, and the second from the fact that it may be tested whether an SOS rule is applicable to a configuration γ without changing γ or even performing any visible activity. This is a concept that is unknown in unconditional rewriting.

The problem with variable usage is solved by employing the concept of let terms, thus introducing bound variables. These are not directly available in term rewriting, either, but they can be hidden completely with the help of the $\lambda\sigma$-calculus (see Abadi et al. [1]). This calculus is a formulation of λ-calculus that explicitly manages the substitutions resulting from β-reduction (or, here, the evaluation of let terms). Since it only uses simple rewrite rules, no extension of the rewriting formalism is required for its implementation. In order to guarantee correct evaluation of let terms, however, the leftmost-innermost rewriting strategy has to be assumed (corresponding to *call-by-value* reduction).

Testing the applicability of a rewrite rule is also implemented without having to resort to additional concepts (such as conditional rewriting). Here, special properties of SOS systems are exploited that allow to supply the configuration terms included with contexts containing information about the number of SOS rewriting steps that are still allowed, and the SOS rules that may be used for these steps.

The rewrite system $\mathcal{R}$ generated in this way from an SOS system $\mathcal{S}$ simulates it very closely (as proved in Appendix B). Any rewriting sequence of $\mathcal{R}$ corresponds to a transition sequence of $\mathcal{S}$, and any transition sequence of $\mathcal{S}$ is represented by a rewriting sequence.

The usefulness of the rewrite systems derived from SOS definitions is demonstrated in the example proofs, where the representation mechanism is only visible because of the need to add appropriate contexts to configurations. If the basic rewrite system is sufficiently complete to decide all Boolean conditions, then there is never the need to consider details of intermediate terms concerning the $\lambda\sigma$-calculus or contexts.

Finding a rewrite system that models a denotational function definition is much easier than the corresponding problem for SOS systems. Such a definition already consists of equations, and these equations have a natural orientation that allows to view them as rewrite rules. In this case, the real problem is to find suitable representations for the data types that underlie a denotational definition. These are defined by a set of domain equations, for which in general simple sets do not provide solutions. Moreover, recursive functions are used for which the existence of (least) fixed points is required. Because of these reasons, denotational data types usually possess a more complex structure than those needed for SOS definitions. This complexity is reflected by laws about the domains that cannot be expressed without explicit use of quantifiers, in contrast to the SOS case where the assumption of implicit universal quantification over the rules suffices.

In Chapter 4, a method is presented how to employ the $\lambda\sigma$-calculus to model quantifiers as term constructors, provided the formulas expressed in this way stay rather simple. This method can also be applied to the cpo data structures of the example definitions in Chapter 10. Unlike in the SOS case, however, this application of the $\lambda\sigma$-calculus does not remain in the background when it comes to proofs using these rules (to be seen in the example proofs in Appendix C); moreover, it is not very efficient. Therefore, and since the example proof tool (the Larch Prover) provides a simpler alternative to this method (using deduction rules with very restricted use of universal quantifiers), the pure rewrite modelling of cpo's has only been demonstrated in small examples, but not been adopted for larger proofs. In order to be able to work with fixed points, both forms of cpo representation have to rely on some kind of mechanism that allows to control the number of applications of the fixed point expansion rules.

The format of denotational definitions considered in this work is fairly general and is used throughout the literature. For SOS definitions, however, there is a feature that is not allowed by the *ptp/t* format of Def. 5.3 but by other formats, e. g. the GSOS format of Bloom, Istrail, and Meyer [12]. This feature is that of negative antecedents $\gamma \overset{a}{\not\longrightarrow}$ in rules, meaning that from the configuration γ, a transition labelled by a is not possible. This difference reflects a different view of observable properties. With a *ptp/t* system, only transitions are observable; the semantics is described only in terms of what may happen. In contrast, a GSOS system can also observe inability to perform a transition, and hence, the semantics may also be described in terms of what cannot happen.

In another aspect, the *ptp/t* format is more general than the GSOS format and other formats occurring in the literature. The complexity of terms allowed in the premises of rules is not restricted in *ptp/t* rules, in contrast to the other formats. This is motivated again by the different goals to be achieved. The other formats have been devised with the aim to be able to guarantee certain mathematical properties, e. g. that bisimulation is a congruence relation on process terms (see Groote and Vaandrager [60]). The aim of the definition of the *ptp/t* format, however, was to be able to transform a large class of SOS systems into rewrite rules, not regarding the properties of these systems. By the simulation results, the SOS-derived rewrite systems inherit the properties of the SOS systems, and therefore this approach allows to reason about SOS systems even if they fail to possess some otherwise desirable property.

11.1 Practical Experiences with the Simulation

The proofs performed with the Larch Prover and presented in Chapter 10 show that the simulation approach leads to rewrite systems that can be useful for automated verification. These proofs also demonstrate that some automatic support is indispensable for the setting up of the basic rewriting system $\mathcal{B}$.

In the proofs of the simulation theorems (in Appendix B), strong correctness and completeness assumptions about $\mathcal{B}$ have to be made, and so the actual implementation of $\mathcal{B}$ should come as close as possible to the ideal system fulfilling the assumptions. Of course, there must be no compromise when it comes to correctness of $\mathcal{B}$, but with respect to to completeness one is allowed to be more liberal. Here the assumption is that $\mathcal{B}$ can rewrite all simple conditions not depending on the transition relation to either **true** or **false**. But if $\mathcal{B}$ fails to be able to decide a particular condition, then this can only happen because some rule is missing in $\mathcal{B}$, and the only consequence for the current proof is that it gets stuck at some intermediate result. From this result, it is usually not very difficult to see which kind of rule is missing, and so the problem can be resolved.

Nevertheless, each such problem stops the progress of the proof, and therefore $\mathcal{B}$ should be as complete as possible. The tool **gensig** that was developed as part of this work proved to be very useful for this purpose; it is much easier to write down a set of domain equations than the corresponding set of rewrite rules, not because the rules are so complicated, but simply because there are so many of them. Moreover, and perhaps even more important, it is much easier to change the domain equations when one domain turns out be incorrectly or inadequately defined, than to change a whole set of rules.

The algorithm for the transformation of SOS rules from Section 7.2 was implemented as part of this work, too. The resulting tool named **gensos** also was very helpful, since rules in *ptp/t* notation are much easier to understand than the rewrite rules generated from them. In this way, a possible source of errors can be avoided. The use of this tool, however, remains restricted to this particular application, whereas **gensig** can assist in all problems whose solution profits from a structured data representation.

For large applications such as the semantics equivalence proof for $\mathrm{PL}_0^{\mathrm{R}}$ with its about 50 domain equations, one should bear in mind that the system $\mathcal{B}$ generated by **gensig** becomes quite large (easily 1000 rules or more). Moreover, the rules generated by **gensos** are very large compared to usual rewrite rules. Therefore, the approach described here requires a lot of computing power, a lot of storage, and last, but not least, a proof tool that is capable of dealing with large objects. The Larch Prover is an example for a tool satisfying this requirement.

A general observation made during the proofs with LP is that it usually does not make much sense to try and *develop proofs* with a proof tool. Rather, one should have a clear concept beforehand, including at least the proof structure, if not also details, and then try to *check the proof concept* with the help of the tool.

11.2 Future Work

One obvious direction for subsequent work is the application of simulating rewrite systems to other verification problems. First examples might be the "compiling verifications" mentioned in Section 10.1. These have also already been performed by hand [96, 47], and therefore there is again the possibility to compare automated proofs to manual ones and to evaluate their effectiveness. Both these proofs deal with only one style of semantics definition, which makes the simulation aspects simpler than in our examples in Chapter 10.

An interesting modification of the SOS simulation approach might be to simulate a transition relation $\longrightarrow_{\mathcal{S}}$ not on meta-level with a rewrite relation $\longrightarrow_{\mathcal{R}}$ as it is done in this work, but on object level using a representation of the form $\mathit{trans}(\gamma_1, a, \gamma_2)$ to stand for $\gamma_1 \xrightarrow{a}_{\mathcal{S}} \gamma_2$. In this way, additional parameters to the transition relations (such as labels) can be added directly to the representation without having to resort to the trace method of Lemma 5.2. A disadvantage of this modelling is that transitivity of this relation has to be defined by additional rules of the form

$$\mathit{trans}(\gamma_1, a, \gamma_2) \;\wedge\; \mathit{trans}(\gamma_2, b, \gamma_3) \;\Rightarrow\; \mathit{trans}(\gamma_1, ab, \gamma_)$$

If such a rule is to be understood as an unconditional term rewriting rule, then many explicit instantiations are required to apply it inside of proofs. (Examples of such instantiations can be found in Appendix C, where transitivity of $\leq$ on $\mathbb{N}$ is modelled in the same way.)

Useful extensions of the work presented here would be on the one hand the investigation of different concepts of rewriting (in particular order-sorted rewriting), and on the other hand the adaptation of the techniques to other kinds of semantics definitions than those treated so far (in particular axiomatic ones).

11.2.1 Using Order-sorted Rewriting

In Section 3.1, term rewriting was defined based on many-sorted algebra, where no relation exists between the different sorts of a signature. Order-sorted algebra extends this formalism by introducing a partial ordering on the sorts (see e. g. Goguen and Meseguer [56]). In this framework it is possible to define subsorts and supersorts, and many concepts (e. g. partial functions) can be modelled much more elegantly than by adopting many-sorted algebra. Examples for proof systems that support order-sorted logic are OBJ3 [57] and PVS [104].

The price to be paid for the increase in succinctness of specifications is considerably greater complexity in matching, unification and rewriting (cf. Gnaedig, Kirchner and Kirchner [54]).

> Consider for example the following situation. There are two sorts S and T such that S is a subsort of T, and there are the operators
>
> $f : S \to S$
> $a : \to S$
> $b : \to T$
>
> with the rewrite rule $a \longrightarrow b$. This rule is well-typed since any element of (a set modelling) the sort S is also an element of T. But it must not be applied to the term $f(a)$, since

this would produce the ill-typed term $f(b)$. So matching alone does not suffice to decide whether a rule may be applied; here, also more elaborate type checking is needed than in the many-sorted case.

The part of the work that would profit most from using order-sorted rewriting is the generation of the basic rewriting system $\mathcal{B}$. In the many-sorted approach, a large number of operators have to be defined that map sorts representing subdomains into sorts representing superdomains and back (see Section 4.2.1). These rules could be replaced by the much simpler declaration of subsort relations, if order-sorted algebra was used.

11.2.2 Other Styles of Semantics Definitions

Up to now, no axiomatic definitions have been dealt with. There are, however, two special forms of this style that could be represented by rewrite systems in a way that greatly profits from the results of Chapters 7 and 8.

11.2.2.1 Predicate Transformer Semantics

In definitions of this style proposed by Dijkstra [34], the semantics of language constructs maps predicates (that are interpreted as postconditions) to other predicates (that are interpreted as weakest preconditions).[1]

A "semantic version" of such definitions presented in de Bakker [29][2] that is very close to denotational definitions as introduced in Chapter 6.

Consider e. g. a simplified version of the semantics definition for statements in [29]. Among the possible statements, there are assignments and sequences of statements:

$$Stmt \ni S ::= idf := exp \mid S_1 \, ; \, S_2 \mid \ldots$$

A predicate is a mapping from states $\sigma \in \Sigma$ to truth values:

$$\pi \in \Pi = \Sigma \to \mathbb{B}$$

The weakest precondition of statements is defined by a function

$$wp : Stmt \to [\Pi \longrightarrow \Pi]$$

given by equations like

$$wp\,(idf := exp) = \lambda\,\pi\,.\,\lambda\,\sigma\,.\,\pi\,[\,v\,(exp, \sigma)\,/\,idf\,]$$
$$wp\,(S_1\,;\,S_2) = \lambda\,\pi\,.\,wp\,(S_1)\,(wp\,(S_2)\,\pi) \quad .$$

Here $v : Expr, \Sigma \to \mathsf{Val}$ is a valuation function determining the values of expressions. Obviously, this format is very similar to the clausal format for denotational definitions (see Def. 6.12); since the definition of the mathematical domains needed is also very similar, problems and solutions for such definitions should be the same as in Chapter 8 for the denotational case.

[1] The other direction, mapping preconditions to strongest postconditions is also possible, yet less commonly used.

[2] Section 8.3, pp. 308 ff.

Dijkstra [34] and Dijkstra and Scholten [35] suggest a more “syntactic variant” of predicate transformer semantics. The core of the definition is the same, however; the main differences to de Bakker’s version are the use of simpler mathematical domains, the absence of λ-abstractions, and the avoidance of explicit reference to states by special notational conventions. Since this setting is simpler, application of the techniques of Chapter 8 should also be simpler for this variant than for the “semantic version” above.

11.2.2.2 Proof Rules in the Style of Milne

In [91], Milne proposes an axiomatic semantics definition for a variant of the VDM specification language (cf. Jones [76]). For each kind of statement *stmt* proof rules are provided that give information about certain pre- and postconditions $pre\,[\![stmt]\!]$ and $post\,[\![stmt]\!]$. These are related to weakest preconditions by the following equation:

$$\begin{array}{l} wp\,(stmt)\,\pi \iff \\ (\; pre\,[\![stmt]\!] \;\Rightarrow\; (\; \exists v_1' \in type\,[\![stmt]\!], \ldots, v_n' \in type\,[\![stmt]\!] \;) \\ \qquad\qquad\qquad\quad (\; post\,[\![stmt]\!] \;\wedge\; \pi\,[\, v_1', \ldots, v_n' \,/\, v_1, \ldots, v_n \,]\;)\;) \end{array}$$

where $v_1, \ldots, v_n$ are the variables that *stmt* may write to; v_i' refers to the value of v_i after execution of *stmt*.

Examples for rules defining $pre\,[\![stmt]\!]$ and $post\,[\![stmt]\!]$ are the following:

$$pre\,[\![idf := exp]\!] =_{df} pre\,[\![exp]\!]$$

$$post\,[\![idf := exp]\!] =_{df} (\; idf = exp \;)$$

$$\begin{array}{l} pre\,[\![S_1\,;\,S_2]\!] =_{df} \\ \quad pre\,[\![S_1]\!] \;\wedge\; (\; post\,[\![S_1]\!] \;\Rightarrow\; pre\,[\![S_2]\!]\;) \end{array}$$

$$\begin{array}{l} post\,[\![S_1\,;\,S_2]\!] =_{df} \\ \quad (\exists v_1'' \in type\,[\![v_1]\!], \ldots, v_n'' \in type\,[\![v_n]\!]\,) \\ \qquad post\,[\![S_1]\!]\,[v_1'', \ldots, v_n''/v_1', \ldots, v_n'] \;\wedge \\ \qquad post\,[\![S_2]\!]\,[v_1'', \ldots, v_n'', w_1, \ldots, w_m \,/\, v_1, \ldots, v_n, w_1, \ldots, w_m] \end{array}$$

where the v_i are those variables that both S_1 and S_2 may write to, and the w_j are those variables that only S_1 may write to.

In order to generate rewrite systems from such equations, some problems have to be overcome:

- Quantifiers must be dealt with. This could be done in the style of Section 4.2.5.1 provided the formulas become not too complex.
- In order to define the conditions for loops, deduction rules are used. Since these rules are more general than the transition rules of Section 5.2, this seems to go beyond the methods presented here; the Larch Prover, however, supports deduction rules (see Section 9.3 and Appendix C).

- There are many references to intermediate values of variables in the rules. Since these are mostly bound by quantifiers, they could be replaced by de Bruijn indices like in Section 8.2.

The above considerations show that it is reasonable to assume that semantics definitions in the *wp*-calculus and in the style of Milne can also be simulated with the techniques of this work.

Bibliography

[1] MARTIN ABADI, LUCA CARDELLI, PIERRE-LOUIS CURIEN, AND JEAN-JACQUES LÉVY. Explicit substitutions. In *Proceedings of the 17th ACM Symposium on Principles of Programming Languages*, pages 31–46, 1990.

[2] LUCA ACETO, BARD BLOOM, AND FRITS VAANDRAGER. Turning SOS rules into equations. In *Proceedings of the 7th IEEE Symposium on Logic in Computer Science, Santa Cruz, CA*, pages 113–124, 1992. Full version available as CWI Report CS-R 9218, Centrum voor Wiskunde en Informatica, Amsterdam, June 1992.

[3] E. ASTESIANO, A. GIOVINI, F. MAZZANTI, G. REGGIO, AND E. ZUCCA. The Ada challenge for new formal semantic techniques. In PETER. L. J. WALLIS, editor, *Ada: Managing the Transition - Proceedings of the 1986 Ada - Europe International Conference, Cambridge, England*. Cambridge University Press, 1986.

[4] JOS C. M. BAETEN, JAN A. BERGSTRA, AND JAN WILLEM KLOP. Term rewriting systems with priorities. In PIERRE LESCANNE, editor, *Proceedings of the Second International Conference on Rewriting Techniques and Applications, Bordeaux, France*, LNCS 256, pages 83–94. Springer-Verlag, May 1987.

