Formally Verified Modular Semantics

Ken Madlener

ISBN: 978-90-8891-946-6

Typeset with LaTeX 2ε

Cover design:	Proefschriftmaken.nl || Uitgeverij BOXPress
Printed by:	Proefschriftmaken.nl || Uitgeverij BOXPress
Published by:	Uitgeverij BOXPress, 's-Hertogenbosch

Formally Verified Modular Semantics

PROEFSCHRIFT

ter verkrijging van de graad van doctor
aan de Radboud Universiteit Nijmegen
op gezag van de rector magnificus prof. mr. S.C.J.J. Kortmann,
volgens besluit van het College van Decanen
in het openbaar te verdedigen op donderdag 9 oktober 2014
om 12:30 uur precies

door

Ken Madlener

geboren op 24 maart 1983
te Zevenaar

Promotor:
prof. dr. Marko van Eekelen

Copromotor:
dr. Sjaak Smetsers

Manuscript commissie:
prof. dr. Bart Jacobs
prof. dr. Herman Geuvers
prof. dr. Peter D. Mosses Swansea University

Acknowledgements

This thesis would have not been possible without the support of many people, to whom I am greatly indebted. First and foremost, I would like to thank my supervisors Marko van Eekelen and Sjaak Smetsers for their continuous support and guidance during the course of my research and finalizing this thesis. I am especially thankful to Marko for arranging additional financial support for the continuation of my work after the original four-year term finished. I would also like to thank Peter Mosses for hosting me during my research visit to Swansea University and the many useful discussions I had with him. I thank Alexandra Silva for her sharp comments on the last chapter of this thesis and for helping me improve it. Next, I would like to thank the members of the manuscript committee, Bart Jacobs, Herman Geuvers and Peter Mosses for agreeing to referee my thesis.

My visit to Swansea was made possible by Bart Jacobs, the head of the Digital Security research group, which I was proud to be part of. I have had many pleasant (research) discussions with members of the Digital Security group and computer science researchers who I have met at conferences and research schools. In particular, I would like to thank Gerhard de Koning Gans for his strong support and being a good friend, Łukasz Chmielewski, my roommate and friend, Alejandro Tamalet, who was the co-author of the paper underlying the third chapter of this thesis, Bas Spitters for his clever insights, Robbert Krebbers for explaining the latest tricks in COQ to me, Martin Churchill, who was my roommate during my stay in Swansea, and Uri Klein for his encouragement.

Finally, I would like to thank my family and close friends for their everlasting support. However, this series of acknowledgements would not be complete without expressing eternal gratitude to my wife Krista, for her love and support.

Ken
London, Ontario, May 2014

Table of Contents

CHAPTER 1

Introduction

One cannot pay a code reviewer enough to do his job; some bugs simply escape the human eye. For mission-critical code, of which failure would have catastrophic consequences for human life, the possibility of the existence of bugs is simply unacceptable. Applying traditional software testing methods to this type of software does not always yield a satisfactory level of certainty about its correctness. As Edsger W. Dijkstra put it "program testing can at best show the presence of errors but never their absence" [DDH72]. One may resort to formal methods to mathematically prove software correctness, where "formal" refers to the act of writing statements in symbolic, unambiguous "mathematical" form. A correctness proof entirely eliminates the possibility of the existence of bugs, to the extent that they can be found within the model.

Enabling the formal verification of real-life software has been a long-standing goal in computer science. Most mainstream languages such as C, C++ or JAVA are designed from an engineering perspective, making their semantics (i.e. their "meaning") less amenable to concise mathematical description. Correspondingly, formal verification results in long proofs involving many case distinctions and side-conditions to be dealt with. Pen and paper proofs of software correctness are therefore deemed less trustworthy, as it is easy to make mistakes. This makes a strong case for the development of software tools to support formal verification. There are three main strands: model-based testing, model checking and theorem proving. Model-based testing entails the generation of test sequences from a model of the system behavior and requirements. The system under test is executed with these test sequences as inputs. Model checking exhaustively and automatically checks that a finite-state transition system (the model) meets

its specification. It generally aims at temporal properties such as the absence of deadlocks and starvation. A theorem prover provides an underlying logic (i.e. a formal language together with a number of axioms) to express theorems, and provides a mechanism to verify their proofs. The software verification problem at hand is formulated as a logical formula to be verified within the theorem prover.

All of the above techniques have in common that they help the user to find counterexamples that violate the specification of a program. Theorem proving is clearly the most general-purpose; its use is not limited to the verification of specific programs, but can also be used to reason about programming language theory, or even more generally, mathematics. The same reasons that the verification of individual programs benefits from theorem proving apply to the verification of results in programming language theory.

This thesis focuses on the topic of reasoning about programming language semantics with the assistance of theorem provers. It proposes a number of approaches to the formalization and analysis of semantics within theorem provers, inspired by a case-study on the verification of real-life software.

The guiding principle for the work in this thesis has been practical applicability. This goal is shared with the Laboratory for Quality Software (LaQuSo)[1], from which financial support was received. LaQuSo bridges the gap between academia and industry through the application of results of fundamental research. Chapter 2 of this thesis carries out a case-study on the verification of mission-critical software, which was the source of inspiration for the rest of the work in this thesis. Related work from Nijmegen is [vETHSU08, LSVE08, STT+09].

1.1 Programming Language Semantics

A programming language consists of a syntax and a semantics. Its syntax says what strings (of characters) are valid programs, and its semantics assigns meaning to valid programs. There are several styles to describe a semantics, of which the following two are relevant to this thesis:

- **Operational semantics [Plo81].** The operational semantics of a language is determined by a set of operational rules. These rules generate the set of valid computation steps which describe how one state may evolve into another state, thereby giving rise to a state transition system. In particular, when the set of computation steps is obtained by recursion over the programs, then one calls the operational semantics "structural", abbreviated as SOS.

[1] LaQuSo is a joint activity between Eindhoven University of Technology and Radboud University Nijmegen, see `http://www.laquso.com`.

- **Denotational semantics [Sco70, SS71].** This style describes the meanings of programs as mathematical objects (called denotations). As an example, a denotation of a program could be a partial function which maps input to output. An important aspect of denotational semantics is compositionality: the denotation of a program term should correspond to a combination of the denotations of its sub-terms.

One could say that operational semantics provides a description from an implementation point of view, and denotational semantics provides a more abstract perspective. These views should be consistent for the language in question, and indeed, this is a favorite textbook theorem, see e.g. [Win93]. Such assertions are called "meta-theoretic", because they make a verdict about the set of programs of the language as a whole.

It should be emphasized that both of the above styles of semantics pertain to a concrete programming language. One level higher in the abstraction ladder one can speak of meta-theoretic results that span entire classes of languages. Using the language of category theory, Turi and Plotkin [TP97] have formulated a theory, also called "bialgebraic semantics". At the core of their framework are the bialgebras, which carry both operational (structurality) and denotational (compositionality) characteristics. In bialgebraic semantics, both the model of the operational semantics and the model of the denotational semantics are derived from a shared bialgebra. These models are entirely independent of a concrete syntax, behavior or semantics, i.e. all language-specific details have been abstracted away.

A fair body of research has dedicated to developing a general theory for SOS, in which the concept of rule format plays a central rôle (see [MRG07] for an overview). These formats are syntactic restrictions on the operational rules, which guarantee some form of well-behavedness of their associated operational semantics. The GSOS rule format [BIM95], which originates from the theory of SOS, has been given a categorical interpretation by means of bialgebras, and it has been proved that the operational and denotational semantics generated from any set of GSOS rules are consistent for every input program [TP97]. This implies that the operational semantics enjoys the compositionality of denotational semantics.

1.1.1 Lightweight semantics

A couple of decades of research has proven that it is a complex task to develop a complete formal semantics of mainstream programming languages. Reference manuals of programming languages rarely provide any formal description other than the syntax; usually concepts are introduced informally and by means examples, see e.g. the Java Language Specification [JSGB98]. Indeed, mainstream

imperative programming languages exhibit computational behavior consisting of a mixture of side-effects (caused by changes to the state), ordinary or abrupt termination, and continuations (e.g. goto-statements). This makes it a serious challenge to model mainstream languages in a structured fashion, let alone reason about them.

Most textbooks on programming language semantics omit complexities by using idealized example languages. However, a restricted, so-called lightweight semantics of the program at hand might still be helpful to uncover bugs. The caveat is that one cannot entirely rule out the existence of bugs that might occur outside of what has been modeled. In Chapter 2 we carry out a case-study on real-life software written in C++ by means of a lightweight model, which nevertheless revealed bugs which were left unnoticed after manual code review. Chapter 3 presents a number of general theorems about assignments in recursive data structures, based on a lightweight semantics of an object-oriented language. We fully expect that these results also hold in languages such as JAVA and C++.

1.1.2 Modularity in semantics

If one insists on a complete description of a programming language semantics, then modularity is the holy grail. By structuring a language as a combination of language-independent components, modularity promotes better understanding through simplification. Neither operational nor denotational semantics in their original form provide support for modularity.

For denotational semantics, this issue was partially solved by Moggi's introduction of monads as a way to structure computational behavior [Mog89]. Potential for extension is provided by the use of monads "with a hole", also called "monad transformers". The operations associated with the original monad should be "lifted" through the applied monad transformer. Liang and Hudak [LHJ95] provide ad hoc liftings of operations for a collection of pairs of monad transformers, however, the number of combination possibilities grows quadratically in the number of monad transformers. Another approach is to generate monads from algebraic operations and equations, which has been investigated by Plotkin and Power, see e.g. [PP02].

On the operational side, Mosses [Mos04] has developed a modular variant, which will be the subject of Chapter 4. It can be viewed as a foundational basis for a component-based approach to semantics, which assigns meaning to higher-level language constructs (of mainstream languages) by means of syntactic combinations of language-independent (and therefore reusable) fundamental constructs. The PLANCOMPS project, a joint effort by three universities in the United Kingdom, is presently developing a library of fundamental constructs.

The building blocks which form the basis of a modular approach to semantics can be combined in different ways, leading to a variety of programming languages. This raises the question whether there is a relationship between such languages. We look at this question in Chapter 6 of this thesis. We use bialgebraic semantics as the foundation for this work. The level of abstraction provided by it allows us to express the relationship between two languages as a translation between bialgebras, which induces a translation between the corresponding operational semantics models. Moreover, its categorical roots permit a natural formalization in a theorem prover, e.g. based on the COQ formalization presented in Chapter 5.)

1.2 Theorem Provers

A theorem prover provides a logic (i.e. a formal language together with a number of axioms) and a program that mechanically verifies proofs of theorems written in that logic. Where pen and paper proofs usually have a share of "handwaving", a theorem prover is not satisfied until it has been provided with a complete proof, consisting all the steps necessary to derive that the conclusion follows from the axioms. As Jacobs et al. [JVDBH+98] put it "a theorem prover is like a skeptical colleague who patiently checks all details and is willing to do routine checks". The obvious advantage of mechanization is that it leaves no room for human error. This can help to make proofs about programming language semantics tractable, which are notoriously hard to reason about due to the amount of bookkeeping involved, caused by case distinctions, side-conditions and side-effects of the language in question. In brief, one could say that theorem provers are the embodiment of the principle that providing proofs generally is difficult, but checking them is "easy" (and can therefore be done mechanically).

There are two main branches in the world of theorem provers: the automated and the interactive theorem provers. Automated theorem provers operate with very little user intervention by relying on heuristics to discover proofs, but are inherently limited to domain-specific theories. Examples are ACL2 [KSM13], SAT-solvers, and the prover of the KEY project [BHS07] for a first-order dynamic logic of JAVA.

Interactive theorem provers (also called "proof assistants") are as the name suggests not fully automatic, but are in return often more generally applicable. This thesis applies the interactive theorem provers COQ [The12], mainly developed by Inria in France, and PVS [ORS92], developed by SRI International in the US, which both incorporate a functional programming language in their higher-order logic. COQ is directed towards the formalization of constructive mathematics, while PVS is more geared towards the verification of instances (e.g.

programs). COQ's logic is constructive, which makes it possible to seamlessly extract certified programs from its proofs. Both theorem provers include domain-specific strategies and decision procedures to automate certain types of proofs, effectively making them automated for certain fragments of their languages. Other interactive theorem provers which have been used in connection with programming language semantics are ISABELLE [Pau94], which is applied by e.g. [Hui01, Ohe01], and SPARKLE [DMvEP08] for the functional language CLEAN.

1.2.1 Formalization of semantics in theorem provers

We mention two large projects that formalize semantics in theorem provers. The LOOP project [JVDBH+98] in Nijmegen formalizes a significant fragment of sequential JAVA as a denotational semantics in both PVS and ISABELLE. Another notable project by Inria in France is the COMPCERT project [Ler09], which formalizes almost all of the ANSI C language in COQ, and moreover verifies the correctness of an optimizing compiler for this language. Outside formal semantics there exist examples where theorem proving has brought mistakes to light, e.g. in the original proof of the Kepler conjecture [HHM+10]. Chapter 2 of this thesis consists of a case-study that reveals some mistakes in thoroughly reviewed program code, by the help of PVS.

Even though research has come a long way over the past years, the price one pays for the generality offered by interactive theorem provers is that carrying out proofs can be quite laborious. An important factor is the choice of the set of definitions one decides to work with, as there are usually several degrees of freedom. One often discovers the advantages or disadvantages of a certain set of definitions after the fact. This makes the scalability of theorem proving a serious remaining challenge, and many academic papers have been dedicated to case-studies applying domain-specific techniques to formalize programming language theory and mathematics. All of the research chapters of this thesis, except for Chapter 6, are based on theorem prover formalizations.

It is an important distinction whether the formalization of a semantics is a "shallow embedding" or a "deep embedding". In the latter case, the semantics in the formalization consists of a data-structure for representing the syntax of the language itself, and a function which maps a program (an instance of the datatype for the language) to its semantics. In the former case, the semantics of the program at hand is merely the result of a direct translation into the theorem prover's logic (which could be obtained through the deep-embedding map). A meta-theoretic proof often enumerates over all the programs of the language, which requires a deep embedding if it is to be carried out in a theorem prover. Shallow embeddings are more suitable to analyze a concrete program. The LOOP project is a slight twist to this dichotomy. It provides a compiler to translate JAVA to PVS, thus

the resulting PVS code is a shallow embedding of the original program [vdBJ01]. The COMPCERT project provides a deep embedding of ANSI C; the correctness of the compiler is verified with respect to every possible input program.

The case-study of Chapter 2 verifies a concrete piece of software and is based on a shallow embedding. Chapters 3–5 are based on deep embeddings. A standard approach to encoding the set of valid computation steps of a structural operational semantics in COQ is to encode the operational rules as constructors of an inductive datatype. Inspired by the bialgebraic framework, in Chapter 5 we propose a different approach to the encoding of operational semantics, by providing a datatype for operational rules in general (subject to a particular format). In a broad sense, the former approach is a shallow embedding of operational rules, while the latter is a deep embedding. The bialgebraic approach taken in Chapter 5 enables formal reasoning about the relationship between programming languages.

1.3 Future Work

The added value of formal methods to the development of mission-critical software has gained recognition by the industry over the last decade. Several industry standards, such as DO-178C / ED-12C for aviation, EN-50128 for railways, and the more general IEC-61508, permit the use of tools that implement formal methods to augment or replace certain test-cases. However, a concern raised in the DO-178C standard is that it is not always clear that the formalization upon which a verification is based is conservative, that is, the formalization should not admit false results [Joy10].

This problem would be immediately solved if the software in question was written in a programming language with a formal semantics and native theorem prover support, such as COQ or PVS. However, conventional software development uses imperative languages, which prevents this from being a realistic option in the near future.

Use of esoteric language features creates situations in which the behavior of a program may be hard to predict for a programmer, and may even depend on the compiler used, due to bugs in the compiler itself or underspecification of the reference manual. For this reason, it is common that mission-critical components of software are written in a "safe" subset of the full language, which promotes clarity. This direction is taken for example by the SPARK language [Bar03], a subset of ADA, which is in fact specifically targeted at the development of mission-critical software.

In the presence of a modular semantics it should be a straightforward task to derive the semantics of subset languages. Moreover, since the full and subset languages are both obtained from the same building blocks, there is a formal

connection between both languages. The verification process can profit from not having to carry the weight of the full semantics, while the conservativity of the results can be proved formally within the same framework. For example, if the mission-critical component in question never throws an exception, one could analyze it in a sub-language which ignores exceptions. The results obtained in this thesis (Chapter 6 in particular) provide foundations for further work in this direction.

1.4 Overview and Contributions

We provide an overview of this thesis and highlight the contributions of the author. Apart from changes in layout, the content of each of the chapters is the original publication, with the exception of the last chapter which includes the proofs that were omitted in the published version for space reasons.

Chapter 2.
Ken Madlener, Sjaak Smetsers, and Marko van Eekelen. A formal verification study on the Rotterdam storm surge barrier. *Proceedings of the 12th International Conference on Formal Engineering Methods (IFCEM'10)*, volume 6447 of *Lecture Notes in Computer Science*, pages 287–302. Springer, 2010.

This chapter presents the results of the validation and verification of a crucial component of BOS, a large safety-critical system that decides when to close and open the Maeslantkering, a storm surge barrier near the city of Rotterdam in the Netherlands. BOS was specified in the formal language Z and model checking has been applied to some of its subsystems during its development. A lightweight model of the C++ code and the Z specification of the component was manually developed in the theorem prover PVS. As a result, some essential mismatches between specification and code were identified. The Z specification itself is also validated by the use of challenge theorems, to assess particular design choices. Tools have been used to exhaustively search for inconsistencies between the original specification and the challenge theorems, which led to deeper issues with the specification itself.

This chapter presents the results of a LaQuSo project commissioned by the Dutch Ministry of Transport, Public Works and Water Management and the Nuclear Research & consultancy Group (NRG). The verification and validation work was carried out by the author, who has also written most of the original publication [MSvE10]. This work was presented at the 12th International Conference on Formal Engineering methods in Shanghai, China. It has been informally communicated by the ministry that the issues that were found in the case-study have now been fixed.

Chapter 3.
Alejandro N. Tamalet and Ken Madlener. Reasoning about assignments in recursive data structures. *Proceedings of the 13th Brazilian Symposium on Formal Methods (SBMF'10)*, volume 6527 of *Lecture Notes in Computer Science*, pages 161–176. Springer, 2010.

This chapter presents a framework to reason about the effects of assignments in recursive data structures. It defines an operational semantics for a core language based on Bertrand Meyer's ideas for a semantics for the object-oriented language EIFFEL [Mey03]. A series of field accesses, e.g. $f_1 \cdot f_2 \cdot \ldots \cdot f_n$, can be seen as a path on the heap. This chapter provides rules that describe how these multidot expressions are affected by an assignment. Using multidot expressions to construct an abstraction of a list, it proves the correctness of a list reversal algorithm. This approach does not require induction and the reasoning about the assignments is encapsulated in the mentioned rules. This chapter also discusses how to use this approach when working with other data structures and how it compares to the inductive approach. The framework, rules and examples have been developed and their correctness was proved in PVS.

The work for this chapter was initiated by Alejandro N. Tamalet in his master's thesis [Tam06]. The author and Tamalet have equally contributed to the PVS formalization and writing the published paper [TM10]. This work was presented at the 13th Brazilian Symposium on Formal Methods in Natal, Brazil.

Chapter 4.
Ken Madlener, Sjaak Smetsers, and Marko van Eekelen. Formal component-based semantics. *Proceedings of the 8th Workshop on Structural Operational Semantics (SOS'11)*, volume 62 of *Electronic Proceedings in Theoretical Computer Science*, pages 17–29. 2011.

Component-Based Semantics is a solution proposed by Mosses [Mos09] aimed at improving the scalability of semantics of programming languages. It is expected that this framework can also be used effectively for modular meta-theoretic reasoning. This chapter presents a formalization of Component-Based Semantics in the theorem prover COQ. It is based on Modular SOS, a variant of SOS, and makes essential use of dependent types, while profiting from type-classes. This formalization constitutes a contribution towards modular meta-theoretic formalizations in theorem provers. As a small example, a modular proof of determinism of a mini-language is developed.

This work consists of the author's own contributions published in [MSvE11], and was presented at Structural Operational Semantics 2011 in Aachen, Germany. The

author received guidance from Marko van Eekelen and Sjaak Smetsers in writing the published paper.

Chapter 5.
Ken Madlener and Sjaak Smetsers. GSOS Formalized in Coq. *Proceedings of the 7th International Symposium on Theoretical Aspects of Software Engineering (TASE'13)*, pages 199–206. IEEE, 2013.

Structural operational semantics provides a well-known framework to describe the semantics of programming languages, lending itself to formalization in theorem provers. The formalization of syntactic SOS rule formats, which enforce some form of well-behavedness, has so far received less attention. GSOS is a rule format that enjoys the property that the operational semantics and denotational semantics, both derived from the same set of GSOS rules, are consistent. This chapter formalizes the underlying theory in the theorem prover COQ, and proves the consistency property, also known as the adequacy theorem. The inspiration for our work has been drawn from the field of bialgebraic semantics.

This work consists of the author's own contributions published in [MS13], and was presented by the author at The 7th International Symposium on Theoretical Aspects of Software Engineering in Birmingham, UK. The author received guidance from Sjaak Smetsers in writing the published paper.

Chapter 6.
Ken Madlener, Sjaak Smetsers and Marko van Eekelen. Modular Bialgebraic Semantics and Algebraic Laws. *Proceedings of the 17th Brazilian Symposium on Programming Languages (SBLP'13)*, volume 8129 of *Lecture Notes in Computer Science*, pages 46–60. Springer, 2013.

The ability to independently describe operational rules is indispensable for a modular description of programming languages. This chapter introduces a format for open-ended rules and proves that conservatively adding new rules results in well-behaved translations between the models of the operational semantics. Silent transitions in our operational model are truly unobservable, which enables one to prove the validity of algebraic laws between programs. Algebraic laws are a variant of bisimulation relations between open terms, relating the branching structures of the state transition systems corresponding to the open terms. We also show that algebraic laws are preserved by extensions of the language and that they can be instantiated. The work presented in this chapter is developed within the framework of bialgebraic semantics.