[5] JOS C. M. BAETEN AND C. VERHOEF. A congruence theorem for structured operational semantics with predicates. In EIKE BEST, editor, *Proceedings of CONCUR '93*, LNCS 715, pages 477–492. Springer-Verlag, 1993.

[6] JOS C. M. BAETEN AND W. P. WEIJLAND. *Process Algebra*. Cambridge Tracts in Theoretical Computer Science 18. Cambridge University Press, 1990.

[7] HENDRIK PIETER BARENDREGT. *The Lambda Calculus: Its Syntax and Semantics*. North-Holland, Second edition, 1984.

[8] RUDOLF BERGHAMMER, HERBERT EHLER, AND HANS ZIERER. Towards an algebraic specification of code generation. Technical Report TUM-WF0-06-87-I07-350/1, Institut für Informatik, Technische Universität München, June 1987.

[9] JAN A. BERGSTRA AND JAN WILLEM KLOP. Conditional rewrite rules: Confluence and termination. *Journal of Computer and System Sciences* **32**, 323–362, 1986.

[10] G. BERRY. A hardware implementation of pure Esterel. Rapport de Recherche 06/91, Ecole des Mines, CMA, Sophia-Antipolis, France, 1991.

[11] DINES BJØRNER, C. A. R. HOARE, JONATHAN BOWEN, ET AL.. A ProCoS project description - ESPRIT BRA 3014. *Bulletin of the EATCS* **39**, 60–73, 1989.

[12] BARD BLOOM, SORIN ISTRAIL, AND ALBERT R. MEYER. Bisimulation can't be traced: Preliminary report. In *Proceedings of the 15th ACM Symposium on Principles of Programming Languages (POPL), San Diego, California*, pages 229–239, 1988.

[13] DOEKO BOSSCHER. Term rewriting properties of SOS axiomatisations. In M. HAGIYA AND J. C. MITCHELL, editors, *Proceedings of the International Conference on Theoretical Aspects of Computer Software*, LNCS 789. Springer-Verlag, 1994.

[14] JONATHAN BOWEN ET AL.. A ProCoS II project description: ESPRIT Basic Research project 7071. *Bulletin of the EATCS* **50**, 128–137, 1993.

[15] JONATHAN BOWEN AND PARITOSH PANDYA. Specification of the ProCoS level 0 instruction set. ProCoS Technical Report OU JB2, Oxford University, May 1990.

[16] ROBERT S. BOYER AND J STROTHER MOORE. *A Computational Logic Handbook*. Academic Press, 1988.

[17] MANFRED BROY. Experiences with software specification and verification using LP, the Larch proof assistant. SRC Report 93, Digital Systems Research Center, November 1992.

[18] NICOLAS GOVERT DE BRUIJN. Lambda-calculus notation with nameless dummies. *Indagationes Mathematicae* **34**, 381–392, 1972.

[19] BRUNO BUCHBERGER. History and basic features of the critical-pair/completion procedure. *Journal of Symbolic Computation* **3**(1), 3–38, February 1987.

[20] BETTINA BUTH AND KARL-HEINZ BUTH. Correctness proofs for META IV written code generator specifications using term rewriting. In ROBIN BLOOMFIELD, LYNN MARSHALL, AND ROGER JONES, editors, *VDM '88, VDM – The Way Ahead. Proceedings of the 2nd VDM-Europe Symposium, Dublin, Ireland*, LNCS 328, pages 406–433. Springer-Verlag, September 1988.

[21] BETTINA BUTH AND KARL-HEINZ BUTH. PAMELA - an approach to automatic software verification in industrial applications. In Genser et al. [53], pages 65–70.

[22] BETTINA BUTH AND KARL-HEINZ BUTH. An approach to automatic proof support for code generator verification. In ROBERT GIEGERICH AND SUSAN L. GRAHAM, editors, *Code Generation – Concepts, Tools, Techniques. Proceedings of the International Workshop on Code Generation, Dagstuhl, Germany, 20–24 May 1991*, Workshops in Computing Series, pages 193–209. Springer-Verlag, 1992.

[23] KARL-HEINZ BUTH. Simulation of transition systems with term rewriting systems. Bericht 9212, Institut für Informatik und Praktische Mathematik, Christian-Albrechts-Universität Kiel, 1992.

[24] KARL-HEINZ BUTH. Simulation of SOS definitions with term rewriting systems. In DONALD SANNELLA, editor, *Programming Languages and Systems. Proceedings of the 5th European Symposium on Programming (ESOP '94), Edinburgh, UK*, LNCS 788, pages 150–164. Springer-Verlag, April 1994.

[25] HASKELL B. CURRY AND ROBERT FEYS. *Combinatory Logic, Vol. I*. North-Holland, Second edition, 1968.

[26] BARRY K. DANIELS, editor. *Safety of Computer Control Systems 1990. Proceedings of the IFAC/EWICS/SARS Symposium, Gatwick, UK*. IFAC, Pergamon Press, November 1990.

[27] MAX DAUCHET. Simulation of turing machines by a left-linear rewrite rule. In NACHUM DERSHOWITZ, editor, *Proceedings of the 3rd International Conference on Rewriting Techniques and Applications, Chapel Hill, North Carolina, USA*, LNCS 355, pages 109–120. Springer-Verlag, April 1989.

[28] JOHN DAWES. *The VDM-SL reference guide*. Pitman, 1991.

[29] JACOBUS WILLEM DE BAKKER. *Mathematical Theory of Program Correctness*. Series in Computer Science. Prentice-Hall International, 1980.

[30] ROBERT A. DE MILLO, RICHARD A. LIPTON, AND ALAN J. PERLIS. Social processes and proofs of theorems and programs. *Communications of the ACM* **22**(5), 271–280, May 1979.

[31] ROBERT DE SIMONE. Higher-level synchronising devices in Meije-CCS. *Theoretical Computer Science* **37**, 245–267, 1985.

[32] NACHUM DERSHOWITZ. Termination of rewriting. *Journal of Symbolic Computation* **3**(1), 69–115, February 1987.

[33] NACHUM DERSHOWITZ AND JEAN-PIERRE JOUANNAUD. Rewrite systems. In van Leeuwen [114], chapter 6, pages 243–320.

[34] EDSGER W. DIJKSTRA. *A Discipline of Programming*. Series in Automatic Computation. Prentice-Hall International, 1976.

[35] EDSGER W. DIJKSTRA AND CAREL S. SCHOLTEN. *Predicate Calculus and Program Semantics*. Springer-Verlag, 1990.

[36] DANIEL J. DOUGHERTY. Adding algebraic rewriting to the untyped lambda calculus. In RONALD V. BOOK, editor, *Proceedings of the 4th International Conference on Rewriting Techniques and Applications, Como, Italy*, LNCS 488, pages 37–48. Springer-Verlag, April 1991.

[37] DANIEL J. DOUGHERTY. Some lambda calculi with categorical sums and products. In CLAUDE KIRCHNER, editor, *Proceedings of the 5th International Conference on Rewriting Techniques and Applications, Montreal, Canada*, LNCS 690, pages 137–151. Springer-Verlag, June 1993.

[38] WOLFGANG EHRENBERGER, editor. *Safety of Computer Control Systems 1988. Proceedings of the IFAC Symposium, Fulda, FRG*. IFAC, Pergamon Press, November 1988.

[39] HARTMUT EHRIG AND BERND MAHR. *Fundamentals of Algebraic Specification 1: Equations and Initial Semantics*. Springer-Verlag, 1985.

[40] URBAN ENGBERG, PETER GRØNNING, AND LESLIE LAMPORT. Mechanical verification of concurrent systems with TLA. In Martin and Wing [88], pages 86–97.

[41] MARCIN ENGEL, MARCIN KUBICA, JAN MADEY, DAVID LORGE PARNAS, ANDERS P. RAVN, AND A. JOHN VAN SCHOUWEN. A formal approach to computer systems requirements documentation. In ROBERT L. GROSSMAN, ANIL NERODE, ANDERS P. RAVN, AND HANS RISCHEL, editors, *Hybrid Systems*, LNCS 736, pages 452–474. Springer-Verlag, 1993.

[42] JAMES H. FETZER. Program verification: The very idea. *Communications of the ACM* **31**(9), 1048–1063, September 1988.

[43] ROBERT W. FLOYD. Assigning meanings to programs. In J. T. SCHWARTZ, editor, *Mathematical Aspects of Computer Science*, volume 19 of *Proceedings of Symposia on Applied Mathematics*, pages 19–32. American Mathematical Society, 1967.

[44] RANDY FORGAARD AND JOHN V. GUTTAG. REVE: A term rewriting system generator with failure-resistant Knuth-Bendix. In *Proceedings of an NSF Workshop on the Rewrite Rule Laboratory, Sept. 6–9, 1983*, pages 5–31, Schenectady, NY, April 1984. General Electric Corporate Research and Development Report No. 84GEN008.

[45] MARTIN FRÄNZLE. Compiling specification of ProCoS programming language level 0. ProCoS Technical Report Kiel MF4, Christian-Albrechts-Universität Kiel, April 1990.

[46] MARTIN FRÄNZLE. Compiling specification of ProCoS programming language level 1. ProCoS Technical Report Kiel MF6, Christian-Albrechts-Universität Kiel, April 1990.

[47] MARTIN FRÄNZLE. Operational failure approximation. In DINES BJØRNER, HANS LANGMAACK, AND C. A. R. HOARE, editors, *Provably Correct Systems*, pages 453–484. Danmarks Tekniske Højskole. Final monograph of ESPRIT-BRA "ProCoS", 1992.

[48] ULRICH FRAUS. *Conditional Inductive Theorem Proving*. PhD thesis, Fakultät für Mathematik und Informatik, Universität Passau, 1994. Forthcoming.

[49] ULRICH FRAUS. Inductive theorem proving for algebraic specifications – TIP system user's manual. Technical Report MIP-9401, Fakultät für Mathematik und Informatik, Universität Passau, February 1994.

[50] HEINZ H. FREY, editor. *Safety of Computer Control Systems 1992. Proceedings of the IFAC Symposium, Zürich, Switzerland*. IFAC, Pergamon Press, October 1992.

[51] STEPHEN J. GARLAND AND JOHN V. GUTTAG. An overview of LP, the Larch Prover. In NACHUM DERSHOWITZ, editor, *Proceedings of the Third International Conference on Rewriting Techniques and Applications*, LNCS 355, pages 137–155. Springer-Verlag, 1989.

[52] STEPHEN J. GARLAND AND JOHN V. GUTTAG. *A Guide to LP, The Larch Prover*. Massachussetts Institute of Technology, November 1991. Release 2.2.

[53] ROBERT GENSER, ERWIN SCHOITSCH, AND PETER KOPACEK, editors. *Safety of Computer Control Systems 1989. Proceedings of the IFAC/IFIP Workshop, Vienna, Austria.* IFAC, Pergamon Press, December 1989.

[54] ISABELLE GNAEDIG, CLAUDE KIRCHNER, AND HÉLÈNE KIRCHNER. Equational completion in order-sorted algebras (extended abstract). In MAX DAUCHET AND MAURICE NIVAT, editors, *Proceedings of the 13th Colloquium on Trees in Algebra and Programming (CAAP '88), Nancy, France*, LNCS 299, pages 165–184. Springer-Verlag, March 1988.

[55] KURT GÖDEL. Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme I. *Monatshefte für Mathematik und Physik* **38**, 173–198, 1931.

[56] JOSEPH A. GOGUEN AND JOSÉ MESEGUER. Order-sorted algebra I: Equational deduction for multiple inheritance, overloading, exceptions and partial operations. Technical Monograph PRG-80, Programming Research Group, Oxford University Computing Laboratory, December 1989.

[57] JOSEPH A. GOGUEN AND TIMOTHY WINKLER. Introducing OBJ3. Technical Report SRI-CSL-88-9, SRI International, August 1988.

[58] MICHAEL J. C. GORDON. *The Denotational Description of Programming Languages*. Springer-Verlag, 1979.

[59] MICHAEL J. C. GORDON. HOL: A proof generating system for higher-order logic. In G. BIRTWISTLE AND P.A. SUBRAMANYAM, editors, *VLSI Specification, Verification and Synthesis*, pages 73–128. Kluwer, 1988.

[60] JAN FRISO GROOTE AND FRITS VAANDRAGER. Structured operational semantics and bisimulation as a congruence. *Information and Computation* **100**(2), 202–260, October 1992.

[61] C. A. GUNTER AND D. S. SCOTT. Semantic domains. In van Leeuwen [114], chapter 12, pages 633–674.

[62] JOHN V. GUTTAG AND JAMES J. HORNING. Report on the Larch Shared Language and Larch Shared Language handbook. *Science of Computer Programming* **6**(2), 103–157, March 1986.

[63] JOHN V. GUTTAG AND JAMES J. HORNING. A tutorial on Larch and LCL, a Larch/C interface language. In SØREN PREHN AND HANS TOETENEL, editors, *VDM '91. Formal Software Development Methods. Proceedings of the 4th International Symposium of VDM Europe, Noordwijkerhout, The Netherlands. Volume 2: Tutorials*, LNCS 552, pages 1–78. Springer-Verlag, October 1991.

[64] JOHN V. GUTTAG AND JAMES J. HORNING. *Larch: Languages and Tools for Formal Specification*. Springer-Verlag, 1993.

[65] MARITTA HEISEL, WOLFGANG REIF, AND WERNER STEPHAN. Tactical theorem proving in program verification. In MARK E. STICKEL, editor, *Proceedings of the 10th International Conference on Automated Deduction*, LNCS 449, pages 117–131. Springer-Verlag, 1990.

[66] HANS HERMES. *Aufzählbarkeit, Entscheidbarkeit, Berechenbarkeit*. Heidelberger Taschenbücher 87. Springer-Verlag, Third edition, 1978.

[67] C. A. R. HOARE. An axiomatic basis for computer programming. *Communications of the ACM* **12**(10), 576–583, October 1969.

[68] C. A. R. HOARE. *Communicating Sequential Processes*. Series in Computer Science. Prentice-Hall International, 1985.

[69] C. A. R. Hoare, I. J. Hayes, He Jifeng, C. C. Morgan, A. W. Roscoe, J. W. Sanders, I. H. Sorensen, J. M. Spivey, and B. A. Sufrin. Laws of programming. *Communications of the ACM* **8**(8), 672–686, August 1987. Corrigenda no. 9, p. 770.

[70] Dieter Hofbauer and Ralf-Detlef Kutsche. *Grundlagen des maschinellen Beweisens.* Vieweg, 1989.

[71] Gérard Huet and Dallas S. Lankford. On the uniform halting problem for term rewriting systems. Rapport Laboria 283, INRIA, Le Chesnay, France, 1978.

[72] Gérard Huet and Derek C. Oppen. Equations and rewrite rules: A survey. In R. V. Book, editor, *Formal Languages: Perspectives and Open Problem*, pages 349–405. Academic Press, 1980.

[73] Heinrich Hussmann. Rapid prototyping for algebraic specifications – RAP system user's manual – version 2.0. Technical Report MIP–8504, Fakultät für Mathematik und Informatik, Universität Passau, February 1987. 2nd edition.

[74] Heinrich Hussmann, Alfons Geser, Christian Rank, L. Lavazza, and S. Crespi-Reghizzi. Rapid prototyping with algebraic specification. In Martin Wirsing and Jan A. Bergstra, editors, *Algebraic Methods: Theory, Tools and Applications*, LNCS 394, chapter III, pages 329–444. Springer-Verlag, 1989.

[75] Inmos Ltd. *occam 2 Reference Manual.* Series in Computer Science. Prentice-Hall International, 1988.

[76] Cliff B. Jones. *Systematic Software Development using VDM.* Series in Computer Science. Prentice-Hall International, Second edition, 1990.

[77] Stefan Kahrs. *λ-rewriting.* PhD thesis, Fachbereich Mathematik und Informatik, Universität Bremen, January 1991.

[78] Stéphane Kaplan. Conditional rewrite rules. *Theoretical Computer Science* **33**, 175–193, 1984.

[79] Jan Willem Klop. Combinatory reduction systems. Mathematical Centre Tracts 127, Mathematisch Centrum, Amsterdam, 1980.

[80] Jan Willem Klop. Term rewriting systems. Report CS-R9073, Centrum voor Wiskunde end Informatica, Amsterdam, December 1990.

[81] Donald E. Knuth and Peter B. Bendix. Simple word problems in universal algebras. In J. Leech, editor, *Proceedings of the Conference on Computational Problems in Abstract Algebra, Oxford, 1967*, pages 263–298. Pergamon Press, 1970.

[82] Yassine Lakhneche. Equivalence of denotational and structural operational semantics of PL_0^R. ProCoS Technical Report Kiel YL1, Christian-Albrechts-Universität Kiel, 1991.