The inspiration for the work in this chapter was drawn from the author's consultation with Mosses at Swansea University. It consists of the author's own

contributions. This chapter is based on a paper accepted for presentation and publication at the 17th Brazilian Symposium on Programming Languages, which was held in Brasília, Brazil. The author received guidance from Marko van Eekelen and Sjaak Smetsers in writing the accepted paper [MSvE13]. Credits are also due to Alexandra Silva whose sharp comments helped improve the chapter.

CHAPTER 2

A Verification Study on the Rotterdam Storm Surge Barrier

Abstract. This chapter presents the results of the validation and verification of a crucial component of BOS, a large safety-critical system that decides when to close and open the Maeslantkering, a storm surge barrier near the city of Rotterdam in the Netherlands. BOS was specified in the formal language Z and model checking has been applied to some of its subsystems during its development. A lightweight model of the C++ code and the Z specification of the component was manually developed in the theorem prover PVS. As a result, some essential mismatches between specification and code were identified. The Z specification itself is also validated by the use of challenge theorems, to assess particular design choices. Tools have been used to exhaustively search for inconsistencies between the original specification and the challenge theorems, which led to deeper issues with the specification itself.

2.1 Introduction

Humans increasingly rely on automation, the advantage being that a computer can make unprejudiced decisions, not being influenced by mood or other conditions. It is therefore often considered to be safer to let a computer be in control. A study showed that this is the case for the Maeslantkering, a storm surge barrier near

the Dutch city of Rotterdam. The barrier consists of two hollow floating walls, connected by steel arms to pivot points. Each of these arms is as large as the Eiffel Tower. The barrier operates fully autonomously and is therefore sometimes called the largest robot in the world.

A control system called BOS (Dutch: Beslis & Ondersteunend Systeem) makes the decisions about closing and opening the barrier. This system was developed by CGI Nederland B.V.[1] for Rijkswaterstaat (RWS) – a division of the Dutch Ministry of Transport, Public Works and Water Management. When BOS predicts that the water level will rise so high that it could flood the city of Rotterdam, it has the responsibility to close the barrier. This makes BOS a safety-critical system. On the other hand, Rotterdam is a major port, so the barrier should close only when really necessary. Unnecessarily closing the barrier costs millions of Euros because of restricted ship traffic.

It is often loosely said that even computers make mistakes. Verification and validation efforts must be undertaken to reduce the severity of this risk. The IEC-61508 standard [IEC96] recommends the use of formal methods for safety-critical systems. The BOS project adhered to the standard by using the formal language Z [Spi89] in combination with the ZTC type checking tool [Jia94] for specification. Some of its subsystems have been model checked using SPIN [Hol97]. This level of formal support during the development turned out to be a cost-effective approach for the BOS project [TWC01]. The system has been in operation since 1997 and a test closure is performed annually. On November 11$^{\text{th}}$, 2007 BOS closed the barrier on its own for the first time because of a combination of high tide and storm.

With the advent of theorem provers and powerful decision procedures, more rigorous approaches to formal verification come within reach, even for large software systems such as BOS. The Nuclear Research and consultancy Group (NRG) and RWS commissioned Radboud University Nijmegen to carry out a 3-month project to do a case-study in applying formal verification to a part of BOS. NRG's field of operation is nuclear applications, where safety standards are even higher. Their objective was to investigate if formal verification can increase confidence in the safety of software. The conducted work is carried out on a crucial component of BOS selected by experts at RWS and CGI of 800 lines of sequential C++ code. The code has been verified against the existing Z specification, and the specification itself has been validated.

Given the man hours available for the actual verification and validation work in the project (roughly 2 months, one PhD student), the task was challenging and presented several hurdles that had to be taken. Since the code had not been formally verified before, there was no strict correspondence between code and

[1] Called CMG Den Haag B.V. at the time of the development of BOS.

specification. To make the two fit together and to isolate the selected component, an understanding of the code that goes deeper than what is written in the formal specification is required. The formalization has been carried out in the PVS theorem prover [ORS92] by means of a manually developed lightweight semantics.

The development of the BOS system is an effort linking several disciplines. This induces the risk of interface problems caused by misunderstandings that undermine robustness. Because a specification is the result of a translation of the designer's intuition into a formal language, it cannot be formally verified. We were nevertheless able to semi-systematically validate the specification with the help of challenge theorems. A challenge theorem is a property that from the point of view of the person performing the validation is plausible and should hold. With our understanding of the domain we were able to formulate several challenge theorems. PVS and its testing features enabled us to discover some extreme situations in which the component might behave suspiciously.

The main contribution of this chapter is an exposition demonstrating how lightweight modeling of industrial C++ code may enable one to find mismatches between specifications and code, based on a concrete case-study. Although we are confident that our PVS model faithfully reflects the semantics of the source code, future projects that verify safety-critical software may desire formal guarantees. We sketch an approach in this chapter to resolve issues regarding the connection between the modeled semantics and the true semantics of the component as future work. This chapter also demonstrates how challenge theorems can assist one in validating specifications.

This chapter is organized as follows. Section 2.2 gives a description of the selected component. The model is described in Section 2.3. Validation of the specifications is discussed in Section 2.4. The case-study is evaluated in Section 2.5. Related work is discussed in Section 2.6 and future work on ensuring correspondence between model and executed code is discussed in Section 2.7. Section 2.8 concludes.

2.2 The Considered Component: DEW

BOS makes decisions based on water level predictions computed from hydrological and meteorological information. When the expected water levels are considered to be too high, then it starts the procedure to close the barrier: commands to a system that operates the valves, pumps and motors of the barrier are sent and authorities are informed (via fax and pager). While in operation, BOS runs a script that continuously makes calls to native functions (written in C++). These functions can be categorized into sending out a command, a status request and

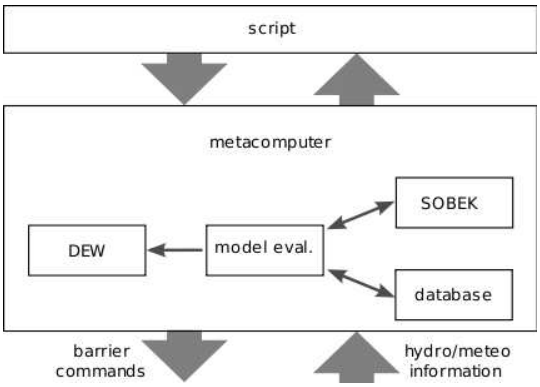


Figure 2.1: A schematic representation of the relevant BOS components.

a request for a decision. The component we consider in this study, determine excessive water level (DEW), is one of the functions that make the decisions.

A system called SOBEK, developed by Delft Hydraulics[2], generates the water level predictions. It computes model-based predictions for the next 24 hours in steps of 10 minutes for three physical locations: Rotterdam, Dordrecht and Spijkenisse (these are cities in the Netherlands). With each call, DEW receives a number of parameters from the script: for each of the locations a maximum water level and the desired evaluation interval of the predictions. To reduce the load on SOBEK, predictions are stored in a database. DEW obtains the predictions from SOBEK via the hydraulic-model evaluator. The model evaluator first tries to look up the requested prediction in the database, and if it does not exist, issues a new request to SOBEK.

DEW searches the predictions for a point in time where the maximum water level at one of the locations is exceeded by the prediction and it raises a flag if an excess is found. Some threshold is taken into account so that the barrier will only close when the excess is critical. An excess is considered to be *critical* when one of the maximum water levels is exceeded, and 20 minutes later the predicted water level is at least as high. In all other cases DEW will tell the script not to close the barrier.

Particular about the script language is that it works with lifted values. For example, a boolean in the script can be either `true`, `false` or `undetermined`. This also goes for the three maximum water levels and evaluation period provided by the script.

[2]Now called Deltares.

2.2.1 Z specification

The Z specification of DEW, the decision component, is composed of a number of Z schemas using the standard Z operators piping ($\gg$) and disjunction ($\vee$). For presentation purposes we have translated the original Dutch names to English and slightly simplified the formulas. The main schema consists of the following composition:

$$\begin{aligned} DEW \;==\; & SetEvaluationParams \gg DetModelEvaluation \gg \\ & (EvaluationFailed \vee CoreDEW) \end{aligned}$$

SetEvaluationParams puts the parameters from the script into the appropriate form. The output is passed on using the piping operator to *DetModelEvaluation*, which specifies the behavior of the hydraulic-model evaluator. If *DetModelEvaluation* is successful, then the schema *CoreDEW* is chosen, where the real work of DEW takes place, otherwise, *EvaluationFailed* is chosen.

Schema CoreDEW. The schema *CoreDEW* takes two input parameters: *LocList* and *Interval*. *LocList* is a partial function from locations to a maximum water level (it is partial because sometimes not all locations have a determined maximum water level). The prediction data is obtained from an external schema that represents the model evaluator. This is represented as a table of records *LocSeqs* (for predictions sequences per location) in which each record consists of a location l (which corresponds to a location in the domain of *LocList*), a prediction *vals* (a function mapping time to the predicted water level), and a point in time *TBegin* at which the prediction starts. Both time and water levels are represented as natural numbers. The result of *CoreDEW* consists of a flag *Excess* indicating whether an excess will occur, and a point in time *ExpTime* corresponding to the first critical excess. Since it may be the case that such a critical excess does not exist (even if *Excess* is true), this value is lifted. The condition for the existence of an excess is defined in the design document as:

$$\begin{aligned} & return!.Excess = \\ & \quad \textbf{if } (\exists\, s : LocSeqs;\ l : \mathrm{dom}\ LocList;\ i : \mathbb{N} \bullet \\ & \qquad i \in 1..(\#s.vals) \wedge s.loc = l \wedge (s.vals\ i).val > (LocList\ l)) \\ & \quad \textbf{then } ExDet\ true \textbf{ else } ExDet\ false \end{aligned}$$

It says that if there is an index i in the domain of $s.vals$ which is greater than the maximum water level at location l given by *LocList* l, then there is an excess, otherwise there is no excess.

Critical excesses are excesses such that 20 minutes later the water level is at least as high. (In the BOS documentation these are therefore called *non-decreasing* excesses.) The critical excesses have to lie within the predetermined evaluation interval. The starting time is obtained by taking the minimum of begin times of all locations.

$$TStart == \min \{l : LocSeqs \bullet l.TBegin\}$$

For the sake of completeness, we also give the definition of the existence of a critical excess. The details, however, are not important for the rest of this chapter.

$$\begin{array}{l}
\textbf{let } ExcessTimes == \\
\quad \{s : LocSeqs;\ l : \mathrm{dom}\ LocList;\ i : \mathbb{N} \mid i \in 1..(\#s.vals) - 2 \land \\
\quad\ s.loc = l \land (s.vals\ i).val > (LocList\ l) \land \\
\quad\ (s.vals\ (i+2)).val \geqslant (s.vals\ i).val \bullet \\
\quad TStart + (i-1) * 10\} \cap Interval \bullet \\
(ExcessTimes = \emptyset \Rightarrow return!.ExpTime = EtUndet) \land \\
(ExcessTimes \neq \emptyset \Rightarrow return!.ExpTime = EtDet(\min\ ExcessTimes))
\end{array}$$

In the above, the indices i of critical excesses are mapped to absolute time (using Z's set comprehension notation). The resulting set is intersected with the evaluation interval and checked to be nonempty. Recall that for an excess to be critical, the water level has to be 20 minutes later at least as high (hence, $i+2$).

A careful reader might have noticed two issues with the above specification. First of all, it might seem suspicious in the first definition the interval is not taken into account. This is a correct observation. Secondly, *TStart* is chosen to be the first *TBegin* time of *all* selected prediction sequences. In *ExpTime*, *TStart* is then used as the same offset for every prediction sequence, even if it has a different *TBegin*. Both issues have been resolved in the C++ code. In order to be able to prove consistency between Z specification and C++ code, we have fixed these flaws in the Z specification (see Sections 2.3.2 and 2.3.3).

2.3 Formal Analysis

In this section we describe how a model has been created out of the code of DEW and how it was checked and proved to be consistent with the specification. The use of C++ in BOS is according to "safe" coding guidelines (see e.g. [Hol06]); there was no heavy use of object orientation, no pointer arithmetic, etc. This permitted us to develop a lightweight PVS model of the C++ code. The PVS theorem prover was chosen for the very practical reason of locally available expertise. There exist

theorem provers for Z such as Z/Eves [Saa97], but our focus is on the development of a lightweight semantics of the component's C++ code and the Z specification could easily be transliterated into PVS. In short, PVS is based on higher-order logic with dependent types and predicate subtyping. We do however not make extensive use of these distinctive features of PVS and we believe that many other theorem provers would also suit the job.

2.3.1 Translation of C++ to PVS

Datatypes. BOS works with lifted types to represent the possibility of information being undetermined. A number of subclasses are derived from the C++ class `LType` to represent these types. Although PVS has a standard facility to lift types, we model these classes as records to stay close to the original code. The names of lifted types are prefixed by *L* as a naming convention.

$$LType : \mathbf{TYPE} = [\#fDet : bool\#]$$
$$LInt : \mathbf{TYPE} \quad = LType\ \mathbf{WITH}\ [\#iVal : int\#]$$

Setting the value of a lifted integer to 5, i.e. `li.Set(5)`, would be modeled using

$$LInt\_Set\ (i : int) : LInt = (\#fDet := TRUE, iVal := i\#)$$

as s **WITH** $[somevar := LInt\_Set\ (5)]$, where s is the current state. Whenever a set-function is used, the determined-flag is automatically set to `true`.

As mentioned in Section 2.1, we adopt a lightweight approach to the target program's semantics in this chapter. In our PVS code, C++ integers are modeled as unrestricted integers, which is in line with how integers are perceived in Z. Since in reality the C++ integers are a finite set, this leaves the possibility that the C++ model in PVS and Z specification match, while the behavior of the actual C++ code diverges from the Z specification. See also the discussion in Section 2.7.

Functions. The C++ functions have been translated into separate PVS theories carrying the same names and taking the same arguments. Arguments that are passed on by reference are modeled by making local PVS variables (using **LET** ... **IN**) of the function arguments before the beginning of the function body and then updating the returned variables at the end of the function body. For example, the header of the C++ function `CoreDEWC` (corresponding to the Z schema *CoreDEW*)

```
static flag CoreDEWC (
  const LInt []         cLocList,   // in
  const Interval&       cInterval,  // in
  flag&                 Excess,     // out
  LTime&                ExpTime     // out
)
```

is translated into a PVS function of type

$$
\begin{array}{l}
\mathit{CoreDEW\_pvs}\ (\\
\quad \mathit{cLocList} : [\mathit{Loc} \rightarrow \mathit{LInt}], \\
\quad \mathit{cInterval} : \mathit{Interval} \\
) : [\mathit{flag}, \mathit{flag}, \mathit{LTime}]
\end{array}
$$

The representation of the type `Interval` closely follows the declaration in C++. This works for functions have no side-effects. The functions were all annotated with in/out flags, so these kind of conversions were straightforward (but, indeed, not formally sound). Each function has been modeled as a separate PVS theory.

In PVS, functions are always total. If PVS is not able to deduce totality by itself, it will generate a proof obligation. This way it is enforced that the execution model terminates, which was not a problem for the considered code of DEW.

For-loop. At the heart of DEW's code lies one for-loop. This for-loop runs through the prediction for a single location and searches for excesses within the specified evaluation interval. It makes use of an object `s` which resembles the Z defined prediction sequence (see Section 2.2.1). It also assumes that `MaxLevel` contains the maximum water level of location `s.loc` (selected from the input parameter `cLocList`).

```
for(int item = 0; !ExpTime.IsDetermined () &&
  (item < s.GetSize() - 2); ++item) {
  if(TLoop >= TBegin && TLoop <= TEnd) {
    const int next_wl = s.vals.ElementAt(item).val;
    if(next_wl > MaxLevel) {
      Excess = TRUE;
      // is this a critical excess?
      if(s.vals.ElementAt(item + 2).val >= next_wl)
        ExpTime.Set(TLoop);
    }
  }
  TLoop += stepsize;
}
```

`TBegin` and `TEnd` are the boundaries of `Interval`. Note that the outer conditional corresponds with the intersection in the Z definition of critical excesses. Two variables are being set: `Excess`, a boolean which indicates whether an excess occurs in the prediction, and `ExpTime`, the first time at which a critical excess occurs. This for-loop has been modeled as a recursive function. A measure has to be supplied with a recursive function, which tells PVS that the number of iterations left at some point will be 0.

2.3.2 Communication with the hydraulic-model evaluator

To reduce the load on the prediction engine, i.e. SOBEK, previously computed predictions are stored in a database. The hydraulic-model evaluator acts as a proxy for SOBEK and the database. Calls to the evaluator have to supply the current evaluation time (of the synchronous script). The evaluator then checks whether the prediction already exists in the database, and if it is up to date. If this is the case, it returns the existing prediction. If not, it issues a request to SOBEK to compute a new prediction. The new prediction is then stored in the prediction database with a run-id. The prediction itself is not returned by the evaluator, but only the run-id and a status flag. The request may fail for several reasons: the time of the prediction requested is invalid, SOBEK is in an error state, etc. The status flag `RunStatus` indicates whether the request was successful. If this flag is true, the specification says we may assume that an entry with the run-id exists in the database. We have used this informal information in our PVS model.

From a functional point of view, the model evaluator, SOBEK and the database can be considered as a single entity. Because we do not verify the model evaluator, we treat it as something that has arbitrary output (including its success or failure). The following C++ code obtains the predictions from the database:

```
if(!DB.SelectPrediction(loc, Run.Get()))
{ <...> /* error */ }
else {
  const LocSeq& s = DB.GetLocSeq();
  <...>
}
```

The select operation points `DB` to the right record by changing its internal state. The actual prediction is obtained in the code by `DB.GetLocSeq()` and fed into the for-loop. The code shown above uses the location and run-id (`loc`, `Run.Get()`) as a key to return a unique prediction. In the Z specification, the database may contain several predictions starting at various times. To avoid obvious inconsistencies, we have chosen to change the Z specification so that it also contains a unique prediction per key. This modification was made in accordance with a system expert.

To simulate all possible outputs of the model-evaluator (and therefore also SOBEK and the database), we use uninterpreted variables to represent success or failure of the model evaluator and the database output. The correctness proof has to take every possible value of the variables in account.

$$RunStatus : bool$$

$$Seq : \mathbf{TYPE} = [\#n : Length, \\ val : [below\ (n) \rightarrow height],$$

$$
\begin{aligned}
&\qquad\qquad TBegin : Time \\
&\qquad\quad \#] \\
&LocSeqs : [[Loc, nat] \rightarrow Seq]
\end{aligned}
$$

The model evaluator is started once per execution of DEW, so we let *RunStatus* to be a constant. The C++ functions `SelectPrediction` and `GetLocSeq` are represented in PVS by the the following functions:

$$
\begin{aligned}
&DB\_SelectPrediction : bool = RunStatus \\
&DB\_GetLocSeq\ (loc : Loc, RunId : nat) : Seq = LocSeqs\ (loc, RunId)
\end{aligned}
$$

We do not have an explicit state of `DB` in the model, so we have to supply *loc* and *RunId* as arguments to *DB_GetLocSeq*. With this way of modeling we would have to supply the parameters that initialize a particular object with every function call that is made on it. This is not a useful solution for code which uses many objects with internal states, but works well for the code of DEW.

2.3.3 Verification

The result of a formal verification is either one or more discrepancies between specification and code or a proof that (model of) the code implements the specified behavior. It is common that most discrepancies (if there are any) are already found during the process of modeling. While modeling DEW we found two discrepancies, and we found a third during formal verification itself.

The first problem that was found as a result of modeling is the following. For regular excesses, the evaluation interval was not being taken into account in the specification (see Section 2.2.1), but was in the implementation. This is a design decision of the implementer which happened to be correct. However, the fact that the specification was not updated accordingly, suggests that the implementer was not aware that he was actually fixing something. The second problem found during modeling had to do with the prediction database. In the Z specification, the database may store multiple predictions per location and run-id. In the C++ code, only one prediction per location/run-id is considered (which is obtained from the database, see Section 2.3.2). An assumption that we were not aware of might resolve this issue, but this was not obvious from the specification nor its guiding text.

The main part of the formal verification itself requires proof of the following lemma that equates the specification and implementation (as functions):

$$
\begin{aligned}
&correctness : \textbf{LEMMA} \\
&\quad \forall\ (param : CoreDEWIn) : \\
&\qquad CoreDEWZ\_pvs\ (param) = CoreDEWC\_pvs\ (param)
\end{aligned}
$$

Both return a triple $[flag, flag, LTime]$ (the return flag, *Excess*, *ExpTime*, resp.) which all have to agree under P and every possible $param\}$.

Proof approach. Instead of directly trying to prove the above lemma in PVS, we chose to first develop a toy model in the model checker SPIN [Hol97] to experiment with. The reason is that when developing non-trivial formalizations in a theorem prover, often a considerable amount of time is devoted to debugging specifications and theorems. Using SPIN, we could try out candidate invariants by putting `assert` commands in the model code. The number of possible inputs was incremented until enough confidence in the correctness of the model was obtained. This essentially comprises playing with the ranges of *length*, *height* and the maximum water levels. While doing so, we found a flaw in the code: the last two elements of the array containing the predictions are not being taken into account. After fixing this issue, no other issues were found and the theorem could be proved in the first attempt with approx. 2000 proof commands.

2.4 Validation of the Specification

Interaction of components that are developed by engineers of several disciplines poses the risk of problems with interfacing as a result of misunderstandings. The standard way to validate a specification is picking different examples of system behavior allowed by that particular specification, and see if they fit with the intuition of the designer. This is often infeasible to do exhaustively, hence, this method is sound, but not necessarily complete.

To assess the design choices made in the specification in a more systematic way, we have formulated *challenge theorems*. A challenge theorem is a statement that from the point of view of the person performing the validation might be a valid consequence of the specification. By checking the consistency of a challenge theorem with the existing specification, which may be done automatically with the help of tools, counterexamples are generated that demonstrate how the existing specification and the challenge theorem diverge. These examples may serve as concrete material for discussion with the experts. Accordingly, the existing specification has to be altered or the domain understanding of the person performing the validation has to be improved.