[83] Leslie Lamport. The temporal logic of actions. Technical Report 79, Digital Equipment Corporation, Systems Research Center, December 1991.

[84] Pierre Lescanne. REVE: A rewrite rule laboratory. In Jörg H. Siekmann, editor, *Proceedings of the 8th International Conference on Automated Deduction, Oxford, England*, LNCS 230, pages 695–696. Springer-Verlag, July 1986.

[85] Pierre Lescanne. From $\lambda\sigma$ to $\lambda\upsilon$: a journey through calculi of explicit substitutions. In *Proceedings of the 21st ACM Symposium on Principles of Programming Languages (POPL)*, pages 60–69, 1994.

[86] Jacques Loeckx and Kurt Sieber. *The Foundations of Program Verification.* Wiley-Teubner, Second edition, 1987.

[87] Carlos Loria-Saenz and Joachim Steinbach. Termination of combined (rewrite and λ-calculus) systems. In Michaël Rusinowitch and Jean-Luc Rémy, editors, *Proceedings of the 3rd International Workshop on Conditional Term Rewriting Systems, Pont-à-Mousson, France*, LNCS 656, pages 143–147. Springer-Verlag, July 1992.

[88] URSULA H. MARTIN AND JEANNETTE M. WING, editors. *Proceedings of the First International Workshop on Larch, Dedham, MA, 1992*, Workshops in Computing Series. Springer-Verlag, 1993.

[89] JOHN MCCARTHY. Towards a mathematical science of computation. In *Information Processing 1962*, pages 21–28. North-Holland, 1963.

[90] NIELS MELLERGAARD AND JØRGEN STAUNSTRUP. Generating proof obligations for circuits. In Martin and Wing [88], pages 185–200.

[91] ROBERT MILNE. Proof rules for VDM statements. In ROBIN BLOOMFIELD, LYNN MARSHALL, AND ROGER JONES, editors, *VDM '88, VDM – The Way Ahead. Proceedings of the 2nd VDM-Europe Symposium, Dublin, Ireland*, LNCS 328, pages 318–336. Springer-Verlag, September 1988.

[92] ROBERT MILNE AND CHRISTOPHER STRACHEY. *A Theory of Programming Language Semantics*. Chapman and Hall, 1976.

[93] ROBIN MILNER. Operational and algebraic semantics of concurrent processes. In van Leeuwen [114], chapter 19, pages 1201–1242.

[94] FARON MOLLER. The importance of the left merge operator in process algebras. In MICHAEL S. PATERSON, editor, *Proceedings of the 17th International Colloquium on Automata, Languages and Programming, Warwick, England*, LNCS 443, pages 752–764. Springer-Verlag, July 1990.

[95] PETER D. MOSSES. Denotational semantics. In van Leeuwen [114], chapter 11, pages 575–631.

[96] MARKUS MÜLLER-OLM. Correctness proof for SubLisp to PL_0^R translation. ProCoS Technical Report Kiel MMO3, Christian-Albrechts-Universität Kiel, 1990.

[97] PETER NAUR ET AL.. Revised report on the algorithmic language Algol 60. *Computer Journal* **5**, 349–367, 1963.

[98] FRIEDERIKE NICKL. *Algebraic Specification of Semantic Domain Constructions*. PhD thesis, Fakultät für Mathematik und Informatik, Universität Passau, 1988. Available as Technical Report No. MIP-8815, August 1988.

[99] HANNE RIIS NIELSON AND FLEMMING NIELSON. *Semantics with Applications. A Formal Introduction*. Wiley, 1992.

[100] GORDON D. PLOTKIN. A power domain construction. *SIAM Journal on Computing* **5**, 452–487, 1976.

[101] GORDON D. PLOTKIN. An operational semantics for CSP. In DINES BJØRNER, editor, *Formal Description of Programming Concepts - II*, pages 199–225. North-Holland, 1983.

[102] WILLARD VAN ORMAN QUINE. *Word and Object*. MIT Press, 1960.

[103] J. ALAN ROBINSON. A machine-oriented logic based on the resolution principle. *Journal of the ACM* **12**(1), 23–41, January 1965.

[104] JOHN RUSHBY. A tutorial on specification and verification using PVS. In JAMES C. P. WOODCOCK AND PETER GORM LARSEN, editors, *Tutorial Material for FME '93: Industrial-Strength Formal Methods. Proceedings of the First International Symposium of Formal Methods Europe, Odense, Denmark*, pages 357–406, April 1993.

[105] H. SCHMIDT. Zur halbautomatischen Verifikation eines Compilers mit Hilfe des TIP-Systems. Master's thesis, Fakultät für Mathematik und Informatik, Universität Passau, 1990.

[106] UWE SCHMIDT AND HANS-MARTIN HÖRCHER. Programming with VDM domains. In DINES BJØRNER, C. A. R. HOARE, AND HANS LANGMAACK, editors, *VDM '90, VDM and Z – Formal Methods in Software Development. Proceedings of the 3rd VDM-Europe Symposium, Kiel, FRG*, LNCS 428, pages 122–134. Springer-Verlag, April 1990.

[107] DANA S. SCOTT. Data types as lattices. *SIAM Journal on Computing* **5**, 522–587, 1976.

[108] Elizabeth A. Scott and Kathy J. Norrie. Using LP to study the language PL_0^+. In Martin and Wing [88], pages 227–245.

[109] Natarajan Shankar. *Proof Checking Metamathematics*. PhD thesis, University of Texas at Austin, 1986.

[110] M. B. Smyth. Power domains. *Journal of Computer and System Sciences* **16**, 23–36, 1978.

[111] Guy L. Steele Jr. *Common Lisp. The Language*. Digital Press, 1990.

[112] Joachim Steinbach. *Termination of Rewriting - Extensions, Comparison and Automatic Generation of Simplification Orderings*. PhD thesis, Fachbereich Informatik, Universität Kaiserslautern, 1994.

[113] Joseph E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*. MIT Press, 1977.

[114] Jan van Leeuwen, editor. *Handbook of Theoretical Computer Science, Vol. B: Formal Models and Semantics*. Elsevier/MIT Press, 1990.

[115] Martin Wirsing. Algebraic specifications. In van Leeuwen [114], chapter 13, pages 675–788.

[116] William D. Young. A verified code generator for a subset of Gypsy. Technical Report 33, Computational Logic Inc., Austin, TX, 1988.

Appendix A

Automatic Generation of LP Input

As a part of this work, two tools have been implemented to provide support for the automated proofs described in Chapter 10:

(1) a program generating most of the basic system according to the method of Section 4.2 and

(2) a program transforming SOS definitions into rewrite rules, implementing the algorithm of Section 7.2.

Both of these tools are intended as frontends for the Larch Prover, generating detailed complete LP input files from simpler specifications. Since most application areas have special notations for writing concise problem descriptions that are more readable than LP input files, it is quite common to have such frontend tools. Engberg, Grønning and Lamport [40] e. g. use an ML program to transform specifications written in TLA (the *temporal logic of actions*, cf. Lamport [83]) into LP's input language, and Mellergaard and Staunstrup [90] describe a frontend translating synchronous circuit descriptions. In all cases, the motivation is to avoid tedious routine encodings that distract from the real problem to be solved.

Both of the tools mentioned above will be described by means of small examples exhibiting all the features supported.

A.1 Transforming Domain Specifications with gensig

The program gensig generates the basic rewrite system from a domain specification of the type described in Section 4.2. Moreover, it is able to generate special incarnations of the $\lambda\sigma$-calculus (see Section 3.2) and the definitions that are needed to model cpo structures (see Section 8.2).

gensig is written in C with `lex` and `yacc` and has a size of about 6000 lines of code. It has been developed with the help of the VDM ADT domain compiler (cf. Schmidt and Hörcher [106]), a tool whose purpose is very similar to that of gensig. From similar VDM domain specifications it generates C code[1], thus relieving the developer from tedious implementation of data type representations.

Although designed specially to be used in problems of the kind described in this work, gensig

[1]Other languages such as Pascal or Modula-2 are also supported.

can also be useful in other applications of LP. By modelling the relations between domains as defined in domain equations, it introduces a bit of the data modelling power of order-sorted algebra (see p. 138) into the many-sorted world of LP. The disadvantage compared to the true order-sorted approach is the greater complexity of terms that results from the need to write explicitly the injection operators from subsorts into supersorts that are available in implicit form in an order-sorted setting.

A.1.1 An Example Input File

As a small example for gensig input, consider the following file. It is designed to include most of the features supported by the program in as small a file as possible; it does not, however, have any semantic relevance.

```
 1  % ===================================================================
 2  % gensig-ex: signature file as input for gensig

 3  .MAX VAR 4

 4  .CONFIGURATIONS Conf .TERMINAL mk_Conf(s) .CONTEXT Context .RULES 3

 5  .VAR
 6    cf  : Conf,  n   : Nat,    pr  : Prog,
 7    s   : State, val : Value,  x   : Var

 8  .TYPES
 9     .SPECIAL _SKIP
10    Conf  = Prog * State | State
11  ,  Prog  = _SEQU * Prog * Prog | _SKIP
12  ,  State = Var -> Value
13  ,  Value = is not yet defined
14  ,  Var   = is not yet defined

15  .EXPLICIT SUBSTITUTIONS FOR Value WITH 4 INDICES _val

16  .CPO (Value, <=) WITH BOTTOM botv RULETYPE rewrite
```

The file is structured as follows (comments are started by % and extend to the end of the current line):

(1) First, the maximal index for variables is set (line 3). The effect of setting it to 4 is that for each sort S, the following variables are declared: *base*, *base*1, ..., *base*4, where *base* is the name for variables of sort S as introduced in the variable definition section (lines 5–7).

(2) The next, optional section introduces a name for configurations, a term denoting general terminal configurations, a name for contexts and the number of SOS rules (line 4). This extra information is needed to generate data type rules related to SOS definitions. In particular, a general denotation for terminal configurations must be known in order to be able to automatically construct the rules for the *eval* operator (see Section 3.3.1).

(3) After the variable definition section follows the type definition section (lines 6–14). The types of domain equations that are allowed here have already been explained in Section 4.2.

Tokens token are represented in the form _TOKEN, usually written in capital letters only. The product sign $\times$ is written as *. The meaning of "special" tokens (line 9) is explained below.

(4) The next, optional section states which incarnations of the $\lambda\sigma$-calculus are required (line 15). Parameters are the sort of the de Bruijn indices (here `Value`), their number (here 4) and their base name (here `_val`). (Only one index is actually needed; the remaining ones are just used for abbreviation, see Section 3.2.)

(5) The last section (again optional) contains information about the cpo definitions to be generated (line 16). The type of rules used for these definitions can either be `deduction` meaning LP deduction rules are to be used (this is the default), or `rewrite` meaning the rewrite rule modelling of Section 8.2 is to be applied (see Appendix C for a comparison).

A.1.2 Declarations

From the input file `gensig-ex` above, the command

```
gensig gensig-ex
```

produces a set of four different files containing LP input. The output is split in order to enable selective processing. This is useful since not all of the rules that are generated are needed for all applications.

The first of the output files (`gensig-ex.lp`) contains the declarations of sorts, variables, and operators, and it also includes the induction rules that are generated. Besides the sorts that correspond to domain equations, there are some sorts that are always automatically included (`Nat`, `Type`). Others are included automatically only if contexts are needed, i. e. if the second section of the input file is not empty (`Count`, `Flag`, `CCPair`, `Subst`, representing the component types of contexts, configuration-context pairs, and substitutions, respectively).

```
% =====================================================================
% Basic rewriting system for signature file gensig-ex
% Part 1: sorts, operators and variables
% Input file for LP, rel. 2.4, generated by gensig, v. 1.52
% Generated on Sat Feb 12 14:31:13 1994

declare sorts
  Conf, Nat, Prog, State, Value, Var, Count, Flag, CCPair, Context, Type, Subst, Bool
..

declare variables
  cf, cf1, cf2, cf3, cf4 : Conf
  n, n1, n2, n3, n4 : Nat
  pr, pr1, pr2, pr3, pr4 : Prog
  s, s1, s2, s3, s4 : State
  val, val1, val2, val3, val4 : Value
  x, x1, x2, x3, x4 : Var
  ct, ct1, ct2, ct3, ct4 : Count
  s, s1, s2, s3 : Flag
  ccp, ccp1, ccp2, ccp3, ccp4 : CCPair
  K, K1, K2, K3, K4 : Context
  tp, tp1, tp2, tp3, tp4 : Type
  u, u1, u2, u3, u4 : Subst
  b, b1, b2, b3, b4 : Bool
..

declare operators
  type : Conf -> Type
  _Conf : -> Type
  type : Nat -> Type
  _Nat : -> Type
  type : Prog -> Type
  _Prog : -> Type
  type : State -> Type
  _State : -> Type
  type : Value -> Type
  _Value : -> Type
  type : Var -> Type
  _Var : -> Type
  type : Count -> Type
  _Count : -> Type
  type : Flag -> Type
  _Flag : -> Type
  type : CCPair -> Type
  _CCPair : -> Type
  type : Context -> Type
  _Context : -> Type
  type : Subst -> Type
  _Subst : -> Type
  type : Bool -> Type
  _Bool : -> Type
..

declare operators
  @ : Conf, Context -> CCPair
..

declare operator
  eval : CCPair -> Conf
..

declare operators
  mk_Context : Count, Flag, Flag, Flag -> Context
  s_1 : Context -> Count
  s_2 : Context -> Flag
  s_3 : Context -> Flag
  s_4 : Context -> Flag
..

declare operators
  _NULL : -> Count
..

declare operators
  _ONE : -> Count
..

declare operators
  _MANY : -> Count
..

declare operators
  _ON : -> Flag
..

declare operators
  _OFF : -> Flag
..

set name Conf

declare operators
  mk_Conf : Prog, State -> Conf
  s_1 : Conf -> Prog
  s_2 : Conf -> State
..

declare operators
  mk_Conf : State -> Conf
  to_State : Conf -> State
  is_State : Conf -> Bool
..

assert
  Conf generated by
  mk_Conf : Prog, State -> Conf,
  mk_Conf : State -> Conf
..

set name Prog

declare operators
  mk_SEQU_Prog : Prog, Prog -> Prog
  is_SEQU_Prog : Prog -> Bool
  s_1 : Prog -> Prog
  s_2 : Prog -> Prog
..

declare operators
  _SKIP : -> Prog
..

assert
  Prog generated by
  mk_SEQU_Prog : Prog, Prog -> Prog,
  _SKIP : -> Prog
..

declare operators
  .   : State, Var -> Value
..

% ==============================================================
% Application of explicit substitutions as in [ACCL90]

declare operators
  _val1, _val2, _val3, _val4: -> Value
  id : -> Subst                % identity substitution {x_i/x_i}
  sh : -> Subst                % shift substitution {x_(i+1)/x_i}
  +  : Value, Subst -> Subst    % substitution extension (cons)
  *  : Subst, Subst -> Subst    % substitution concatenation
..

% ==============================================================
% Value is an omega cpo:
declare sort seqValue

declare operators
  botv : -> Value
  <= : Value, Value -> Bool
  lub : seqValue -> Value
  chain : seqValue -> Bool
..

% ==============================================================
% Application of explicit substitutions as in [ACCL90]

declare operators
  _n1, _n2, _n3: -> Nat
  id : -> Subst                % identity substitution {x_i/x_i}
  sh : -> Subst                % shift substitution {x_(i+1)/x_i}
  +  : Nat, Subst -> Subst    % substitution extension (cons)
  *  : Subst, Subst -> Subst    % substitution concatenation
..

declare variables
  seqval, seqval1, seqval2, seqval3, seqval4 : seqValue
..
```

```
142  % === Natural numbers:
143  declare sort Nat

144  declare operators
145    forall : Type, Bool -> Bool
146    0 : -> Nat
147    s : Nat -> Nat
148    <= : Nat, Nat -> Bool
149  ..

150  % === Sequences in Value:
151  declare operators
152    .   : seqValue, Nat -> Value
153  ..

154  declare operators
155    type : seqValue -> Type
156    _seqValue : -> Type
157  ..
```

Comments:

(1) As already mentioned above, there are some sorts that are implicitly generated (lines 6–8). The sorts for the components of contexts (`Count` and `Flag`) are added when there is a configuration definition part in the input file. The same holds for the sort of configuration-context pairs (`CCPair`).

The sort of types of terms (`Type`), the sort of $\lambda\sigma$-calculus substitutions (`Subst`) and the sort of Booleans are always included (the latter mostly for sake of completeness, as `Bool` is the only built-in sort of LP).

(2) For each of the sorts, variables are declared according to the `.MAX VAR` definition in the input file (lines 9–23).

(3) The first block of operators contains those that are needed to determine the type of a term, i. e. for each sort S an operator `type` mapping the sort to the type sort, and a constant of sort `Type` representing S (lines 24–49).