Stating the most general system property, and focusing it gradually on DEW forces us to identify what assumptions we make about other components. For closing the barrier, these are based on the following general theorems:

- If there will be a flood, then the barrier will be closed in a timely manner.
- If the barrier is closed, there would have otherwise been a flood.

For DEW in particular, this means that we have to assume that the predictions are correct and if it decides to close the barrier, then this is properly delegated by BOS as a command to the barrier. The theorems for DEW become:

- If there is an excess in the available predictions that is critical, then DEW will decide to close.
- If DEW has decided to close, there would otherwise have been a critical excess.

We have verified in Section 2.3 that w.r.t. the existing definition of critical excess that DEW behaves correctly. In Section 2.4.2 we validate the definition of critical excess itself.

2.4.1 Decision based on incomplete information

We have for the moment ignored the possibility of failing sensors. This makes the actual case a bit more complicated, because if no prediction data is available at all or is only partially available, then DEW can simply not make a sensible decision and should (and does) therefore raise an alarm. However, when the predictions are only partially available, it should still decide to close the barrier if needed. With each call to DEW, for each of the three locations a maximum water level is supplied which may be undetermined. If one of these locations is undetermined, and no excesses were found on the determined locations, then DEW would say there is no excess. Both Z and C++ agree in this, but this behavior is obviously not safe. The experts agreed with us that this is an issue, but the way DEW is called precludes this problem.

Another issue we found is that the evaluation period (chosen by the script) may not necessarily be within the period of which the predictions are available. In this case DEW would look at the intersection of the two (see the Z specification for critical excesses in Section 2.2.1), which again is not safe.

2.4.2 Critical excesses

In order to assess the DEW's specified conditions to close the barrier, we have formulated two alternative definitions of a critical excess. These variations were motivated by real-life, but possibly rare, scenarios. Without loss of generality we focus on predictions for one location, ℓ say.

As an extreme example, consider a tsunami. In the predictions this would typically look like a short but high peak. Based on this idea, we have defined our own criterion in (2.1) that expresses that the "volume" (to be loosely interpreted)

exceeding the maximum water level may not surpass a predefined amount M (here P_ℓ and $\max_\ell$ denote the prediction and maximum water level respectively):

$$\{i : \mathbb{N} \mid \sum_{k=i}^{\#\mathrm{dom}\,\mathrm{P}_\ell} \max(0, \mathrm{P}_\ell(k) - \max_\ell) > M\} \tag{2.1}$$

Another situation one typically wants to avoid is a possibly slight, but continuous excess of the maximum water level:

$$\{i : 1 \ldots \#\mathrm{dom}\,\mathrm{P}_\ell - T \mid \forall j \in 1 \ldots T : \mathrm{P}_\ell(i+j) > \max_\ell\} \tag{2.2}$$

There are obvious cases in which these two variants do not coincide. In (2.3) below, the original definition for critical excess as in the Z specification (see Section 2.2.1) is expressed:

$$\{i : \mathbb{N} \mid i \in 1 \ldots \mathrm{length}_\ell - 2 \wedge \mathrm{P}_\ell(i+2) \geqslant \mathrm{P}_\ell(i) > \max_\ell\} \tag{2.3}$$

All three versions have been entered into PVS and lemmas (although unprovable) that express their equivalence were formulated. These lemmas implicitly quantify over the PVS variable *LocSeqs* (see Section 2.3.2) and the maximum water level. We used a random testing feature [Owr06] in PVS that allows the user to audit the truth of simple lemmas by trying out a number of instances.

Comparing (2.1), (2.2) and (2.3) gave us an example in the lines of Figure 2.2, where the water level increases rapidly and then steadily decreases until it is below the maximum water level (indicated by the dotted line). To search for different classes of counterexamples, we added the premise that a rapid increase followed by a slow decrease until maximum water level does not occur in the predictions of *LocSeqs*. This resulted in another type of examples, where the water level first increases rapidly, drops and then increases again. It turned out that in those cases the barriers would close too late, as shown in Figure 2.3. It was not clear whether these scenarios may occur in reality or are precluded by undocumented assumptions about behavior of water, so experts were consulted, see Section 2.5.

2.5 Case-study Evaluation

Verification. The original BOS project applied formal methods in the form of formal specification in Z to discover bugs in an early stage of the development. The specifications were used as a reference for implementation. The code is a composition of Z schemas translated into blocks of C++ annotated with the corresponding schema name. The implementation of each schema is based on the

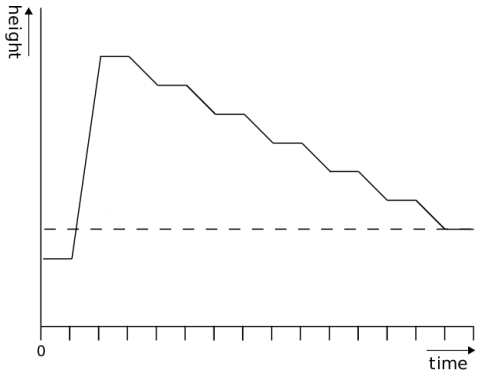


Figure 2.2: No non-decreasing excess.

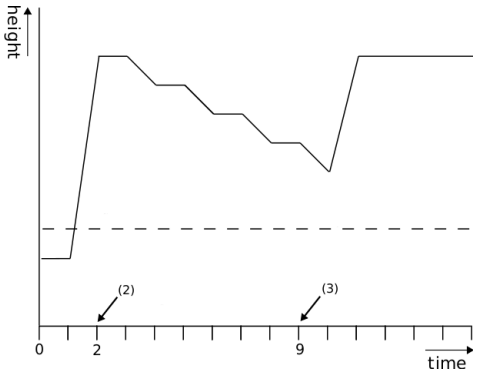


Figure 2.3: At $t = 2$ the barrier closes according to (2), at $t = 9$ it closes according to (3), the Z-spec.

programmer's own interpretations and design decisions. Hence there is no real formal connection between C++ implementation and Z specification. We have found three inconsistencies as a result of this. Two of them became apparent during the formalization in PVS, but a third was missed until real verification was applied. In the first two cases, the code seems to fix a flaw in the specification. Whether these fixes are a deliberate decision or just "luck" is unclear. The manual modeling of C++ code in PVS provided enough assurance about the correspondence between source code and model for this particular case-study. In Section 2.7 we discuss a complementary approach that may be viable for future verification projects on BOS or other safety-critical software.

Validation. We have validated the specification of DEW itself by formulating challenge theorems that focus on a particular high-level property. We considered two aspects: safety and the condition under which DEW decides to close the barrier. Validation of the safety aspect showed that there are two situations in which DEW does not raise an alarm when information is missing (due to undetermined values or incomplete predictions). To validate the closing conditions, we studied the definition of critical excess by comparing it with plausible alternative definitions. In principle these definitions do not have to be complete or correct, as the resulting counterexamples are only used as a guidance to understand the differences between the definitions and for discussion with subject matter experts. Possibly, several iterations are required to synchronize with the domain experts in other projects. We believe that in this case-study our unprejudiced abstract understanding of the domain was an advantage in finding the issues with the specification.

Impact analysis. Based on our findings presented in a preliminary report, subject matter experts performed an impact analysis. They agreed that the issues we found are possible and that the system would exhibit undesirable behavior in those situations. However, the issues found are very unlikely to do harm in reality, because the delays caused are very small compared to the process of detecting a storm and preparing for a closure. So the findings have no impact on the performance of BOS required in practice.

2.6 Related Work

BOS was developed using what one could consider to be lightweight formal methods. It was specified in Z to discover ambiguities in an early stage of its development and parts of it were checked using the SPIN model checker [Hol97]. Experiences with the development of BOS using this approach are described in [WBRG08]. In [TWC01], Hall's seven myths of formal methods are revisited based on the experiences of the BOS project.

The interface between BOS and BESW (a separate system that operates the barrier) was studied initially by Kars [Kar96]. The interface was specified in Promela and model checking was applied using SPIN [Hol97], revealing several flaws. A redesigned and redefined version was developed by RWS and subsequently studied by Ruys [Ruy01] again using Promela and SPIN, applying several abstraction techniques to fight state space explosion. One serious timing error was found.

The emergency closing system of a different storm surge barrier, located in the Eastern Scheld, has been a case-study of the QUEST project in the late 1990ies, which was funded by the German BSI [Slo99]. This project focused on the combination of formal methods in the software development process. One of the aims was a formal verification of the correctness of the open- and close-signals of the barrier.

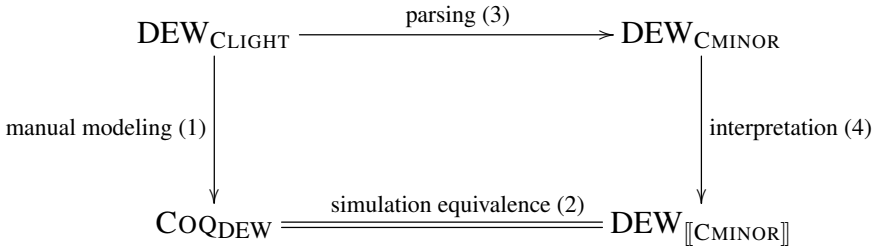
The approach of a lightweight modeling of the source language semantics way has been practiced before in [STT+09] to verify Fiasco's IPC implementation. John et al. have developed a tool that generates out of MISRA-C state transition models encoded in PVS. Due to the fact that PVS is strongly typed, possible run-time errors in the C program result in inconsistencies in the generated PVS specification.

2.7 Future Work: Certified Lightweight Semantics

Industrial safety-critical software, like the decision and support system (BOS) studied in this chapter, is usually developed according to "safe" coding guidelines, e.g. see [Hol06]. These coding rules attempt to increase the ability to more thoroughly check the reliability of such critical applications. In essence, the rules restrict the use of the complete source language to a subset of that language. The advantage of this subset is that its semantics if often much simpler than the full semantics of the complete language. For example, the DEW component of BOS is programmed in C++, but hardly uses any of the object oriented features available in C++. In fact, this made the simple direct translation to PVS (Section 2.3) feasible.

This PVS model, however, was created manually, and may therefore not completely match the original semantics of DEW. For our project the informal justification of correctness of this model was sufficient, but for future verification projects a stronger, preferably provable correspondence between the original code and the derived model might be desired. In the section we discuss how such a correspondence can be established.

Formalizing semantics of real programming languages is a very important trend in computer science. One of the greatest achievements in this area is a fully verified optimizing C compiler developed in the COMPCERT project [Ler09]. In this project a large subset of C, called CLIGHT, is compiled to PowerPC code. The correctness of this compiler, containing various complex optimization passes, has been proven in the theorem prover COQ. By ignoring the small differences between CLIGHT and the subset of C++ in which DEW is specified, we can consider DEW as a CLIGHT program. The semantics of CLIGHT is defined by means of an interpreter for CMINOR which is, in fact, a representation in COQ of the abstract syntax tree created by the CLIGHT parser. This framework can then be used to obtain what we brand as the "official" semantics of the DEW code. On the other hand, we can use the manually derived model presented in this chapter (but now specified in COQ rather than in PVS) as an alternative semantics, and show correctness of this model by proving in COQ that both semantics are equivalent. This proof actually boils down to constructing the simulation relation as depicted in the following diagram.