(4) The next block contains the declarations of the operators corresponding to the domain equations (lines 50–111). For an explanation of the operators, see Section 4.2. Note the two induction rules in lines 89–93 and 104–108.

In lines 101–103, the effect of declaring `_SKIP` a "special token" can be seen. It results in generating a `Prog` object of the form `_SKIP`; if no such declaration would have been given, the result would have been an object of form `mk_SKIP_Prog` instead.

(5) Lines 112–120 contain the declaration of the $\lambda\sigma$-operators for the sort `Value`, followed by those needed to model that it is a cpo (lines 121–157). Note that due to the request for `RULETYPE rewrite` another incarnation of the $\lambda\sigma$-calculus for the sort `Nat` of natural numbers is also needed (lines 130–138, see Section 8.2).

A.1.3 Basic Rules

The rules generated by `gensig` fall into three parts (and hence are written to three files):

(1) data type rules as explained in Section 4.2, written to `gensig-ex-rules.lp`;

(2) rules modelling distribution of substitution application as it is needed for the implementation of the $\lambda\sigma$-calculus, written to `gensig-ex-subst.lp`;

(3) type rules relating sort and their corresponding types (objects of sort `type`) and stating that all types are distinct, written to `gensig-ex-type.lp`.

Only the rules of the first group will be reproduced here as the other rules are generated in a very obvious way, however resulting in rather lengthy files.

```
% =====================================================================
% Basic rewriting system for signature file gensig-ex
% Part 2: rules; requires gensig-ex.lp
% Input file for LP, rel. 2.4, generated by gensig, v. 1.52
% Generated on Sat Feb 12 14:31:13 1994

set activity off

set name CCPair
assert
  eval(cf @ mk_Context(_NULL, s, s1, s2)):Conf -> cf
  eval(mk_Conf(s) @ K):Conf -> mk_Conf(s)
..

make immune CCPair

set name Context
assert
  s_1(mk_Context(ct, s, s1, s2)):Count -> ct
  s_2(mk_Context(ct, s, s1, s2)):Flag -> s
  s_3(mk_Context(ct, s, s1, s2)):Flag -> s1
  s_4(mk_Context(ct, s, s1, s2)):Flag -> s2
..

make immune Context

set name Count
assert
  _NULL = _ONE -> false
  _NULL = _MANY -> false
  _ONE = _MANY -> false
..

make immune Count

set name Flag
assert
  _ON = _OFF -> false
..

make immune Flag

set name Conf
assert
  s_1(mk_Conf(pr, s)):Prog -> pr
  s_2(mk_Conf(pr, s)):State -> s
  is_State(mk_Conf(s))
  to_State(mk_Conf(s)) -> s
  mk_Conf(pr, s) = mk_Conf(s) -> false
  not(is_State(mk_Conf(pr, s)))
..

make immune Conf

set name Prog
assert
  s_1(mk_SEQU_Prog(pr, pr1)):Prog -> pr
  s_2(mk_SEQU_Prog(pr, pr1)):Prog -> pr1
  is_SEQU_Prog(mk_SEQU_Prog(pr, pr1))
  mk_SEQU_Prog(pr, pr1) = _SKIP -> false
  not(is_SEQU_Prog(_SKIP))
..

make immune Prog

set name State
assert
..

make immune State

% === Rules for cpo Value:
set immunity on

set name bottom_Value
assert
  botv <= val
..

set name ord_Value
assert
  % <= is an ordering relation on Value:
  val <= val
  (val1 <= val2 & val2 <= val3) => (val1 <= val3)
  (val1 <= val2 & val2 <= val1) => (val1 = val2)
..

set name chain_Value
assert
  chain(seqval) -> forall(_Nat, (seqval . _n1) <= (seqval . (s(_n1))))
..
% === Rules for natural numbers:
% <= is a total ordering relation on Nat:
set name Nat_le_refl
assert
  n <= n
..
set name Nat_le_trans
assert
  (n1 <= n2 & n2 <= n3) => (n1 <= n3)
..
set name Nat_le_antisym
assert
  (n1 <= n2 & n2 <= n1) => (n1 = n2)
..
set name Nat_le_total
assert
  (n1 <= n2) | (n2 <= n1)
..
set name Nat_le_succ
assert
  (s(n1) <= s(n2)) -> (n1 <= n2)
..

% === More rules for cpo Value:
set name lub_Value
assert
  % lub is upper bound:
  chain(seqval) => ((seqval . n) <= lub(seqval))
  % lub is smaller than all upper bounds:
  (chain(seqval) & forall(_Nat, (seqval . _n1) <= val)) => (lub(seqval) <= val)
  % "constant tail" rule for lub:
  (chain(seqval) & forall(_Nat, (n <= _n1) => ((seqval . n) = (seqval . _n1))))
    => (lub(seqval) = (seqval . n))
..
```

```
106  set immunity off

107  set ordering left-to-right
108  order

109  set activity on
110  make active *
```

Some of the terms in these rules that include overloaded operators have to be "qualified" (i. e. annotated with their result sort) in order to help LP's type checker to understand the input (e. g. in lines 7–18).

Before processing, activity is turned off (line 6), and all rules are declared to be immune (lines 12, 20 etc.). The reason for these settings is that otherwise, LP would start to internormalize all rules, that is, normalize each rule with all the other rules. For sets of rules as large as those set up here, this is rather time consuming, even though it has little effect as most rules are already in normal form. Therefore internormalization is prevented, and terms are only normalized while proving conjectures, but not within the rules themselves. Of course, the rules have to be re-activated after all of them have been processed (line 110).

Lines 56–106 contain the rules for the cpo `Value`. Note that rules about the total ordering `<=` on `Nat` are needed (lines 73–94) in order to be able to deal with sequences of elements of `Value`.

A.2 Translating SOS Rules with gensos

The program gensos implements the translation algorithm from Section 7.2. It has been developed in the same way as gensig and has a size of about 3500 lines of code. Its output relies on a basic rewriting system as produced by gensig.

A.2.1 An Example SOS Input File

As in Section A.1.1, input for and output of gensos will be explained with the help of a small input file not having any semantic significance. Since the task of gensos is to translate a set of SOS rules into rewrite rules and the structure of the SOS rules does not change, it suffices to present just the first SOS rule. (Assume, however, that there are a total of four rules to consider.)

In the abstract form of Def. 5.3, this rule looks as follows:

$$\frac{\vdash \textit{is-special}\,(pr_1) \;\wedge\; pr_1 \longrightarrow pr_3 \;\wedge\; pr_2 \longrightarrow pr_4 \;\wedge\; \textit{is-not-special}\,(pr_4)}{\vdash \langle\, pr_1, pr_2 \,\rangle \longrightarrow pr_3}$$

where configurations are either elements $pr \in \mathit{Prog}$ or pairs thereof. In this rule, $\textit{is-special}\,(pr_1)$ forms the precondition part (pr_1 is not an extra variable), $pr_1 \longrightarrow pr_3 \;\wedge\; pr_2 \longrightarrow pr_4$ forms the transition part, and $\textit{is-not-special}\,(pr_4)$ forms the postcondition part (pr_4 is an extra variable).

The representation of this rule for gensos assumes the domain equation (in gensig input syntax)

```
Config = Prog | Prog * Prog
```

The rule is represented as follows:

```
 1  % ================================================
 2  % gensos-ex: small input file for gensos

 3  .SYSTEMS Config .IN CONTEXT Context .WITH 4 .RULES AND INDICES x

 4  .VAR
 5    cf : Config,
 6    n  : Nat,
 7    b  : Bool,
 8    ct : Context,
 9    pr : Prog

10  .NAME R1

11  INFER mk_Config(mk_Prog(pr1, pr2)) -> mk_Config(pr3)
12  FROM  is_special(pr1),
13        mk_Config(pr1) -> mk_Config(pr3) &
14        mk_Config(pr2) -> mk_Config(pr3,pr4),
15        is_not_special(pr4)
```

The structure of this file is rather simple:

(1) It starts with an introduction of the name of the sort of configurations together with the name of the sort of contexts, the total number of SOS rules and the base name of the de Bruijn indices used (line 3).

(2) The next section contains the variable declarations (lines 4–9); this is just the same as in the input for gensig.

(3) The body of the input file contains a set of groups of rules (in this case only one group with just one rule in it). Each group is preceded by a name declaration (line 10) that is directly translated into a `set name` command for LP. Thus, larger SOS definitions can be structured by introducing different names and single rules can be picked out more easily. In the example from Chapter 10, each rule has been given the name of the corresponding rule in [82].

(4) The format of the rules is that of Def. 5.3. Precondition part `is_special(pr1)`, transition part (`mk_Config(pr1) ->` ...), and postcondition part `is_not_special(pr4)` are separated by commas (lines 11–15).

A.2.2 Generated SOS Derived Rules

From the input file above, the command

```
gensos gensos-ex
```

produces the following output in the file `gensos-ex.lp`:

```
% ================================================================
% Term rewriting system simulating SOS system in file gensos-ex
% Input file for LP, rel. 2.4, generated by gensos, v. 1.33
% Generated on Tue Mar  1 21:50:11 1994

set immunity on

% ===================================================
% Application of explicit substitutions as in [ACCL90]
```

```
declare operators
  x1, x2, x3 : -> Config
  id : -> Subst                % identity substitution {x_i/x_i}
  sh : -> Subst                % shift substitution {x_(i+1)/x_i}
  +  : Config, Subst -> Subst    % substitution extension (cons)
  *  : Subst, Subst -> Subst    % substitution concatenation
..

set name sigma_Config
assert
  % rules for eliminating bound variables
  x1 # id -> x1                           % VarId
  x1 # (cf + u) -> cf                      % VarCons
  % rules for evaluating substitutions
  id * u -> u                              % IdL
  sh * id -> sh                            % ShiftId
  sh * (cf + u) -> u                       % ShiftCons
  (cf + u) * u1 -> (cf # u1) + (u * u1)    % Map
  (u1 * u2) * u3 -> u1 * (u2 * u3)         % Ass
  (cf # u1) # u2 -> cf # (u1 * u2)         % Clos
  % extra rules
  u * id -> u                              % IdR
  x1 + sh -> id                           % VarShift
  (x1 # u) + (sh * u) -> u                % SCons
  x2 -> x1 # sh
  x3 -> x1 # (sh * sh)
..

make passive sigma_Config

% =================================================
% Rewrite rules generated from the transition system

set name R1
assert
mk_Config(mk_Prog(pr1, pr2)) @ mk_Context(_ONE, _ON, s2, s3, s4) ->
   if(is_special(pr1) & is_Prog(x1) & (type(x2) = I(_Config))
      & (comps(x2) = I(s(s(0)))) & (type(s_1(x2)) = I(_Prog))
      & (type(s_2(x2)) = I(_Prog)) & is_not_special(s_2(x2)) & (s_1(x2) = to_Prog(x1)),
   % then
   mk_Config(to_Prog(x1)) @ mk_Context(_NULL, _ON, s2, s3, s4),
   % else
   I(mk_Config(mk_Prog(pr1, pr2))) @ mk_Context(_ONE, _OFF, s2, s3, s4))
   #(eval(mk_Config(pr1) @ mk_Context(_ONE, _ON, _ON, _ON, _ON)):Config
     + (eval(mk_Config(pr2) @ mk_Context(_ONE, _ON, _ON, _ON, _ON)):Config + id))

mk_Config(mk_Prog(pr1, pr2)) @ mk_Context(_MANY, _ON, s2, s3, s4) ->
   if(is_special(pr1) & is_Prog(x1) & (type(x2) = I(_Config))
      & (comps(x2) = I(s(s(0)))) & (type(s_1(x2)) = I(_Prog))
      & (type(s_2(x2)) = I(_Prog)) & is_not_special(s_2(x2)) & (s_1(x2) = to_Prog(x1)),
   % then
   mk_Config(to_Prog(x1)) @ mk_Context(_MANY, _ON, _ON, _ON, _ON),
   % else
   I(mk_Config(mk_Prog(pr1, pr2))) @ mk_Context(_MANY, _OFF, s2, s3, s4))
   #(eval(mk_Config(pr1) @ mk_Context(_ONE, _ON, _ON, _ON, _ON)):Config
     + (eval(mk_Config(pr2) @ mk_Context(_ONE, _ON, _ON, _ON, _ON)):Config + id))
..

make active sigma_Config

order

set immunity off
```

Comments:

(1) As in the rule systems generated by gensig, all rules are made immune to prevent internor-

malization (line 5).

(2) The first block of output contains the required incarnation of the $\lambda\sigma$-calculus (lines 6–34); cf. Section 3.2.

(3) As already mentioned, the rule groups begin with a `set name` command, followed by the two rules generated for each of the SOS rules in the group (one-step and many-step case). Note how terms that are known not to include extra variables (resp. de Bruijn indices) are protected from substitution by surrounding them with the `I` operator (see Section 3.2.3.2). Again, some of the terms in the rules have to be qualified with their result sort in order to pass LP's type checker.

Appendix B

Complete Proofs for the Simulation Results of Chapter 7

B.1 Preliminaries

In this appendix, the results of Section 7.2 will be proved. Besides the abbreviations of Chapter 7, the following notation is used:

Notation B.1

Let $n \in \mathbb{N}, \gamma, \gamma' \in \Gamma$, and K, K_1 be contexts. Then $\boldsymbol{\gamma @ K \xrightarrow{[n]}_{\mathcal{R}} \gamma' @ K_1}$ will be written if $\gamma @ K \xrightarrow{*}_{\mathcal{R}} \gamma' @ K_1$ with exactly n outermost applications of SOS-derived rules. If $n = 1$ and the rule that is applied outermost is the $m-$th for some $m \in [N]$, it will also be written $\boldsymbol{\gamma @ K \xrightarrow{[1(m)]}_{\mathcal{R}} \gamma' @ K_1}$.

The 1-version and the $*$-version of an SOS-derived rule differ only in the first components of the contexts on their left-hand sides and in the bodies of the if expressions on their right-hand sides. Therefore, a rewriting sequence

$$\gamma @ K_1^{(k)} \xrightarrow{[1(k)]}_{\mathcal{R}} \gamma' @ K_{0f}$$

containing configurations in 1-contexts can be transformed into a sequence

$$\gamma @ K^{(k)} \xrightarrow{[1(k)]}_{\mathcal{R}} \gamma' @ K_f$$

containing the same configurations in $*$-contexts and vice versa. The substitutions needed for rewrite rule applications are identical in both cases; only the counter components of the contexts have to be changed and the other form of rule k has to be applied. So the following two lemmas hold:

Lemma B.2

Each rewriting in a 1-context can also be performed in a $*$-context: Let $\gamma, \gamma' \in \Gamma, k \in [N], r_1, \ldots, r_N \in \{\mathsf{on}, \mathsf{off}\}$ such that $\gamma @ K_1^{(k)} \xrightarrow{[1(k)]}_{\mathcal{R}} \gamma' @ K_{0f}$. Then it also holds that $\gamma @ K^{(k)} \xrightarrow{[1(k)]}_{\mathcal{R}} \gamma' @ K_f$.

Lemma B.3

Each one-step SOS rewriting in an $*$-context can also be performed in a 1-context: Let $\gamma, \gamma' \in \Gamma, k \in [N], r_1, \ldots, r_N \in \{\mathsf{on}, \mathsf{off}\}$ and $\gamma \mathbin{@} K^{(k)} \xrightarrow{[1(k)]}_{\mathcal{R}} \gamma' \mathbin{@} K_f$. Then it also holds that $\gamma \mathbin{@} K_1^{(k)} \xrightarrow{[1(k)]}_{\mathcal{R}} \gamma' \mathbin{@} K_{0f}$.

B.2 Completeness

It suffices to prove the following theorem:

Theorem B.4 (completeness)

$$\forall\, M \in \mathbb{N}_0\, \forall\, \gamma, \gamma' \in \Gamma : \gamma \xrightarrow{M}_{\overline{\overline{\mathcal{S}}}} \gamma' \;\Rightarrow\; \gamma \mathbin{@} K_f \xrightarrow{[M]}_{\mathcal{R}} \gamma' \mathbin{@} K_f$$

Proof

Let $M \in \mathbb{N}$ and $\gamma, \gamma' \in \Gamma$ with $\gamma \xrightarrow{M}_{\overline{\overline{\mathcal{S}}}} \gamma'$. The proof proceeds by induction on the total number M' of transitions needed to establish this transition sequence. (Note that $M' \geq M$ because there may be transitions that are needed to prove premises of SOS rules.)

Induction basis: $M' = 0$.

In this case, also $M = 0$, hence $\gamma = \gamma'$, and thus obviously $\gamma \mathbin{@} K_f \xrightarrow{[0]}_{\mathcal{R}} \gamma' \mathbin{@} K_f$.

Induction step: $M' > 0$.