In general, to prove a property of a CLIGHT program, say P, there are two possibilities:

1. By manually constructing a model (step 1 in the diagram) for P, proving the desired property using this model, and showing the equivalence of the model (step 2) based on the official semantics of P (which is obtained via step 3 and 4 in the diagram).

2. By using the official semantics of P directly, i.e. both formulate and prove the desired property directly in $\text{DEW}_{[\![\text{CMINOR}]\!]}$

The main advantage of the first approach is that it provides a *lightweight shallow embedding*: The program is directly translated into COQ in such a way that only those aspects of P that are relevant to the desired property are take into account. Proving properties using such a lightweight abstract model is usually much easier than to use a full concrete model (a model incorporating all the aspects of the entire language). The disadvantage is, of course, that it requires the additional proof obligation in which correspondence between model and official semantics is stated.

Last but certainly not least, by compiling our program with COMPCERT, we obtain for free that resulting code exhibits exactly the same behavior as the model about which the safety properties have been proved: the verified compiler guarantees that properties proved on the (model of the) source code hold for the executable compiled code as well.

2.8 Conclusions

In this chapter we have verified and validated a crucial component of BOS, called DEW. This component is responsible for deciding when the storm surge barrier near Rotterdam should close. BOS was specified in the formal language Z, but no real formal verification was applied during its development. We have developed a formal model of the Z specification and the C++ code in the PVS theorem prover

by means of a manual lightweight modeling of the semantics. This enabled us to find an error in the code, but also two flaws in the Z specification which the code seems to fix. The challenge of verifying DEW was to make a sound model that isolates the relevant code.

Validation of the specification itself revealed deeper issues with the specified and implemented behavior of DEW. It does not raise an alarm when information is missing that is mandatory for the decision being made, which is questionable behavior in the context in which it is being used. Another issue is that the precise conditions under which DEW decides that the barrier must close seem to be incomplete. This last issue seemed quite serious. All issues were confirmed by domain experts.

In future work we plan to carry out the proposed approach to ensure (formal) correspondence between lightweight manual modeling and official semantics on other safety-critical software.

Acknowledgements. The author would like to thank Wouter Geurts from CGI Nederland B.V. for technical support during the project, and Erik Poll and the anonymous reviewers for their valuable comments on a draft version of the published paper.

CHAPTER 3

Reasoning About Assignments in Recursive Data Structures

Abstract. This chapter presents a framework to reason about the effects of assignments in recursive data structures. It defines an operational semantics for a core language based on Bertrand Meyer's ideas for a semantics for the object-oriented language EIFFEL [Mey03]. A series of field accesses, e.g. $f_1 \bullet f_2 \bullet \ldots \bullet f_n$, can be seen as a path on the heap. This chapter provides rules that describe how these multidot expressions are affected by an assignment. Using multidot expressions to construct an abstraction of a list, it proves the correctness of a list reversal algorithm. This approach does not require induction and the reasoning about the assignments is encapsulated in the mentioned rules. This chapter also discusses how to use this approach when working with other data structures and how it compares to the inductive approach. The framework, rules and examples have been developed and their correctness was proved in PVS.

3.1 Introduction

In order to verify pointer programs that manipulate recursive data structures, one generally identifies the pointer structure embedded in the heap with an abstract model. A concrete instance is a mapping of a set of objects on the heap connected by a field such as *next* to an abstract list of objects. The mapping is called

the *abstraction* and the abstract list is called the *abstract model*. An operation performed by the program on a pointer structure on the heap has a corresponding operation on the abstract model. For example, the operations performed by a list reversal algorithm have the combined effect that the abstract list is reversed at the end of the execution. The standard way to define data abstractions is by recursion on the structure of (the data type of) the abstract model.

Verification of pointer programs is a non-trivial task due to the possibility of aliasing. Modifying data through one name implicitly modifies the values associated to all aliased names. If two portions of the heap are disjoint, an assignment in one part of the heap does not affect the other; this is called *local reasoning*. Local reasoning is essential for scalability and several approaches to obtain it have been studied, see e.g. Separation Logic [Rey02] and Region Logic [RBN10].

When it is not known how the heap is partitioned or when working within a region that may contain aliases, we have to reason about how a change to (a portion of) the heap affects the corresponding abstract model. This complements local reasoning. In this chapter we focus on the effects of assignments to abstract models. We present our work in the setting of a core language, inspired by Meyer's ideas for a semantics for the object-oriented language EIFFEL [Mey03].

Our framework allows us to express multidot field access expressions, multidot expressions for short, of the form $f_1 \cdot f_2 \cdot \ldots \cdot f_n$. A multidot expression consisting of a series of *next*-fields describes a path from the head of a list to one of its elements. If we instantiate it with a series of *left* and *right*-fields we can describe the path from the root of a binary tree to any node or leaf. In general, a multidot expression describes a path on the heap where the elements are connected by field accesses.

The main contribution of this chapter is to provide a set of rules that precisely describe the value of a multidot expression after an assignment, and to show how these rules can be applied for verification of programs that manipulate recursive data structures. The given rules are categorised into separation rules, where the assignment has no effect on the multidot expression, and interference rules, where the assignment does have effect on the multidot expression. We have applied these rules to show the correctness of an in-place list reversal algorithm by mapping each element of the list to a multidot expression. We also discuss how to apply the same principles to other recursive data structures and we make a comparison with the standard inductive approach. Our work has been completely carried out in the theorem prover PVS [OSRS01].[1]

This chapter is organised as follows. Section 3.2 gives a short introduction to PVS and introduces the notation. Section 3.3 defines the language we shall work with. In Section 3.4 we present the rules that describe the effects an assignment

[1]The PVS files can be obtained at `http://www.cs.ru.nl/~kmadlene/yaspcc.zip`.

can have on a multidot expression and in Section 3.5 we apply these rules to prove the correctness of a list reversal algorithm and we discuss the applicability to other data structures. We compare the approach described in this chapter with the standard inductive approach and we give pointers for future work in Section 3.6. Related work is discussed in Section 3.7 and conclusions are drawn in Section 3.8.

3.2 Preliminaries

PVS is based on higher-order logic with dependent types and predicate subtyping. Subtyping based on predicates makes type checking undecidable, so when PVS cannot infer the desired type itself, it will generate a proof obligation. Its intuitive syntax is reminiscent of functional languages such as HASKELL. For reasons of presentation, we slightly simplify the actual PVS syntax. We will briefly introduce the notation used in this chapter.

Formulas are terms of type *bool*. We shall use the standard notation for connectives ($\wedge, \vee, \Rightarrow, \neg$), and for quantifiers ($\forall, \exists$). There is a conditional term **IF** ϕ **THEN** M **ELSE** N, for terms M, N of the same type.

Given the types $\sigma, \tau, \sigma_1, \ldots, \sigma_n$, function types are written as $[\sigma \to \tau]$ and record types as $[lab_1 : \sigma_1, ..., lab_n : \sigma_n]$. Given the record types $\rho_1, \ldots, \rho_m$, labelled coproduct types are written as $\{lab_1\ (\rho_1), ..., lab_m\ (\rho_m)\}$. Terms of coproduct type can be constructed with $lab_i\ (M)$, where $M : \rho_i$, and recognised with $lab_i?$. Standard set comprehension notation can be used to define predicate subtypes. New types can be introduced via definitions, like

$$lift\ [\sigma] : \mathbf{TYPE} = \{bottom, up\ (down : \sigma)\}$$

(*bottom* is the unit type and we omit its argument). The *lift* type constructor adds a bottom element to an arbitrary type σ given as a parameter, written in short as $\sigma_\perp\ :=\ lift\ [\sigma]$.

Lists are defined as

$$list\ [\sigma] : \mathbf{TYPE} = \{null, cons\ (car : \sigma, cdr : list)\}$$

There is an infix function $+\!\!+$ that appends two lists. It is overloaded so that when one of its arguments is of type σ, then this argument is converted to a list. The ith element of a list l can be accessed using $nth\ (l, i)$.

3.3 The Model

This section describes an operational semantics of a core object-oriented language. The focus is on the features needed to understand the properties discussed

in the next section, i.e., we do not model some typical object-oriented features like inheritance.

3.3.1 The heap

In our model we consider all values to be an object or void. The set *Object* is defined as an uninterpreted type that represents non-void objects. Instances of $Object_v$ have the possibility of being *void*:

$$Object_v : \mathbf{TYPE} = \{obj\ (obj : Object), void\}$$

A basic approach to model the heap, due to Burstall [Bur72] and more recently emphasised by Bornat [Bor00], is to model it as a collection of functions of type $Object \to Object_v$, one for each class field (i.e. the component). This modelling encodes the fact that changing what object a field points to does not affect other fields. This has the important consequence that whenever one field is updated, we do not need to propagate that update to the other fields. This is sometimes called the component-as-array model [FM07, HM07].

Our heap is a grouping of field functions, indexed by their field names:

$$Heap : \mathbf{TYPE} = [Name \to [Object \to Object_v]]$$

where *Name* is a set representing the field names. Given a heap h and a field name f, $h\ (f)$ is the corresponding field function. This indexing allows us to reason about field names, which is not possible when using a loose set of field functions as in the component-as-array model. There, the names of the fields are fixed by the names of the functions that model them. We use this to express meta-properties about multidot field expressions in Section 3.4. The separation by syntax provided by the component-as-array model is lost in this model, because a field update is now an update on the heap function. With the meta-level properties presented in the rest of this chapter, we obtain a reincarnation of separation by syntax.

The above definition of the heap highlights the relationship with the component-as-array model. However, defining the heap as a function of type $[Object \to [Name \to Object_v]]$ may seem more intuitive. In this definition we first fix an object and then we ask for a field name to obtain its value. As the functions are total (required by PVS), both definitions are in fact equivalent. This means that every field should be defined at every object. This is of course not realistic, however, accesses to undefined fields can be handled by a preliminary static analysis.

3.3.2 Expressions, statements and compositions

We model expressions, statements and their compositions following Meyer's ideas for a semantics for EIFFEL [Mey03]. A distinctive aspect of this approach is that

expressions and statements are evaluated relative to an object, which is provided together with the heap as argument.

We deal with null-pointer dereferencing in language constructs, as opposed to avoiding it by type constraints. In our experience, the second approach leads to cumbersome specifications because the result of each expression and statement must be checked for definedness before composing them.

There are two syntactic categories: expressions (without side-effects) and statements:

$$\begin{aligned}
&Expr : \mathbf{TYPE} = \{ e : [Object_{v\perp}, Heap_{\perp} \to Object_{v\perp}] \mid \\
&\quad \forall (o : Object_{v\perp}, h : Heap_{\perp}) : \\
&\qquad bottom_or_void\,?\,(o,h) \;\Rightarrow\; bottom\,?\,(e\,(o,h)) \} \\
&Stmt : \mathbf{TYPE} = \{ S : [Object_{v\perp}, Heap_{\perp} \to Heap_{\perp}] \mid \\
&\quad \forall (o : Object_{v\perp}, h : Heap_{\perp}) : \\
&\qquad bottom_or_void\,?\,(o,h) \;\Rightarrow\; bottom\,?\,(S\,(o,h)) \}
\end{aligned}$$

To define a semantics for EIFFEL, Meyer works with partial functions [Mey03]. In most theorem provers, including PVS, functions have to be total. For this reason we use lifted arguments, to represent undefinedness. The $bottom\_or\_void\,?\,(o,h)$ predicate returns *true* if and only if o is undefined or void or h is undefined. By using predicate subtypes, we ensure that whenever an expression or statement is evaluated in *void* or in an undefined object or state, the result is undefined. This shifts checking for void or bottom from the specification to type correctness obligations that PVS generates automatically.