Under this condition, also $M > 0$ because SOS transitions can only occur if there is at least one top-level transition. Hence there is a $\gamma" \in \Gamma$ with $\gamma \longrightarrow_{\overline{\overline{\mathcal{S}}}} \gamma" \xrightarrow{M-1}_{\overline{\overline{\mathcal{S}}}} \gamma'$. The induction hypothesis yields $\gamma" \mathbin{@} K_f \xrightarrow{[M-1]}_{\mathcal{R}} \gamma' \mathbin{@} K_f$. Hence it remains to be proved that $\gamma \mathbin{@} K_f \xrightarrow{[1]}_{\mathcal{R}} \gamma" \mathbin{@} K_f$.

There is a $k \in [N]$ such that $\gamma \longrightarrow_{\overline{\overline{\mathcal{S}}}} \gamma"$ is possible due to rule k. This rule is of the form presented in Def. 5.3:

$$\frac{\vdash \bigwedge_{i=1}^{p} b_i \;\wedge\; \bigwedge_{j=1}^{n} \gamma_j \xrightarrow{L_j} \gamma_j' \;\wedge\; \bigwedge_{k=1}^{q} B_k}{\vdash \overline{\gamma} \longrightarrow \overline{\gamma}\,'}$$

Case 1: $n = 0$ (it follows that $q = 0$).

In this case, the rule (7.12) is generated:

$$\overline{\gamma} \mathbin{@} Kv^{(k)} \longrightarrow \mathsf{if} \;\bigwedge_{i=1}^{p} b_i\; \mathsf{then}\; \overline{\gamma}' \mathbin{@} K_f \;\mathsf{else}\; \overline{\gamma} \mathbin{@} \overline{Kv}^{(k)}$$

where $Kv^{(k)} =_{df} \ulcorner\langle *, s_1, \ldots, s_{k-1}, \mathsf{off}, s_{k+1}, \ldots, s_N \rangle\urcorner$. Since rule k has been chosen, there exists a substitution σ that fulfils the conditions (1) to (3) from Def. 5.8. In particular:

$$\gamma = \overline{\gamma}\sigma \tag{1}$$
$$\forall i \in [p] : b_i\sigma =_E \mathsf{true} \tag{2}$$
$$\gamma" = \overline{\gamma}\sigma \tag{3}$$

By general assumption, $\mathcal{B} \subseteq \mathcal{R}$ is complete for conditions; since true is a normal form, it follows that

$$\forall i \in [p] : b_i\sigma \xrightarrow{*}_{\mathcal{R}} \mathsf{true}$$

Moreover for $\sigma_1 =_{df} \{\mathsf{on}/s_i \mid i \in [N], i \neq k\}$:

$$Kv^{(k)}\sigma_1 = \langle *, \mathsf{on}, \ldots, \mathsf{on} \rangle = K_f.$$

The s_i only occur in contexts but not in configurations. Therefore it holds for $\sigma_2 =_{df} \sigma \uplus \sigma_1$ that $\overline{\gamma}\sigma_2 = \gamma$ and $Kv^{(k)}\sigma_2 = K_f$, hence $\gamma @ K_f = (\overline{\gamma} @ Kv^{(k)})\sigma_2$, and also for all $i \in [p]$ that $b_i\sigma_2 = b_i\sigma \xrightarrow{*}_{\mathcal{R}} \mathsf{true}$.

Therefore the derived rule is applicable for $\gamma @ K_f$, and the resulting term is

$$\begin{aligned}
\ulcorner(\overline{\gamma}' @ K_f)\urcorner\sigma_2 &= \ulcorner \llcorner\overline{\gamma}'\sigma_2\lrcorner @ \llcorner K_f\sigma_2\lrcorner \urcorner && [\text{substitution is homomorphic}]\\
&= \ulcorner \llcorner\overline{\gamma}'\sigma\lrcorner @ \llcorner K_f\sigma_1\lrcorner \urcorner && [\sigma \text{ and } \sigma_1 \text{ are disjoint}]\\
&= \ulcorner \gamma" @ \llcorner K_f\sigma_1\lrcorner \urcorner && [\text{condition (3)}]\\
&= \ulcorner \gamma" @ K_f \urcorner && [K_f \text{ is constant}]
\end{aligned}$$

Hence, it has been proved $\gamma @ K_f \xrightarrow{[1(k)]}_{\mathcal{R}} \gamma" @ K_f$, and therefore $\gamma @ K_f \xrightarrow{[1]}_{\mathcal{R}} \gamma" @ K_f$.

Case 2: $n > 0$.

As in Case 1, there are substitutions σ, σ_1 and σ_2 with $\sigma_2 = \sigma \uplus \sigma_1, \overline{\gamma}\sigma = \gamma$ and $Kv^{(k)}\sigma_1 = K_f$ such that

$$\forall i \in [p] : b_i\sigma_2 \xrightarrow{*}_{\mathcal{R}} \mathsf{true} \quad \text{and} \quad \gamma @ K_f = (\overline{\gamma} @ Kv^{(k)})\sigma_2$$

Therefore, the derived rule (7.14) can be applied, the outermost if term on the right-hand side can be reduced to its **then** case, and one obtains:

$$\begin{array}{rl}
\ulcorner \gamma \mathbin{@} K_f \urcorner & = \ulcorner (\overline{\gamma} \mathbin{@} Kv^{(k)}) \urcorner \sigma_2 \\
\xrightarrow{*}_{\mathcal{R}} & \lceil (\textsf{let} \;\; x_1 = \mathit{eval}\,(\, \gamma_1 \mathbin{@} \langle\, L_1, \textsf{on}, \ldots, \textsf{on} \,\rangle\,), \\
& \qquad\quad \ldots, \\
& \qquad\quad x_n = \mathit{eval}\,(\, \gamma_n \mathbin{@} \langle\, L_n, \textsf{on}, \ldots, \textsf{on} \,\rangle\,) \\
& \quad \textsf{in if} \;\; \bigwedge_{i=1}^{n} \llcorner \mathit{type\text{-}ok}\,(x_i, \gamma'_i, \mathit{var}\,(\overline{\gamma})) \lrcorner \wedge \bigwedge_{i=1}^{q} \overline{B_i} \wedge \llcorner \mathit{EV\text{-}match} \lrcorner \\
& \qquad\quad \textsf{then} \;\; \tilde{\gamma}\,' \mathbin{@} K_f \quad \textsf{else} \quad \overline{\gamma} \mathbin{@} \overline{Kv}^{(k)} \;\;) \rceil \, \sigma_2 \\
= & \lceil \textsf{let} \;\; x_1 = \mathit{eval}\,(\, \llcorner \gamma_1 \sigma \lrcorner \mathbin{@} \langle\, L_1, \textsf{on}, \ldots, \textsf{on} \,\rangle\,) \\
& \qquad \ldots, \\
& \qquad x_n = \mathit{eval}\,(\, \llcorner \gamma_n \sigma \lrcorner \mathbin{@} \langle\, L_n, \textsf{on}, \ldots, \textsf{on} \,\rangle\,) \\
& \quad \textsf{in if} \;\; \bigwedge_{i=1}^{n} \llcorner (\mathit{type\text{-}ok}\,(x_i, \gamma'_i, \mathit{var}\,(\overline{\gamma})))\sigma \lrcorner \wedge \bigwedge_{i=1}^{q} \llcorner \overline{B_i}\sigma \lrcorner \wedge \llcorner \mathit{EV\text{-}match} \lrcorner \\
& \qquad \textsf{then} \;\; \llcorner \tilde{\gamma}\,'\sigma \lrcorner \mathbin{@} K_f \quad \textsf{else} \quad \llcorner \overline{\gamma}\sigma \lrcorner \mathbin{@} \llcorner \overline{Kv}^{(k)} \sigma_1 \lrcorner \rceil \\
& \qquad\qquad \left[\begin{array}{l} \text{substitution is homomorphic, definition of } \sigma_2, \\ \mathit{EV\text{-}match} \text{ is free of variables} \end{array} \right]
\end{array}$$

Case 2.1: The SOS rule does not contain extra variables.

It follows that $q = 0$ and $\mathit{EV\text{-}match} = \textsf{true}$. Furthermore by Def. 7.2:

$$(\mathit{type\text{-}ok}\,(x_i, \gamma'_i, \mathit{var}\,(\overline{\gamma})))\sigma \equiv \ulcorner x_i = \gamma'_i\sigma \urcorner \qquad \text{for all } i \in [n]$$

and by definition of $\tilde{\gamma}\,'$ ((4) in Def. 7.5)

$$\tilde{\gamma}\,'\sigma \equiv \overline{\gamma}\,'\sigma \quad .$$

Hence the resulting term for the development above is:

$$\begin{array}{rl}
X \equiv_{df} & \lceil \textsf{let} \;\; x_1 = \mathit{eval}\,(\, \llcorner \gamma_1 \sigma \lrcorner \mathbin{@} \langle\, L_1, \textsf{on}, \ldots, \textsf{on} \,\rangle\,) \\
& \qquad \ldots, \\
& \qquad x_n = \mathit{eval}\,(\, \llcorner \gamma_n \sigma \lrcorner \mathbin{@} \langle\, L_n, \textsf{on}, \ldots, \textsf{on} \,\rangle\,) \\
& \quad \textsf{in if} \;\; \bigwedge_{i=1}^{n} x_i = \llcorner \gamma'_i \sigma \lrcorner \\
& \qquad \textsf{then} \;\; \llcorner \overline{\gamma}\,'\sigma \lrcorner \mathbin{@} K_f \quad \textsf{else} \quad \llcorner \overline{\gamma}\sigma \lrcorner \mathbin{@} \llcorner \overline{Kv}^{(k)} \sigma_1 \lrcorner \rceil
\end{array}$$

By condition (3.1) from Def. 5.8, for all $i \in [n]$ holds that $\gamma_i\sigma \xrightarrow{L_i} \gamma'_i\sigma$, and consequently $\ulcorner \mathit{eval}\,(\, \llcorner \gamma_i \sigma \lrcorner \mathbin{@} \langle\, L_i, \textsf{on}, \ldots, \textsf{on} \,\rangle\,) \urcorner \xrightarrow{*}_{\mathcal{R}} \gamma'_i\sigma$.

[Proof: Let $i \in [n]$.

Case $L_i = 1$:
The transitions in $\gamma_i\sigma \xrightarrow{L_i} \gamma'_i\sigma$ are a proper subset of those counted in M'. Therefore it is true by induction that $\ulcorner \llcorner \gamma_i \sigma \lrcorner \mathbin{@} K_f \urcorner \xrightarrow{[1]}_{\mathcal{R}} \ulcorner \llcorner \gamma'_i \sigma \lrcorner \mathbin{@} K_f \urcorner$. From Lemma B.3, it follows $\ulcorner \llcorner \gamma_i \sigma \lrcorner \mathbin{@} K_{1f} \urcorner \xrightarrow{*}_{\mathcal{R}} \ulcorner \llcorner \gamma'_i \sigma \lrcorner \mathbin{@} K_{0f} \urcorner$, and hence $\ulcorner \mathit{eval}\,(\, \llcorner \gamma_i \sigma \lrcorner \mathbin{@} K_{1f}\,) \urcorner \xrightarrow{*}_{\mathcal{R}} \gamma'_i\sigma$, since $\gamma'_i\sigma$ is completely evaluated in the context K_{0f} (cf. Section 3.3.1) .

Case $L_i = *$:

As in the other case, it follows by induction that $\ulcorner \llcorner \gamma_i\sigma \lrcorner @ K_f \urcorner \xrightarrow{*}_{\mathcal{R}} \ulcorner \llcorner \gamma'_i\sigma \lrcorner @ K_f \urcorner$, hence also $\ulcorner eval(\llcorner \gamma_i\sigma \lrcorner @ \langle L_i, \mathsf{on}, \ldots, \mathsf{on} \rangle) \urcorner \xrightarrow{*}_{\mathcal{R}} \gamma'_i\sigma$ since $\gamma'_i\sigma \in T$ (see Def. 5.3, condition (9)), and terminal configurations are completely evaluated (cf. Section 3.3.1).

□]

Therefore, one obtains

$$\begin{aligned}
X \xrightarrow{*}_{\mathcal{R}} & \lceil \mathsf{let}\ \ x_1 = \llcorner \gamma'_1\sigma \lrcorner,\ \ldots,\ x_n = \llcorner \gamma'_n\sigma \lrcorner \\
& \quad \mathsf{in\ if}\ \ \bigwedge_{i=1}^{n}\ x_i = \llcorner \gamma'_i\sigma \lrcorner \\
& \qquad \mathsf{then}\ \ \llcorner \overline{\gamma}'\sigma \lrcorner @ K_f \\
& \qquad \mathsf{else}\ \ \llcorner \overline{\gamma}\sigma \lrcorner @ \llcorner \overline{Kv}^{(k)}\sigma_1 \lrcorner \rceil \\
\xrightarrow{+}_{\mathcal{R}} & \ulcorner \llcorner \overline{\gamma}'\sigma \lrcorner @ K_f \urcorner && [\lambda\sigma\text{-calculus, } \mathcal{B} \text{ is complete for conditions}] \\
= \ & \ulcorner \gamma\text{''} @ K_f \urcorner && [\overline{\gamma}'\sigma = \gamma\text{''} \text{ (condition (3.3) from Def. 5.8)}]
\end{aligned}$$

and thus $\ulcorner \gamma @ K_f \urcorner \xrightarrow{[1]}_{\mathcal{R}} \ulcorner \gamma\text{''} @ K_f \urcorner$.

<u>Case 2.2</u>: There are extra variables in the SOS rule.

In this case

$$\begin{aligned}
& \forall i \in [n] : (type\text{-}ok(x_i, \gamma'_i, var(\overline{\gamma})))\sigma \ \equiv \\
& \qquad \lceil type(x_i) = \mathsf{\_T}_{\gamma'_i} \ \wedge\ \#\mathsf{comp}(x_i) = \llcorner \#comp(\gamma'_i) \lrcorner \ \wedge \\
& \qquad\quad \bigwedge_{k=1}^{\llcorner \#comp(\gamma'_i) \lrcorner} \llcorner type\text{-}ok(\ulcorner x_i \downarrow k \urcorner, \gamma'_i/k, var(\overline{\gamma}))\sigma \lrcorner \rceil \\
& \tilde{\gamma}' = \overline{\gamma}'\sigma_{EV} \\
& \overline{B_i} = B_i\sigma_{EV}
\end{aligned}$$

where

$$\sigma_{EV} =_{df} [\, x_{j_v} w_v / v \mid v \text{ is an extra variable} \,]$$

and $\mathsf{\_T}_{\gamma'_i}$ is a type constant representing the type $T_{\gamma'_i}$. Therefore the resulting term for the development above is:

$$\begin{aligned}
X \equiv_{df} & \lceil \mathsf{let}\ \ x_1 = eval(\llcorner \gamma_1\sigma \lrcorner @ \langle L_1, \mathsf{on}, \ldots, \mathsf{on} \rangle), \\
& \qquad \ldots, \\
& \qquad x_n = eval(\llcorner \gamma_n\sigma \lrcorner @ \langle L_n, \mathsf{on}, \ldots, \mathsf{on} \rangle) \\
& \quad \mathsf{in\ if}\ \ \bigwedge_{i=1}^{n} [\, type(x_i) = \mathsf{\_T}_{\gamma'_i} \ \wedge\ \#\mathsf{comp}(x_i) = \llcorner \#comp(\gamma'_i) \lrcorner \ \wedge \\
& \qquad\qquad \bigwedge_{k=1}^{\llcorner \#comp(\gamma'_i) \lrcorner} \llcorner type\text{-}ok(\ulcorner x_i \downarrow k \urcorner, \gamma'_i/k, var(\overline{\gamma})) \lrcorner \,]\, \sigma \ \wedge \\
& \qquad\quad \bigwedge_{i=1}^{q} \llcorner (B_i\sigma_{EV})\sigma \lrcorner \ \wedge\ \llcorner EV\text{-}match \lrcorner \\
& \qquad \mathsf{then}\ \ \llcorner (\overline{\gamma}'\sigma_{EV})\sigma \lrcorner @ K_f \quad \mathsf{else} \quad \llcorner \overline{\gamma}\sigma \lrcorner @ \llcorner \overline{Kv}^{(k)}\sigma_1 \lrcorner \rceil
\end{aligned}$$

By condition (3) from Def. 5.8, there exists an extension σ' of σ onto $var(R_k)$ with $\gamma_j\sigma' = \gamma_j\sigma$

for all $j \in [n]$ (because there are no extra variables in γ_i by condition (6) in Def. 5.3) and

$$\forall j \in [n] : \gamma_j \sigma \xrightarrow{L_j}_j \gamma'_j \sigma' \tag{4}$$

$$\forall j \in [q] : B_j \sigma' =_E \mathsf{true} \tag{5}$$

$$\gamma" = \overline{\gamma}\,'\sigma' \tag{6}$$

As in Case 2.1, one obtains for $j \in [n]$

$$\ulcorner \mathit{eval}\,(\,\llcorner \gamma_j \sigma \lrcorner\,@\,\langle\, L_j, \mathsf{on}, \ldots, \mathsf{on} \,\rangle\,)^{\urcorner} \xrightarrow{*}_{\mathcal{R}} \gamma'_j \sigma' \quad .$$

Hence:

$$\begin{aligned}
X \;\xrightarrow{*}_{\mathcal{R}}\; & \lceil \mathsf{let}\;\; x_1 = \llcorner \gamma'_1 \sigma' \lrcorner,\; \ldots,\; x_n = \llcorner \gamma'_n \sigma' \lrcorner \\
& \quad \mathsf{in\ if} \;\; \bigwedge_{i=1}^{n} \,[\; \mathit{type}\,(x_i) = \_\mathsf{T}_{\gamma'_i} \;\wedge\; \#\mathsf{comp}(x_i) = \llcorner \#\mathit{comp}\,(\gamma'_i) \lrcorner \;\wedge \\
& \qquad\qquad \bigwedge_{k=1}^{\llcorner \#\mathit{comp}\,(\gamma'_i) \lrcorner} \;\llcorner \mathit{type\text{-}ok}\,(\ulcorner x_i \downarrow k^{\urcorner}, \gamma'_i/k, \mathit{var}\,(\overline{\gamma})) \lrcorner \,]\,\sigma \;\wedge \\
& \qquad\quad \bigwedge_{i=1}^{q} \llcorner (B_i \sigma_{EV}) \sigma \lrcorner \;\wedge\; \llcorner \mathit{EV\text{-}match} \lrcorner \\
& \qquad \mathsf{then} \;\; \llcorner (\overline{\gamma}\,'\sigma_{EV}) \sigma \lrcorner\,@\,K_f \quad \mathsf{else} \quad \llcorner \overline{\gamma} \sigma \lrcorner\,@\,\llcorner \overline{Kv}^{\,(k)} \sigma_1 \lrcorner \rceil \\
\xrightarrow{+}_{\mathcal{R}}\; & \lceil \mathsf{if} \;\; \bigwedge_{i=1}^{n} \,[\; \mathit{type}\,(\gamma'_i \sigma') = \_\mathsf{T}_{\gamma'_i} \;\wedge\; \#\mathsf{comp}(\gamma'_i \sigma') = \llcorner \#\mathit{comp}\,(\gamma'_i) \lrcorner \;\wedge \\
& \qquad\qquad \bigwedge_{k=1}^{\llcorner \#\mathit{comp}\,(\gamma'_i) \lrcorner} \;\llcorner \mathit{type\text{-}ok}\,(\ulcorner \gamma'_i \sigma' \downarrow k^{\urcorner}, \gamma'_i/k, \mathit{var}\,(\overline{\gamma})) \lrcorner \,]\,\sigma \;\wedge \\
& \qquad \bigwedge_{i=1}^{q} \llcorner (B_i \sigma_{EV}) \sigma \lrcorner \;\wedge\; \llcorner \mathit{EV\text{-}match}[\,\gamma'_i \sigma' / x_i \mid i \in [n]\,] \lrcorner \\
& \quad \mathsf{then} \;\; \llcorner (\overline{\gamma}\,'\sigma_{EV}) \sigma \lrcorner\,@\,K_f \quad \mathsf{else} \quad \llcorner \overline{\gamma} \sigma \lrcorner\,@\,\llcorner \overline{Kv}^{\,(k)} \sigma_1 \lrcorner \rceil \\
& \hspace{20em} [\,\lambda\sigma\text{-calculus}\,]
\end{aligned}$$

where $\sigma'_{EV} =_{df} [\,(\gamma'_{j_v} \sigma') w_v / v \mid v \text{ is an extra variable}\,]$.

Since all components of the structure of γ'_i also occur in $\gamma'_i \sigma'$, it follows easily from the definition of *type-ok* that the *type-ok* part of the **if** condition equals true, and therefore can be rewritten by $\mathcal{B}$ (and thus also by $\mathcal{R}$) to true since $\mathcal{B}$ is complete for such conditions. Furthermore one can deduce that $(B_i \sigma'_{EV})\sigma \xrightarrow{*}_{\mathcal{R}} \mathsf{true}$ for $i \in [q]$.

[Proof: Let v be an extra variable. By definition holds $\gamma'_{j_v} w_v = v$, and the occurrence of v in γ'_{j_v} is the same as that of $v\sigma'$ in $\gamma'_{j_v}\sigma'$. Thus $(\gamma'_{j_v}\sigma')w_v = v\sigma'$, and by completeness of $\mathcal{B}$ follows $(\gamma'_{j_v}\sigma')w_v \downarrow_{\mathcal{B}} v\sigma'$. Consequently for $i \in [q]$:

$$\begin{aligned}
(B_i \sigma'_{EV})\,\sigma \;&=\; (B_i[\,(\gamma'_{j_v}\sigma')w_v/v \mid v \text{ is an extra variable}\,])\,\sigma \\
&\downarrow_{\mathcal{B}}\; (B_i[\,v\sigma'/v \mid v \text{ is an extra variables}\,])\,\sigma \\
&=\; B_i \sigma' \qquad\qquad\qquad\qquad [\,\sigma' \text{ is an extension of } \sigma\,]
\end{aligned}$$

By condition (5), $B_i\sigma' =_E \mathsf{true}$, thus $B_i\sigma' \xrightarrow{*}_{\mathcal{B}} \mathsf{true}$ by completeness of $\mathcal{B}$. So we have $(B_i\sigma'_{EV})\sigma \downarrow_{\mathcal{B}} B_i\sigma' \xrightarrow{*}_{\mathcal{B}} \mathsf{true}$, and by correctness of $\mathcal{B}$ follows $(B_i\sigma'_{EV})\sigma =_E \mathsf{true}$. Again by completeness of $\mathcal{B}$, one can deduce $(B_i\sigma'_{EV})\sigma \xrightarrow{*}_{\mathcal{B}} \mathsf{true}$, and hence $(B_i\sigma'_{EV})\sigma \xrightarrow{*}_{\mathcal{R}} \mathsf{true}$. $\Box$]

EV-match $[\,\gamma'_i\sigma'/x_i \mid i \in [n]\,]$ also rewrites to true.

[Proof: By definition of *Ev-match*, it holds that

$$\begin{aligned}&EV\text{-}match\,[\,\gamma'_i\sigma'/x_i \mid i \in [n]\,] \equiv\\ &\quad \bigwedge \{\ulcorner (\llcorner \gamma'_k\sigma' \lrcorner) w'_v = (\llcorner \gamma'_{j_v}\sigma' \lrcorner) w_v \urcorner \mid v \text{ is an extra variable},\\ &\qquad \gamma'_k w'_v \text{ occurrence of } v \text{ different from } \gamma'_{j_v} w_v\,\}.\end{aligned}$$

Let v be an extra variable, $v = \gamma'_j w'_v$, and $(j, w'_v) \neq (j_v, w_v)$. Then one must prove $\ulcorner (\llcorner \gamma'_j\sigma' \lrcorner) w'_v = (\llcorner \gamma'_{j_v}\sigma' \lrcorner) w_v \urcorner \xrightarrow{*}_{\mathcal{R}} \mathsf{true}$. Since substitution application commutes with projection to subterms, it holds that

$$(\gamma'_j\sigma')w'_v =_E (\gamma'_j w'_v)\sigma' =_E v\sigma',$$

$$(\gamma'_{j_v}\sigma')w_v =_E (\gamma'_{j_v} w_v)\sigma' =_E v\sigma',$$

hence $(\gamma'_j\sigma')w'_v = (\gamma'_{j_v}\sigma')w_v$, and by completeness of $\mathcal{B}$ follows the desired result. $\Box$]

Therefore the whole if condition rewrites to true, and thus

$$\begin{aligned} X &\xrightarrow{+}_{\mathcal{R}} \ulcorner \llcorner (\overline{\gamma}'\sigma'_{EV})\,\sigma \lrcorner \,@\, K_f \urcorner \\ &\xrightarrow{*}_{\mathcal{R}} \ulcorner \llcorner \overline{\gamma}'\sigma' \lrcorner \,@\, K_f \urcorner && [\text{as in the proof of } (B_i\sigma'_{EV})\sigma \xrightarrow{*}_{\mathcal{R}} \mathsf{true} \text{ above}]\\ &= \ulcorner \gamma\text{''} \,@\, K_f \urcorner && [\text{by } (6)]\end{aligned}$$

and so $\gamma \,@\, K_f \xrightarrow{[1]}_{\mathcal{R}} \gamma\text{''} \,@\, K_f$.

$\Box$

From the result proved, one immediately obtains normal-form completeness, and with Lemma B.3 also one-step completeness.

B.3 Correctness

Similar to the case of completeness, it suffices to prove

Theorem B.5 (correctness)

$$\forall M \in \mathbb{N}\, \forall \gamma, \gamma' \in \Gamma : \gamma \,@\, K_f \xrightarrow{[M]}_{\mathcal{R}} \gamma' \,@\, K_f \;\Rightarrow\; \gamma \xrightarrow{M}_{\mathcal{S}} \gamma'$$

Proof

Let $M \in \mathbb{N}$ and $\gamma, \gamma' \in \Gamma$ such that $\gamma \,@\, K_f \xrightarrow{[M]}_{\mathcal{R}} \gamma' \,@\, K_f$. The proof proceeds by induction on the total number M' of applications of SOS-derived rules in this sequence (again, observe that $M' \geq M$).

Induction basis: $M' = 0$.

Since γ and K_f do not contain subterms that are configuration/context pairs, in this case $\gamma \,@\, K_f \xrightarrow{*}_{\mathcal{B}} \gamma' \,@\, K_f$. Since $\mathcal{B}$ is not concerned with configuration/context pairs, this means $\gamma \xrightarrow{*}_{\mathcal{B}} \gamma'$, and by correctness of $\mathcal{B}$ follows $\gamma =_E \gamma'$, and thus $\gamma \xrightarrow{0}_{\mathcal{S}} \gamma'$ by Def. 5.7.

Induction step: $M' > 0$.

In this case, also $M > 0$ and there is some $\gamma" \in \Gamma$ with $\gamma \,@\, K_f \xrightarrow{[1]}_{\mathcal{R}} \gamma" \,@\, K_f \xrightarrow{[M-1]}_{\mathcal{R}} \gamma' \,@\, K_f$. By induction holds $\gamma" \xrightarrow{M-1}_{\mathcal{S}} \gamma'$ and it remains to prove $\gamma \xrightarrow{1}_{\mathcal{S}} \gamma"$.

Let $k \in [N]$ such that $\gamma \,@\, K_f \xrightarrow{[1(k)]}_{\mathcal{R}} \gamma" \,@\, K_f$ uses only SOS-derived rule k outermost. Let D be the number of applications of SOS-derived rules in this sequence that occur on inner levels.

Case 1: $D = 0$.

In this situation, rule k must have the form (7.12):

$$\overline{\gamma} \,@\, Kv^{(k)} \rightarrow \textsf{if} \bigwedge_{i=1}^{p} b_i \textsf{ then } \overline{\gamma}' \,@\, K_f \textsf{ else } \overline{\gamma} \,@\, \overline{Kv}^{(k)}$$

Contexts can only be modified in SOS-derived rules. Therefore, there are $\gamma_M, \gamma'_M \in \Gamma$ such that[1]

$$\gamma \,@\, K_f \xrightarrow{*}_{\mathcal{B}} \gamma_M \,@\, K_f \rightarrow_{\mathcal{R}'} \gamma'_M \,@\, K_f \xrightarrow{*}_{\mathcal{B}} \gamma" \,@\, K_f$$

(remember $\mathcal{R} = \mathcal{B} \uplus R'$) and by Def. 3.18 and the definition of the rules for the $\textsf{if}$ operator (see Section 4.2.5) there is a substitution σ with

$$\overline{\gamma}\sigma = \gamma_M, \overline{\gamma}'\sigma = \gamma'_M, Kv^{(k)}\sigma = K_f$$
$$\forall i \in [p] : b_i\sigma \xrightarrow{*}_{\mathcal{B}} \textsf{true}$$

Since $\mathcal{B}$ is not concerned with contexts, $\gamma \xrightarrow{*}_{\mathcal{B}} \gamma_M$ and $\gamma'_M \xrightarrow{*}_{\mathcal{B}} \gamma"$, and by correctness of $\mathcal{B}$ follows $\gamma =_E \gamma_M, \gamma'_M =_E \gamma"$. Similarly it holds for $i \in [p]$ that $b_i\sigma =_E \textsf{true}$. Summarizing, there exists a substitution σ for the variables in $\overline{\gamma}$ with

(1) $\gamma =_E \gamma_M = \overline{\gamma}\sigma$, hence $\gamma =_E \overline{\gamma}\sigma$

(2) $\forall i \in [p] : b_i\sigma =_E \textsf{true}$

(3.3) $\gamma" =_E \gamma'_M = \overline{\gamma}'\sigma$, hence $\gamma" =_E \overline{\gamma}'\sigma$

i. e. $\gamma \rightarrow_{\mathcal{S}} \gamma"$ using rule k. (Conditions (3.1) and (3.2) from Def. 5.5 hold vacuously since there are no extra variables in rule k.)

[1]Without loss of generality we may assume that there are no detours that result from failing type checks in SOS-derived rules. Such detours would only lengthen the sequence without changing the situation.

Case 2: $D > 0$.

In this case, the SOS-derived rule has the form (7.14):

```
$\overline{\gamma} @ Kv^{(k)} \longrightarrow$
    if $\bigwedge_{i=1}^{p} b_i$
      then let $x_1 = eval(\gamma_1 @ \langle L_1, \mathsf{on}, \ldots, \mathsf{on} \rangle)$,
               $\ldots$,
               $x_n = eval(\gamma_n @ \langle L_n, \mathsf{on}, \ldots, \mathsf{on} \rangle)$
           in if $\bigwedge_{i=1}^{n} \llcorner type\text{-}ok(x_i, \gamma_i', var(\overline{\gamma}))\lrcorner \wedge \bigwedge_{i=1}^{q} \overline{B_i} \wedge \llcorner EV\text{-}match \lrcorner$
                then $\tilde{\gamma}' @ K_f$
                else $\overline{\gamma} @ \overline{Kv}^{(k)}$
      else $\overline{\gamma} @ \overline{Kv}^{(k)}$
```

As in case 1, there are γ_M, γ_{M_i} $(i \in [n]), \tilde{\gamma}'_M \in \Gamma$, and a substitution σ such that

```
$\ulcorner \gamma @ K_f \urcorner \xrightarrow{*}_{\mathcal{B}} \ulcorner \gamma_M @ K_f \urcorner$
    $\longrightarrow_{\mathcal{R}'} X \equiv_{df} \lceil$ let $x_1 = eval(\gamma_{M_1} @ \langle L_1, \mathsf{on}, \ldots, \mathsf{on} \rangle)$,
                                $\ldots$,
                                $x_n = eval(\gamma_{M_n} @ \langle L_n, \mathsf{on}, \ldots, \mathsf{on} \rangle)$
                            in if $\bigwedge_{i=1}^{n} \llcorner type\text{-}ok(x_i, \gamma_i', var(\overline{\gamma}))\,\sigma \lrcorner \wedge$
                                  $\bigwedge_{i=1}^{q} \llcorner \overline{B_i}\,\sigma \lrcorner \wedge \llcorner EV\text{-}match \lrcorner$
                               then $\tilde{\gamma}'_M @ K_f$
                               else $\gamma_M @ \overline{K}^{(k)} \rceil$
    $\xrightarrow{*}_{\mathcal{R}} \ulcorner \gamma" @ K_f \urcorner$
```

and

$$\gamma_M = \overline{\gamma}\sigma \tag{1}$$

$$K_f = Kv^{(k)}\sigma \tag{2}$$

$$\gamma_{M_i} = \gamma_i\sigma \quad (i \in [n]) \tag{3}$$

$$\tilde{\gamma}'_M = \tilde{\gamma}'\sigma \tag{4}$$

$$\overline{K}^{(k)} = \overline{Kv}^{(k)}\sigma \tag{5}$$

$$b_i\sigma \xrightarrow{*}_{\mathcal{B}} \mathsf{true} \ (i \in [p]) \tag{6}$$

Note that by (6), the outermost **if** of rule (7.14) could be removed using the rules about this operator. In the last step of the derivation above, $\xrightarrow{*}_{\mathcal{R}}$ is used (unlike in Case 1, where $\xrightarrow{*}_{\mathcal{B}}$ was used), because the **let** clauses have been evaluated.