The expression *Current* (called *this* or *self* in some languages) returns the current object:

$$\begin{aligned}
&Current : Expr = \lambda\,(o : Object_{v\perp}, h : Heap_{\perp}) : \\
&\quad \mathbf{IF}\ bottom_or_void\,?\,(o,h)\ \mathbf{THEN}\ bottom\ \mathbf{ELSE}\ o
\end{aligned}$$

The operators $\bullet$ and ; compose expressions and statements. If x is an expression, S an statement and r is either of them, we define:

$$\begin{aligned}
&S;r = \lambda\,(o : Object_{v\perp}, h : Heap_{\perp}) : r\,(o, S\,(o,h)) \\
&x \bullet r = \lambda\,(o : Object_{v\perp}, h : Heap_{\perp}) : r\,(x\,(o,h), h)
\end{aligned}$$

The normal uses are state compositions $S;T$ and field access $x \bullet y$. The overloading allows us also to write $S;x$, which returns the value of evaluating x after the statement S, and $x \bullet S$, which can be thought as a qualified call of S from x.

We define in PVS an automatic conversion that translates a name f into the expression $\lambda\,(o : Object_{v\perp}, h : Heap_{\perp}) : h\,(f)\,(o)$ whenever needed. This allows us to express a field access directly as $x \bullet f$. We also define a conversion that

translates an $o : Object$ into $obj\ (o) : Object_v$, and one that translates an $o : Object_v$ into $up\ (o) : Object_{v\perp}$, to reduce the amount of syntax.

IF-statements are mapped to **IF**-expressions in the logic of PVS. **WHILE**-statements can be modeled as recursive PVS functions, provided that a measure can be given (which is required by PVS to ensure that the function is total). This chapter does not treat **WHILE**-statements in depth; see the LOOP project [JVDBH+98] for the PVS semantics of the JAVA version of **WHILE**.

3.3.3 Assignments

At its core, an assignment is an update of a heap-function in a particular point (consisting of a field and an object):

$$
\begin{array}{l}
update\ (f : Name, p : Object, h : Heap, q : Object_v) : State = \\
\quad \lambda\ (g : Name)\ (o : Object) : \\
\qquad \textbf{IF}\ p = o\ \wedge f = g\ \textbf{THEN}\ q\ \textbf{ELSE}\ h\ (g)\ (o)
\end{array}
$$

Our model forces us to explicitly deal with undefinedness due to dereferencing of void. The *update* operation is encapsulated in an operator := that assigns an object q to the field f of the object p in the heap h:[2]

$$
\begin{array}{l}
:= (f : Name, q : Object_v) : Stmt = \\
\quad \lambda\ (p : Object_{v\perp}, h : Heap_\perp) : \\
\qquad \textbf{IF}\ bottom\_or\_void\,?\,(p)\ \vee\ bottom\,?\,(h)\ \textbf{THEN}\ bottom \\
\qquad \textbf{ELSE}\ update\ (f, obj\ (down\ (p)), down\ (h), q)
\end{array}
$$

If the assignment is made in an undefined state or tries to assign to void, the error is propagated. Note that this always causes the entire program to terminate, as our language (as presented in this chapter) does not support exception handling. This is required by the definition of *Stmt*. We shall use the above variable names throughout the rest of this chapter. The next step is to define local assignments $f := e$ and qualified assignments $e_1.f := e_2$. These definitions are not relevant for the development of this chapter and we therefore omit them.

An assignment affects a field access if and only if the object where the field is evaluated is the one where the assignment was made and the field being accessed is the one that was assigned to. This is summarised in the following two basic separation and interference properties (both assume that o, h is not bottom or void):

Property 1. *If $p \neq o$ or $f \neq g$, then $g\ (o, (f := q)\ (p, h)) = g\ (o, h)$.*

[2] In the PVS formalisation we have called this function $\Leftarrow$, because := is reserved.

Property 2. *If $p = o$ and $f = g$, then $g\ (o, (f := q)\ (p,h)) = q$.*

The proofs of these two properties amount to expanding the definition of := and applying several case-splits. When the assignment is replaced with a qualified assignment $e_1 \bullet f := e_2$, then analogous properties hold, but $p = o$ is replaced by $e_1\ (o,h) = o$.

One has to explicitly apply properties 1 and 2 as proof steps to reason about the effect of an assignment in the presented semantics. The key condition is $p = o \wedge f = g$. The latter is a syntactical comparison and thus can be done automatically. However, most of the time comparison between objects cannot be discharged automatically, unless we have information about the layout of the heap, see Section 3.6.

3.4 The Effect of Assignments on Multidot Expressions

In this section we look at expressions of the form

$$(g_1 \bullet \ldots \bullet g_n)\ (o, (f := q)\ (p,h)), \tag{3.1}$$

where the g_i and f are field names, o and p are $Object_{v\perp}$ and q is of type $Object_v$. Because undefinedness due to dereferencing void is not an essential part of the discussion, we shall omit it in the rest of this chapter.

Properties 1 and 2 describe the result of a very simple multidot, namely one where n is equal to 1. There, $p = o \wedge f = g$ is the condition which determines the result. In multidot field expressions of arbitrary length a similar condition determines the result, but now it must be taken into account that in the path from o to

$$(g_1 \bullet \ldots \bullet g_n)\ (o, (f := q)\ (p,h)),$$

the field f of the object p can be traversed more than once if there is a loop. Thus we are interested in the set of indexes k such that:

$$p = (g_1 \bullet \ldots \bullet g_{k-1})\ (o,h) \text{ and } f = g_k\,.$$

The properties we present in this section are categorised into *separation rules*, where the assignment has no effect on the multidot field expression, and *interference rules*, where the assignment does have effect on the multidot field expression. Moreover, we now have a choice to look at the heap h before the assignment, or at the heap $h' = (f := q)\ (p,h)$ after the assignment. For the separation properties this does not make a difference, but for the interference properties it does.

The properties we derive about multidot expressions in this section are at the meta-level. Although it is possible to use them to reason about a particular multidot in a program, the intended use is to reason about the effects of assignments on recursive data structures. Examples that demonstrate the application are given in Section 3.5.

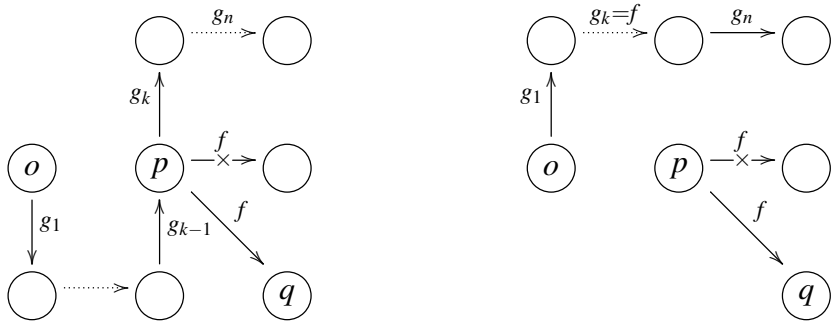
To improve readability, the notation for multidot expressions differs from the actual syntax used in PVS. In the last subsection we show the concrete PVS formalisation of a property. We will use graphs representing a portion of the heap h' to show examples of the properties. In these graphs nodes are objects and edges are labelled with an attribute name. An edge $\xrightarrow{f}$ from an object o to an object p means that $f\,(o,h') = p$. The edge removed by the heap update is depicted as $\xrightarrow{f}\!\!\!\!\times$.

3.4.1 Looking at the heap before the assignment

The assignment in (3.1) may or may not modify the multidot field expression. Graphically, what matters is whether the edge that has changed belongs to path followed by the multidot field expression or not. A particular edge is determined by its object of origin and the field name. Hence, the condition that determines whether the assignment influences the multidot expression is whether or not the following set is empty:

$$K_{pre} := \{k : nat \mid k < n \wedge p = (g_1 \bullet \ldots \bullet g_{k-1})\,(o,h) \wedge f = g_k\}.$$

We start with the case where K_{pre} is empty, i.e., the edge changed by the assignment is not part of the multidot expression, as shown in Figure 3.1.



(a) $p = (g_1 \bullet \ldots \bullet g_{k-1})\,(o,h)$, but $f \neq g_k$. (b) $f = g_k$, but $p \neq (g_1 \bullet \ldots \bullet g_{k-1})\,(o,h)$.

Figure 3.1: Examples where K_{pre} is empty.

As the edge that changed was not part of the multidot expression, the assignment does have an effect on it.

Property 3. *(empty_K_{pre}) If K_{pre} is empty, then*

$$(g_1 \bullet \ldots \bullet g_n)\ (o,h') = (g_1 \bullet \ldots \bullet g_n)\ (o,h)\,.$$

Now consider the case where K_{pre} is not empty. Figure 3.2 depicts an example with two indexes i and k in K_{pre} such that $k<i$. If there are several indexes in K_{pre},

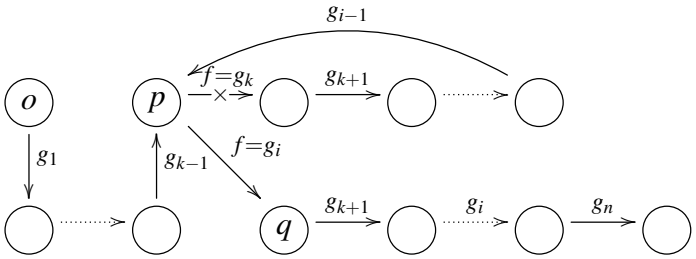


Figure 3.2: Example with two indexes $k<i$ in K_{pre}.

it means that there are several loops starting at p. The assignment breaks the first edge in these loops. In the heap after the assignment, the edge that joins p with q is determined by the *least* element in K_{pre}.

Property 4. *(min_K_{pre}) If $k = min\ (K_{pre})$, then*

$$(g_1 \bullet \ldots \bullet g_n)\ (o,h') = (g_{k+1} \bullet \ldots \bullet g_n)\ (q,h')\,.$$

Since the assignment may also affect the path that goes from q to the final value, the right hand side must still be evaluated in h'.

3.4.2 Looking at the heap after the assignment

Instead of looking at when the multidot expression follows the edge that changed in h, we will now look at when it follows the new edge in h'. That is, we will look at the set:

$$K_{pos} := \{k : nat \mid k<n \wedge p = (g_1 \bullet \ldots \bullet g_{k-1})\ (o,h') \wedge f = g_k\}.$$

If the new edge is never traversed, the multidot expression does not change.

Property 5. *(empty_K_{pos}) If K_{pos} is empty, then*

$$(g_1 \bullet \ldots \bullet g_n)\ (o,h') = (g_1 \bullet \ldots \bullet g_n)\ (o,h)\,.$$

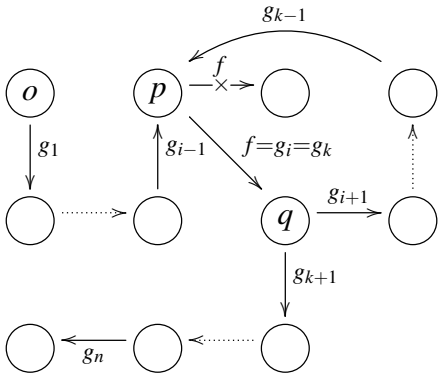


Figure 3.3: Example with two indexes $i < k$ in K_{pos}.