Considering the context K_f of the final term, one conclude that the **then** part of T has been chosen in this rewriting sequence. Therefore the condition must rewrite to **true** (after evaluation of the **let** clauses), and since $\mathcal{B}$ is correct, this means that the condition equals **true** in E.

Case 2.1: The corresponding SOS rule does not contain extra variables.

Then it must be the case that $\tilde{\gamma}\,' = \overline{\gamma}\,', q = 0, \mathit{Ev\text{-}match} = \textbf{true}$, and $\mathit{type\text{-}ok}\,(x_i, \gamma_i', \mathit{var}\,(\overline{\gamma}))\,\sigma \;\equiv\; \ulcorner x_i = \llcorner \gamma_i'\sigma \lrcorner \urcorner$ for $i \in [n]$. Furthermore, none of the x_i occurs in the **then** or **else** case of the rule. Consequently,

$$X \xrightarrow{*}_{\mathcal{R}} \tilde{\gamma}\,'_M \;@\; K_f \xrightarrow{*}_{\mathcal{B}} \gamma\text{''} \;@\; K_f \quad , \tag{7}$$

the latter because there is only one outermost application of SOS-derived rules in the sequence $\gamma \;@\; K_f \xrightarrow{[1]}_{\mathcal{R}} \gamma\text{''} \;@\; K_f$. Since the condition rewrites to **true**, for each $i \in [n]$ there must be a term T_i such that

$$\ulcorner \mathit{eval}\,(\,\llcorner \gamma_i \sigma \lrcorner \;@\; \langle\, L_i, \mathsf{on}, \ldots, \mathsf{on} \,\rangle\,) \urcorner \xrightarrow{*}_{\mathcal{R}} \; T_i \;{}_{\mathcal{R}}\!\xleftarrow{*} \gamma_i'\sigma \tag{8}$$

$\gamma_i'\sigma$ is a configuration without a context; hence $\gamma_i'\sigma \xrightarrow{*}_{\mathcal{B}} T_i$ must hold, and by correctness of $\mathcal{B}$ follows $\gamma_i'\sigma =_E T_i$.

Now one can prove: $\forall i \in [n] : \gamma_i \sigma \xrightarrow{L_i}_{\mathcal{S}} \gamma_i'\sigma$.

[Proof: Let $i \in [n]$.
Case $L_i = 1$: Then

$$\ulcorner \mathit{eval}\,(\,\llcorner \gamma_i \sigma \lrcorner \;@\; \langle\, 1, \mathsf{on}, \ldots, \mathsf{on} \,\rangle\,) \urcorner \xrightarrow{*}_{\mathcal{R}} T_i.$$

Since *eval* terms only occur in SOS-derived rewrite rules, but not in the SOS rules themselves, γ_i' is not an *eval* term, and neither is T_i because the substitution σ also does not involve *eval* terms. So (8) must contain the successful application of an SOS-derived rule, because the *eval* operator can be removed only then (see Section 3.3.1). So there is a configuration $\tilde{\gamma}_i \in \Gamma$ with

$$\ulcorner \mathit{eval}\,(\,\llcorner \gamma_i \sigma \lrcorner \;@\; \langle\, 1, \mathsf{on}, \ldots, \mathsf{on} \,\rangle\,) \urcorner \;\xrightarrow{*}_{\mathcal{R}}\; \ulcorner \mathit{eval}\,(\,\tilde{\gamma}_i \;@\; K_{0f}\,) \urcorner \xrightarrow{*}_{\mathcal{R}} T_i \tag{9}$$

and also $\gamma_i \sigma \;@\; \langle\, 1, \mathsf{on}, \ldots, \mathsf{on} \,\rangle \;\xrightarrow{*}_{\mathcal{R}}\; \tilde{\gamma}_i \;@\; K_{0f}$. By Lemma B.2, it also holds that $\gamma_i \sigma \;@\; K_f \xrightarrow{[1]}_{\mathcal{R}} \tilde{\gamma}_i \;@\; K_f$. Since (9) is a proper subsequence of the initial sequence containing at least one SOS application less, it follows by induction that $\gamma_i \sigma \rightarrow_{\mathcal{S}} \tilde{\gamma}_i$. $\tilde{\gamma}_i$ cannot contain configuration/context pairs (otherwise the type check in R would have failed), hence $\tilde{\gamma}_i \xrightarrow{*}_{\mathcal{B}} T_i$ and by correctness of $\mathcal{B}$ also $\tilde{\gamma}_i =_E T_i$. So $\tilde{\gamma}_i =_E \gamma_i'\sigma$ and thus $\gamma_i \sigma \rightarrow_{\mathcal{S}} \gamma_i'\sigma$ by Lemma 5.6.

Case $L_i = *$: In this case

$$\ulcorner \mathit{eval}\,(\,\ulcorner \gamma_i \sigma \urcorner \;@\; \langle\, *, \mathsf{on}, \ldots, \mathsf{on} \,\rangle\,) \urcorner \xrightarrow{*}_{\mathcal{R}} T_i.$$

As in the other case, there must be a point in this sequence where the *eval* operator is removed, i. e. some $P \in \mathbb{N}$ and $\tilde{\gamma}_i \in T'$ (see Section 3.3.1) such that

$$\begin{aligned} \ulcorner eval\,(\,\llcorner \gamma_i\sigma \lrcorner\,@\,\langle\,*, \mathsf{on}, \ldots, \mathsf{on}\,\rangle\,)\urcorner &\xrightarrow{[P]}_{\mathcal{R}} \ulcorner eval\,(\,\tilde{\gamma}_i\,@\,\langle\,*, \mathsf{on}, \ldots, \mathsf{on}\,\rangle\,)\urcorner \\ &\longrightarrow_{\mathcal{R}} \tilde{\gamma}_i \\ &\xrightarrow{*}_{\mathcal{R}} T_i \end{aligned}$$

(assuming without loss of generality that no detours occur in this sequence). The induction hypothesis yields $\gamma_i\sigma \xrightarrow{P}_{\mathcal{S}} \tilde{\gamma}_i$, and as in the other case $\tilde{\gamma}_i \xrightarrow{*}_{\mathcal{B}} T_i$ holds, hence $\tilde{\gamma}_i =_E T_i$, and thus $\gamma_i\sigma \xrightarrow{*}_{\mathcal{S}} \gamma_i'\sigma$.

□]

Since $\gamma\,@\,K_f \xrightarrow{*}_{\mathcal{B}} \gamma_M\,@\,K_f$, it follows as above $\gamma \xrightarrow{*}_{\mathcal{B}} \gamma_M$ and hence $\gamma =_E \gamma_M$. Similarly one can derive $\tilde{\gamma}\,'_M =_E \gamma$", and on the whole this yields

(1) $\gamma =_E \gamma_M = \overline{\gamma}\sigma$, hence $\gamma =_E \overline{\gamma}\sigma$

(2) $\forall i \in [p] : b_i\sigma = \mathsf{true}$

(3.1) $\forall i \in [n] : \gamma_i\sigma \xrightarrow{L_i}_{\mathcal{S}} \gamma_i'\sigma$

(3.3) $\gamma" =_E \tilde{\gamma}\,'_M = \tilde{\gamma}\,'\sigma = \overline{\gamma}\,'\sigma$, hence $\gamma" =_E \overline{\gamma}\,'\sigma$

i. e. $\gamma \longrightarrow_{\mathcal{S}} \gamma"$ using rule k (condition (3.2) holds vacuously since $q = 0$).

Case 2.2: The corresponding SOS rule contains extra variables.

Somewhere in the sequence $X \xrightarrow{*}_{\mathcal{R}} \gamma"\,@\,K_f$ β-reduction (substitution application) takes place (because the reduction strategy is *call-by-value*, only after the let clauses have been completely evaluated). The resulting term has the form

$$T_1 \equiv (\,\mathsf{if}\ \ldots\ \mathsf{then}\ \ldots\ \mathsf{else}\ \ldots)\,\sigma_x$$

where for $i \in [n]$ there are $M_i \in \mathbb{N}_0$ and $\tilde{\gamma}_i \in \Gamma$ such that

$$\sigma_x =_{df} [\,\tilde{\gamma}_i/x_i \mid i \in [n]\,]$$

and

$$\gamma_i\sigma\,@\,K_{1f} \xrightarrow{[M_i]}_{\mathcal{R}} \tilde{\gamma}_i\,@\,K_{0f}\ \text{, if } L_i = 1 \tag{10}$$

$$\gamma_i\sigma\,@\,K_f \xrightarrow{[M_i]}_{\mathcal{R}} \tilde{\gamma}_i\,@\,K_f\ \text{, if } L_i = * \tag{11}$$

From the construction of the SOS-derived rules and the rules for removing *eval*, it follows that $M_i = 1$ if $L_i = 1$.

According to Def. 5.5, the goal is to find an extension σ' of σ onto $var\,(R_k)$ with

(3.1) $\forall i \in [n] : \gamma_i\sigma \xrightarrow{L_j}_{\mathcal{S}} \gamma_i'\sigma'$

(3.2) $\forall i \in [q] : B_i\sigma' =_E \mathsf{true}$

(3.3) $\gamma" =_E \overline{\gamma}\,'\sigma'$

Define $\sigma' =_{df} \sigma_x \circ (\,\sigma \cup \sigma_{EV}\,)$, where $\sigma_{EV} =_{df} [\,x_{j_v}w_v/v \mid v \in EV_k\,]$. Then it can be proved that σ' fulfils (3.1) – (3.3).

Proof for (3.1):

It suffices to prove

$$\forall i \in [n] : \tilde{\gamma}_i =_E \gamma_i' \sigma'$$

because this implies (3.1) by (10) resp. (11), Lemma 5.6, and the induction hypothesis.

So let $i \in [n]$. Since the condition of T_1 is rewritten to true with rules of $\mathcal{B}$, it is, by correctness of $\mathcal{B}$, also E-equal to true. Hence it may be assumed that

$$\mathsf{true} =_E \textit{type-ok}\,(x_i, \gamma_i', \textit{var}\,(\overline{\gamma}))\,\sigma\,)\,\sigma_x = \textit{type-ok}\,(\tilde{\gamma}_i, \gamma_i', \textit{var}\,(\overline{\gamma}))\,\sigma$$

which is by definition of *type-ok* (cf. Def. 7.2) the term

$$\ulcorner \tilde{\gamma}_i = \llcorner \gamma_i' \sigma \lrcorner \urcorner \tag{12}$$

if there are no extra variables in γ_i', or

$$\ulcorner \mathsf{type}(\tilde{\gamma}_i) = \llcorner \textit{type}\,(\tilde{\gamma}\,'i) \lrcorner \urcorner \tag{13}$$

otherwise, if γ_i' is not a tuple constructor term, or

$$\begin{aligned} &\ulcorner \mathsf{type}(\tilde{\gamma}_i) = \_\mathsf{T}_{\gamma_i'} \ \wedge\ \#\mathsf{comp}(\tilde{\gamma}\,'i) = \llcorner \#\textit{comp}\,(\gamma_i'\sigma) \lrcorner \wedge \\ &\bigwedge_{i=1}^{\llcorner \#\textit{comp}\,(\gamma_i') \lrcorner} \llcorner \textit{type-ok}\,(\ulcorner \tilde{\gamma}_i \downarrow k \urcorner, \gamma_i'/k, \textit{var}\,(\overline{\gamma}))\,\sigma \lrcorner \urcorner \end{aligned} \tag{14}$$

otherwise. Note that the *type-ok* conditions do not contain extra variables, just references to the x_j. Therefore, σ_{EV} has no effect on these conditions.

This means that the goal has been proved for case (12), and that the term structure of $\gamma_i'\sigma$ equals that of $\tilde{\gamma}_i$ in cases (13) and (14). It remains to show E-equality on extra variable positions. So let v be an extra variable in γ_i' with $v = \gamma_i' w_v'$. It has to be proved that $\tilde{\gamma}_i w_v' =_E (\gamma_i'\sigma')w_v'$.

Because the condition of T_1 is E-equal to true, it must hold that

$$\textit{EV-match}\,\sigma_x =_E \mathsf{true} \tag{15}$$

If $i = j_v$ and $w_v' = w_v$, then

$$\begin{aligned} (\gamma_i'\sigma')w_v' &=_E (\gamma_i' w_v')\sigma' && [\text{positions in terms are not changed by substitutions}] \\ &=_E v\sigma' && [\text{definition of } w_v'] \\ &= (x_{j_v} w_v)\sigma_x && [\text{definition of } \sigma'] \\ &= (x_i w_v')\sigma_x && [\text{case hypothesis}] \\ &= \tilde{\gamma}_i w_v' && [\text{definition of } \sigma_x] \end{aligned}$$

Otherwise:

$$\begin{aligned} (\gamma_i'\sigma')w_v' &=_E (\gamma_i' w_v')\sigma' \\ &=_E v\sigma' \\ &= (x_{j_v} w_v)\sigma_x && [\text{up to here as in the other case}] \\ &= \tilde{\gamma}_{j_v} w_v && [\text{definition of } \sigma_x] \\ &= \tilde{\gamma}_i w_v' && [\text{prerequisite (15)}] \end{aligned}$$

Proof for (3.2):

Let $i \in [q]$. Since the **if** condition is rewritten to **true**, it must hold that $(\overline{B_i}\sigma)\,\sigma_x \xrightarrow{*}_{\mathcal{R}} \textsf{true}$. By definition, $\overline{B_i} = B_i \sigma_{EV}$, and $\sigma \circ \sigma_{EV} = (\sigma_{EV} \cup \sigma)$, because $\mathrm{dom}(\sigma) \cap \mathrm{dom}(\sigma_{EV}) = \emptyset$. Furthermore by definition $\sigma' = \sigma_x \circ (\sigma \cup \sigma_{EV})$, and hence $B_i\sigma' \xrightarrow{*}_{\mathcal{R}} \textsf{true}$. There are no contexts in this rewriting sequence; thus one obtains $B_i\sigma' \xrightarrow{*}_{\mathcal{B}} \textsf{true}$, and by correctness of $\mathcal{B}$ follows $B_i\sigma' =_E$ **true**.

Proof for (3.3):

One can deduce

$$\begin{array}{rcll}
\overline{\gamma}\,'\sigma' & = & (\overline{\gamma}\,'(\sigma \cup \sigma_{EV}))\,\sigma_x & [\,\text{definition of } \sigma'\,] \\
 & = & ((\overline{\gamma}\,'\sigma_{EV})\sigma)\,\sigma_x & [\,\text{see the proof for (3.2)}\,] \\
 & = & (\tilde{\gamma}\,'\sigma)\,\sigma_x & [\,\text{definition of } \tilde{\gamma}\,' \text{ in Def. 7.5}\,] \\
 & \xrightarrow{*}_{\mathcal{R}} & \gamma\text{''} & [\,\text{the } \textsf{then} \text{ part of } T \text{ gives the result}\,]
\end{array}$$

This rewriting sequence does not contain contexts; hence $\overline{\gamma}\,'\sigma' \xrightarrow{*}_{\mathcal{B}} \gamma$'', and by correctness of $\mathcal{B}$ follows $\overline{\gamma}\,'\sigma' =_E \gamma$''.

□

From the proved result, one immediately obtains normal form correctness, and by Lemma B.2 also one-step correctness.

Appendix C

Rewrite Rules vs. Deduction Rules

In Section 4.2.5.1, a method has been presented to include quantifiers as term constructors into the language by defining appropriate rewrite rules (including the $\lambda\sigma$-calculus). In this appendix, the use of these rules will be demonstrated in a small example proof with the Larch Prover, and compared to a proof of the same theorem that uses LP's deduction rules instead of quantification on term level.

C.1 The Example Theorem

In abstract form, the theorem to be proved states that the set of functions between two cpo's, when equipped with the "natural" ordering, also forms a cpo:

Theorem C.1

Let $\langle C, \sqsubseteq_C \rangle$ and $\langle D, \sqsubseteq_D \rangle$ be cpo's. Let the relation $\sqsubseteq$ on $F =_{df} C \to D$ be defined by

$$\forall f_1, f_2 \in F : f_1 \sqsubseteq f_2 \Leftrightarrow_{df} \forall c \in C : f_1(c) \sqsubseteq_D f_2(c)$$

Then $\langle F, \sqsubseteq \rangle$ is a cpo.

The proof of this theorem has three main parts (cf. [29]):

(1) Show that $\sqsubseteq$ is a partial ordering on F.

(2) Show that $\bot_F =_{df} \lambda c \in C.\bot_D$ is the least element in F.

(3) Show that for each chain $\langle f_i \rangle_{i \in \mathbb{N}}$ in F

$$\bigsqcup_{i \in \mathbb{N}} f_i = \lambda c \in C. \bigsqcup_{i \in \mathbb{N}} f_i(c)$$

C.2 Proof Using Only Rewrite Rules

First consider the automated proof that only uses rewrite rules. The specification of the input domains (as input for the tool gensig, see Appendix A.1) looks as follows:

```
% ====================================================================
% lcpo-sig: signature file for simulation of cpos with ls calculi
% Input file for gensig

.max var 4

.rules 0

.var
  c  : Dom1,
  d  : Dom2,
  f  : F,
  n  : Nat,
  sf : seqF

.types
  F  = Dom1 -> Dom2,
  seqF = Nat -> F

.explicit substitutions for
  Nat with 4 indices _n,
  Dom1 with 4 indices _c,
  Dom2 with 4 indices _d

.cpos
  (Dom1, <=) with bottom botc ruletype rewrite,
  (Dom2, <=) with bottom botd ruletype rewrite

```

gensig does not generate rules about quantifiers; so they have to be added individually:

```
% ============================================================
% lcpo-extras.lp : Additional rules dealing with quantifiers
% To be used with lcpo-sig-*.lp.

set name generalize
assert
  forall(tp, true) -> true
..

set name specialize_Nat
assert
  forall(_Nat, b) => (b # (n + id))
..

set name specialize_Dom1
assert
  forall(_Dom1, b) => (b # (c + id))
..

% To detect some nestings of quantifiers ranging over the same sort:
declare operator
  quantifier_clash : -> Bool
..

% Substitution distribution rules for quantified formulas:
% -------------------------------------------------------
% Substitution of one argument sort distributes only over the other sorts;
% if an attempt is made to distribute over the same sort as in
% (#)  forall(tp, b) # (x + u)    , x : tp
% this is an error ("quantifier clash"). The only way how such a term can come
% up is by specialization, and specialization corresponds to eliminating a quantifier.
% In the case (#), there must have been nested quantifiers of the same sort which is
% not allowed because of scoping problems (see Ch. 6).

set name subst_quant
assert
```

```
 forall(_Nat, b) # (n + u) -> quantifier_clash
 forall(_Nat, b) # (c + u) -> forall(_Nat, b # (c + u))
 forall(_Nat, b) # (d + u) -> forall(_Nat, b # (d + u))

 forall(_Dom1, b) # (n + u) -> forall(_Dom1, b # (n + u))
 forall(_Dom1, b) # (c + u) -> quantifier_clash
 forall(_Dom1, b) # (d + u) -> forall(_Dom1, b # (d + u))

 forall(_Dom2, b) # (n + u) -> forall(_Dom2, b # (n + u))
 forall(_Dom2, b) # (c + u) -> forall(_Dom2, b # (c + u))
 forall(_Dom2, b) # (d + u) -> quantifier_clash

 quantifier_clash # u -> quantifier_clash
..

% Universal quantifiers of different sorts may be swapped. The rules expressing this
% fact must be passive since they form a non-terminating TRS:
set activity off
set name forall_swap_from_Nat
assert
 forall(_Nat, forall(_Dom1, b)) -> forall(_Dom1, forall(_Nat, b))
 forall(_Nat, forall(_Dom2, b)) -> forall(_Dom2, forall(_Nat, b))
..

set name forall_swap_from_Dom1
assert
 forall(_Dom1, forall(_Nat, b)) -> forall(_Nat, forall(_Dom1, b))
 forall(_Dom1, forall(_Dom2, b)) -> forall(_Dom2, forall(_Dom1, b))
..

set name forall_swap_from_Dom2
assert
 forall(_Dom2, forall(_Nat, b)) -> forall(_Nat, forall(_Dom2, b))
 forall(_Dom2, forall(_Dom1, b)) -> forall(_Dom1, forall(_Dom2, b))
..
set activity on
```

The proof with LP runs with the following proof script:

```
% ============================================================
% lscpo.lp : Proof of the following theorem:
% Let (C,<=) and (D,<=) be cpo's, F = C -> D and for f1, f2 \in F
% define  f1 <= f2 iff (\forall c \in C: f1 . c <= f2 . c).
% Then (F,<=) is a cpo.
%
% cpo's are modelled with rewrite rules and ls calculi

% The proof starts from a frozen basic system:
thaw ~/LP/cpo/lcpo

% ==================================================
% Definition of the po-set (F,<=)

declare operators
  <= : F, F -> Bool
  chain : seqF -> Bool
  botf : -> F
..

% --------------------------------------------------
% extensionality on F:
set name ext_F
assert forall(_Dom1, f1 . _c1 = f2 . _c1) => (f1 = f2)

% --------------------------------------------------
% ordering:
set name ord_F
```

```
assert f1 <= f2 -> forall(_Dom1, (f1 . _c1) <= (f2 . _c1))

% ----------------------------------------------------
% <= is an ordering on F:

prove f <= f % <= is reflexive:
% immediate
[] % [reflexivity]

% ------------------------------------------
prove % <= is antisymmetric:
  ((f1 <= f2) & (f2 <= f1)) => (f1 = f2)
  by =>-method
..

% We always have to describe the effect of substitutions on all constants:
assert
  f1c # u -> f1c
  f2c # u -> f2c
..

% The claim follows directly from antisymmetry on Dom2:
instantiate b by (f1c . _c1) <= (f2c . _c1), c by _c1 in specialize_Dom1
instantiate b by (f2c . _c1) <= (f1c . _c1), c by _c1 in specialize_Dom1
instantiate d1 by f1c . _c1, d2 by f2c . _c1 in ord_Dom2
instantiate f1 by f1c, f2 by f2c in ext_F
[] % [antisymmetry]

% ----------------------------------------------------
prove % <= is transitive:
  ((f1 <= f2) & (f2 <= f3)) => (f1 <= f3)
  by =>-method
..

% We always have to describe the effect of substitutions on all constants:
assert
  f1c # u -> f1c
  f2c # u -> f2c
  f3c # u -> f3c
..

% The claim follows directly from antisymmetry on Dom2:
instantiate b by (f1c . _c1) <= (f2c . _c1), c by _c1 in specialize_Dom1
instantiate b by (f2c . _c1) <= (f3c . _c1), c by _c1 in specialize_Dom1
instantiate d1 by f1c . _c1, d2 by f2c . _c1, d3 by f3c . _c1 in ord_Dom2
[] % [transitivity]

% ----------------------------------------------------
% bottom:
set name bottom_F
assert
  botf . c -> botd
  botf # u -> botf
..

% ----------------------------------------------------
% chains:
set name chain_F
assert chain(sf1) -> forall(_Nat,(sf1 . _n1) <= (sf1 . (s(_n1))))

% ====================================================
% Claim 1: botf is the smallest element in F.

set name claim1

prove botf <= f
% immediate:
[] % claim1
```

```

% ==================================================
% Claim 2: There is a lub for each chain in F.

declare operator sfc : -> seqF

set name assumption
assert
  sfc # u -> sfc
  chain(sfc)
..

% --------------------------------------------------
% We have to prove that there exists an fc \in F with
% (1)  \forall n \in Nat : sfc . n <= fc
% (2)  \forall f \in F: (\forall n \in Nat: sf.n <= f) => fc <= f

% First step:

declare operator scd : Dom1 -> seqDom2

set name construction1
assert scd(c) . n -> (sfc . n) . c

% ----------------------------------------
set name scd_lemma
prove chain(scd(c))

% We must prove
%   forall(_Nat, ((sfc . _n1) . c) <= ((sfc . s(_n1)) . c))
% and have
% assumption.1: forall(_Nat,
%                forall(_Dom1, ((sfc . _n1) . _c1) <= ((sfc . s(_n1)) . _c1)))
% We need to swap the quantifiers ...

rewrite assumption with forall_swap_from_Nat

% ... in order to specialize the assumption:

instantiate
  b by forall(_Nat, ((sfc . _n1) . _c1) <= ((sfc . s(_n1)) . _c1))
  in specialize_Dom1
..
[] % [scd_lemma]

% ------------------------------
% Second step:

declare operator fc : -> F

set name construction2
assert
  fc # u -> fc
  fc . c -> lub(scd(c))
..

% ----------------------------------------
% Prove (1):
set name claim2

prove (sfc . n) <= fc

% Current subgoal:
% forall(_Dom1, ((sfc . n) . _c1) <= lub(scd(_c1))) == true

instantiate seqd by scd(_c1) in lub_Dom2
% .. result: lub_Dom2.1.1:
%            ((sfc . n) . _c1) <= lub(scd(_c1)) == true
```

```
165  [] % [claim2]
166
167  % ---------------------------------------
168  % Prove (2):
169
170  declare operator fc1 : -> F
171
172  set name assumption
173  assert
174    fc1 # u -> fc1
175    (sfc . n) <= fc1
176  ..
177
178  set name claim2_2
179
180  prove fc <= fc1
181
182  % Current subgoal: forall(_Dom1, lub(scd(_c1)) <= (fc1 . _c1))
183
184  % We need the verbatim form of the following lemma:
185  set immunity on
186  set name lemma1
187
188  prove forall(_Nat, (scd(c) . _n1) <= (fc1 . c))
189
190  set immunity off
191
192  % Follows from the assumption about sfc:
193  instantiate
194    b by ((sfc . _n1) . _c1) <= (fc1 . _c1),
195    c by c
196    in specialize_Dom1
197  ..
198  [] % [lemma1]
199  % -----------------------------
200
201  instantiate seqd by scd(_c1), d by (fc1 . _c1) in lub_Dom2
202
203  qed
```

Note that in order to have a correctly working $\lambda\sigma$-calculus, the effect of substitutions must be specified for all constants, even those that are generated by LP during the proof (e. g. see lines 44–48). The protection feature mentioned in Section 3.2.3.2 only saves this work in special cases, including the SOS simulation of Chapter 7.

C.3 Proof Also Using Deduction Rules

A proof for Theorem C.1 that makes use of LP's deduction rules instead of explicit quantifiers as term constructors starts from a similar domain specification. The only differences are that there are no "explicit substitution" rules required and that the "ruletype" is "deduction" instead of "rewrite". Moreover, it does not need additional definitions like those about quantifiers in the previous section. The proof script is the following:

```
% ==========================================================
% fcpo-ded.lp : Proof of the following theorem:
% Let (C,<=) and (D,<=) be cpo's, F = C -> D and for f1, f2 \in F
% define  f1 <= f2 iff (\forall c \in C: f1 . c <= f2 . c).
% Then (F,<=) is a cpo.
%
% cpo's are modelled with deduction rules.

% The proof starts from a frozen basic system:
thaw ~/LP/cpo/cpo

% ================================================
% Definition of the po-set (F,<=)

declare operators
  <= : F, F -> Bool
  chain : seqF -> Bool
  botf : -> F
..

% ------------------------------------------------
% extensionality on F:
set name ext_F

assert
  when (forall c) f1 . c = f2 . c
  yield f1 = f2
..

% ------------------------------------------------
% ordering:
set name ord_F

assert
  when (forall c) (f1 . c) <= (f2 . c)
  yield f1 <= f2
..

assert (f1 <= f2) => ((f1 . c) <= (f2 . c))

% ------------------------------------------------
% <= is an ordering on F:

prove f <= f % <= is reflexive

% This follows immediately from the definition (but deduction rules must
% be properly instantiated to make things work):

instantiate f1 by f, f2 by f in deduction-rule ord_F

[] % [reflexivity]

% ---------------------------------------
prove % <= is antisymmetric:
  ((f1 <= f2) & (f2 <= f1)) => (f1 = f2)
  by =>-method
..

% This follows from antisymmetry on Dom2:
instantiate f1 by f1c, f2 by f2c in rewrite-rule ord_F
instantiate f1 by f2c, f2 by f1c in rewrite-rule ord_F
instantiate d1 by f1c . c, d2 by f2c . c in rewrite-rule ord_Dom2
[] % [antisymmetry]

% ------------------------------------------------
prove % <= is transitive:
  ((f1 <= f2) & (f2 <= f3)) => (f1 <= f3)
  by =>-method
..

% This follows from transitivity on Dom2:
instantiate f1 by f1c, f2 by f2c in rewrite-rule ord_F
instantiate f1 by f2c, f2 by f3c in rewrite-rule ord_F
instantiate d1 by f1c . c, d2 by f2c . c, d3 by f3c . c in rewrite-rule ord_Dom2
[] % transitivity
```

```

% -------------------------------------------------
% bottom:
set name bottom_F

assert botf . c -> botd

% -------------------------------------------------
% chains:
set name chain_F

assert
  when (forall n) (sf . n) <= (sf . s(n))
  yield chain(sf)
..

assert chain(sf) => ((sf . n) <= (sf . (s(n))))

make passive chain_F

% =================================================
% Claim 1: botf is the smallest element in F.

set name claim1

prove botf <= f

instantiate f1 by botf, f2 by f in deduction-rule ord_F
[] % claim1


% =================================================
% Claim 2: There is a lub for each chain in F.

declare operator sfc : -> seqF

set name assumption
assert chain(sfc)

% -------------------------------------------------
% We have to prove that there exists an fc \in F with
% (1)  \forall n \in Nat : sfc . n <= fc
% (2)  \forall f \in F: (\forall n \in Nat: sfc.n <= f) => fc <= f

% First step:

declare operator scd : Dom1 -> seqDom2

set name construction1
assert scd(c) . n -> (sfc . n) . c

set name scd_lemma
prove chain(scd(c))

% We need the verbatim form of the following lemma:
set immunity on

% ---------------------------------------
% Lemma:

prove ((scd(c)) . n) <= ((scd(c)) . s(n))

set immunity off

instantiate sf by sfc in chain_F
instantiate f1 by (sfc . n), f2 by (sfc . s(n)) in ord_F
[] % Lemma
% ---------------------------------------
```

```

instantiate seqd by scd(c) in deduction-rule chain_Dom2
[] % [scd_lemma]

% ------------------------------
% Second step:

declare operator fc : -> F

set name construction2
assert fc . c -> lub(scd(c))

% ---------------------------------------
% Prove (1):
set name claim2

prove (sfc . n) <= fc

% ------------------------------
set name lemma

prove ((sfc . n) . c) <= (fc . c)

% This follows from the definitions:
instantiate seqd by scd(c) in lub_Dom2
[] % [lemma]
% ------------------------------

instantiate f1 by (sfc . n), f2 by fc in deduction-rule ord_F
[] % [claim2]

% ---------------------------------------
% Prove (2):

declare operator fc1 : -> F

set name assumption
assert (sfc . n) <= fc1

set name claim2_2

prove fc <= fc1

% Lemma needed in order to apply the definition of <= on F:

set name lemma1
prove (fc . c) <= (fc1 . c)

% ------------------------------
% Since fc . c -> lub(scd(c)), we prove:

set name lemma2
prove (scd(c) . n) <= (fc1 . c)

% Follows from the assumption about sf:
instantiate f1 by (sfc . n), f2 by fc1 in rewrite-rule ord_F
[] % [lemma2]
% ------------------------------

instantiate seqd by scd(c), d by (fc1 . c) in deduction-rule lub_Dom2
[] % [lemma1]

% ------------------------------
% Now the claim follows from the definition of <=:

instantiate f1 by fc, f2 by fc1 in deduction-rule ord_F

qed
```

C.4 Comparison of the Proofs

The structure of the two proofs is the same; both have been derived from the abstract proof. On the detail level, however, there are some differences. In the following, we refer to the proof in Section C.2 as the "rewrite proof" and to the proof in Section C.3 as the "deduction proof".

- The need to specify substitution rules for all constants makes the rewrite proof longer than would be desirable. This kind of internals should be hidden from the user.
- Since deduction rules are not always applied automatically, the deduction proof needs more explicit instantiations of rules than the rewrite proof. See e. g. the application of the definition of $\sqsubseteq$ in the proofs of claim 1 (rewrite proof, lines 94–96, and deduction proof, lines 101–104).
- Due to the much larger set of rewrite rules needed, the rewrite proof takes more time and needs more space than the deduction proof. Compare the statistics gained on a Sun SparcStation 10/40; for the rewrite proof it reads:

```
Recent            Success            Failure          Total
------         Count     Time     Count     Time       Time
Ordering          41     0.00         0     0.00       0.00
Rewriting        189     1.29      2134     1.62       2.91
Deductions         6     0.12        44     0.00       0.12
Unification        0     0.00         0     0.00       0.00
Prover                                                 0.87
GC's               2
Total time                                             8.65

Heap size  =     218,731 words
```

  and for the deduction proof:

```
Recent            Success            Failure          Total
------         Count     Time     Count     Time       Time
Ordering          42     0.01         0     0.00       0.01
Rewriting         93     0.39      1102     0.75       1.14
Deductions        15     0.17       350     0.13       0.30
Unification        0     0.00         0     0.00       0.00
Prover                                                 0.38
GC's               1
Total time                                             5.05

Heap size  =     143,261 words
```

So the performance of the deduction proof is considerably better than that of the rewrite proof. Since this is a very important aspect in an interactive proof environment, therefore the representation of quantified formulas with rewrite rules in the style of Section 4.2.5.1 has not been adopted for larger proofs.

Index

Italic page numbers refer to pages that contain a defining occurrence of the corresponding item.