Now assume that there is at least one index in K_{pos}. In Figure 3.3 we see an example with two such indexes i and k with $i < k$. In this case the result of the multidot expression can be described as either $(g_{k+1} \bullet \ldots \bullet g_n)\,(q,h')$ or as $(g_{i+1} \bullet \ldots \bullet g_k \bullet g_{k+1} \bullet \ldots \bullet g_n)\,(q,h')$. If we take the greatest index in K_{pos}, we get the shortest path to the resulting value and since the rest of the edges are not affected by the assignment we can describe the result in terms of h. This is expressed in the following properties.

Property 6. *($forall\_K_{pos}$) For all k in K_{pos},*

$$(g_1 \bullet \ldots \bullet g_n)\,(o,h') = (g_{k+1} \bullet \ldots \bullet g_n)\,(q,h')\,.$$

Property 7. *(max_K_{pos}) If $k = max\,(K_{pos})$, then*

$$(g_1 \bullet \ldots \bullet g_n)\,(o,h') = (g_{k+1} \bullet \ldots \bullet g_n)\,(q,h)\,.$$

3.4.3 PVS formalisation

Given a list of names *fs*, the dot composition of the corresponding attributes is formalised as

multidot (*fs* : *list* [*Name*]) : **RECURSIVE** *Expr* =
  IF *null*?(*fs*) **THEN** *Current*
  ELSIF *null*?(*cdr* (*fs*)) **THEN** *car* (*fs*)
  ELSE *car* (*fs*)•*multidot* (*cdr* (*fs*))
MEASURE *length* (*fs*)

Note that because $e \bullet Current = e$ does not hold when e evaluates to void, we cannot simply append *Current* at the end of the multidot expression.

As an example of the PVS formalisation, we show a property that combines $empty\_K_{pos}$ and max_K_{pos} in a property at the source code level. Since it is written as an equality between functions, it can be used as a rewrite rule.

$$
\begin{array}{l}
multidot\_after\_assignment\_pos : \textbf{LEMMA} \\
\quad \forall\, (f : Name, gs : list\,[Name], x, e : Expr, o : Object_{v\perp}, h : Heap_{\perp}) : \\
\quad\quad (x.f := e; multidot\,(gs))\,(o,h) = \\
\quad\quad\quad \textbf{LET}\ h' = (x.f := e)\,(o,h), \\
\quad\quad\quad\quad K_{pos} = \lambda\,(k : below\,[length\,(hs)]) : \\
\quad\quad\quad\quad\quad\quad x\,(o,h) = multidot\,(take\,(gs,k))\,(o,h') \wedge \\
\quad\quad\quad\quad\quad\quad f = nth\,(gs,k)\ \textbf{IN} \\
\quad\quad\quad \textbf{IF}\ bottom\,?\,(h')\ \textbf{THEN}\ bottom \\
\quad\quad\quad \textbf{ELSIF}\ empty\,?\,(K_{pos})\ \textbf{THEN}\ multidot\,(gs)\,(o,h) \\
\quad\quad\quad \textbf{ELSE LET}\ k = max\,(K_{pos})\ \textbf{IN} \\
\quad\quad\quad\quad \textbf{IF}\ k = length\,(gs) - 1\ \textbf{THEN}\ e\,(o,h) \\
\quad\quad\quad\quad \textbf{ELSE}\ (e.multidot\,(drop\,(gs,k+1)))\,(o,h)
\end{array}
$$

This property describes in terms of h all the possible outcomes of $multidot\,(gs)$ when evaluated in h'. If the assignment resulted in an error then the result is an error. If K_{pos} is empty then the multidot expression is unchanged. Otherwise, let k be the greatest element in K_{pos}. The result is then as stated in max_K_{pos} (with a shift of indexes due to lists starting at 0 in PVS). But again because $e.Current$ is not equal to e when evaluated on void, we have to make a special case for when the multidot expression ends exactly at e. There is a similar lemma that combines $empty\_K_{pre}$ and min_K_{pre}.

The intuitive way to prove these properties is by induction on gs. The intention is to reason about the last edge of the multidot expression and use the inductive hypothesis on the path that leads to it. The problem with this approach is that on the non-empty case we have to reason about a list of the form $cons\,(g,gs)$. Therefore, we get to reason about the first edge, not the last one. To overcome this problem we defined a function $multidot\_rev$ that chains the arguments in the reverse order. Then we wrote lemmas that are adapted to work with the reversed list, and we proved them by induction on gs. Finally, the original lemmas were proven using their reversed counterpart by instantiating gs with $reverse\,(gs)$.

3.5 Linearised Abstractions

In this section we look at examples of abstract models expressed in terms of multidot field expressions. We call this style of specifying *linearised*, because it is not by recursion on the structure of the abstract model. The properties derived

in the previous section provide us a set of tools to reason about the effects of an assignment to a linearised abstraction.

3.5.1 Paths

The following definition abstracts a path embedded in the heap to a list l of *Object*s. The ith object in l is the object on the heap that can be accessed by requesting the first i fields describing the path.

$$\begin{array}{l} Path\ (gs : list\ [Name], l : list\ [Object]) \\ \quad (o : Object_{v\perp}, h : Heap_{\perp}) : bool = \\ \quad length\ (gs) + 1 = length\ (l) \wedge \\ \quad \forall\ (i : below\ [length\ (l)]) : \\ \quad\quad multidot\ (take\ (gs, i))\ (o, h) = nth\ (l, i) \end{array}$$

Due to the possibility of undefinedness, we define the abstractions as predicates about the heap and the current object rather than as functions because in PVS functions must be total.

With the use of the spatial separation lemmas for multidot expressions we can prove the following separation lemma for paths (recall that $h' = (f := q)\ (p,h)$):

Property 8. *If for all $i < length\ (l)$ it holds that $p \neq nth\ (l,i)$ or $f \neq nth\ (gs,i)$, and $\neg bottom?\ (f\ (p,h))$, then*

$$Path\ (gs,l)\ (o,h') = Path\ (gs,l)\ (o,h).$$

Thinking again in terms of graphs, this lemma says that if an edge outside the path is modified, then the path is not affected by the assignment. To give an idea of how the multidot rules are applied, we sketch the proof of this lemma.

Proof sketch. We are supposed to show that the *Path* predicates are logically equivalent. In expanded form, we have to show that the following predicates are equivalent:

$$\forall\ (i_1 : below\ [length\ (l)]) : (g_1 \bullet \ldots \bullet g_{i_1})\ (o,h') = nth\ (l, i_1) \tag{3.2}$$

$$\forall\ (i_2 : below\ [length\ (l)]) : (g_1 \bullet \ldots \bullet g_{i_2})\ (o,h) = nth\ (l, i_2) \tag{3.3}$$

To show that (3.2) implies (3.3), we instantiate i_1 with i_2 and apply $empty\_K_{pos}$. Then we have to show that K_{pos} is indeed empty. If this was not the case then there would be a k such that

$$p = (g_1 \bullet \ldots \bullet g_{i_k})\ (o,h') = nth\ (l,k) \text{ and } f = g_k,$$

which is a contradicts the assumption that p is not in l. For the converse direction, we apply $empty\_K_{pre}$ in an analogous way. □

The interference property for paths describes how a path ending in p can be joined with a path beginning at q:

Property 9. *If* $p \notin l_0 +\!\!+ q +\!\!+ l_1$ *and* $c = car\ (l_0 +\!\!+ p)$*, then*

$$Path\ (gs_1 +\!\!+ f +\!\!+ gs_2, l_0 +\!\!+ p +\!\!+ q +\!\!+ l_1)\ (c, h') =$$
$$(Path\ (gs_1, l_0 +\!\!+ p)\ (c,h) \wedge Path\ (gs_2, q +\!\!+ l_1)\ (q,h))$$

The proof uses the multidot rules $empty\_K_{pos}$ and max_K_{pos} for the implication from left to right and it uses the rules $empty\_K_{pre}$ and min_K_{pre} from right to left.

An important point about the proofs using linearised abstractions is that the induction is encapsulated in the rules about multidot expressions; to prove the above properties, we did not apply induction.

3.5.2 Example: verification of an in-place list reversal algorithm

The *Path* abstraction can be specialised by $Path\ (g,l)$, which instantiates the regular *Path* with a list of g-fields. By requiring the last node of $Path\ (next, l)$ to point to void, we obtain an abstraction for lists on the heap:

$List\ (l : list\ [Object])$
$\quad (o : Object_{v\perp}, h : Heap_{\perp}) : bool =$
$\quad Path\ (next, l)\ (o,h) \wedge$
$\quad$ **IF** $cons\,?\,(l)$ **THEN** $void\,?\,(next\ (last\ (l), h))$
$\quad$ **ELSE** $void\,?\,(o)$

Note that $List\ (null)\ (o,h)$ is true iff $void\,?\,(o)$ is true, i.e. an empty list is represented by void. Similar separation and interference properties as the ones for *Path* can be proved for *List*.

To prove the correctness of the annotated in-place list reversal algorithm listed in Figure 3.4, we use standard Hoare-style reasoning. The annotations have type $Asrt : [Object_{v\perp}, Heap_{\perp} \to bool]$ and a Hoare-triple has the following meaning for $P, Q : Asrt$ and $S : Stmt$:

$$\{P\}\ S\ \{Q\} := \begin{array}{l} \forall\, (o : Object_{v\perp}, h : Heap_{\perp}): \\ \quad P\ (o,h) \Rightarrow Q\ (o, S\ (o,h)) \end{array}$$

As can be seen in Figure 3.4, the current object o and the updated heap $S\ (o,h)$ distribute over the connectives. So, the actual work to verify the correctness

of the list reversal algorithm amounts to simplifying expressions of the form $(g.List\ (l))\ (o,(e_1.f := e_2)\ (o,h))$. By expanding the definitions of dot and assignment, this can be brought into the form of $List\ (l)\ (o',(f := q)\ (p,h'))$, on which the separation and interference rules for the *List* abstraction can be applied.

$\{\lambda\ (o,h) : \neg bottom\_or\_void\ ?\ (o,h)\ \wedge\ a.List\ (As)\ (o,h)\}$
$b := void;$
WHILE $(\lambda\ (o,h) : \neg void\ ?\ (a\ (o,h)))$ **DO**
$\{\lambda\ (o,h) : \neg bottom\_or\_void\ ?\ (o,h)\ \wedge$
$\quad \exists\ (as, bs : list\ [Object]) :$
$\quad\quad (a.List\ (as))\ (o,h)\ \wedge\ (b.List\ (bs))\ (o,h)\ \wedge$
$\quad\quad disjoint\ ?\ (as,bs)\ \wedge\ append\ (reverse\ (as), bs) = reverse\ (As)\}$
$\quad tmp := a;$
$\quad a := a.next;$
$\quad tmp.next := b;$
$\quad b := tmp;$
OD
$\{\lambda\ (o,h) : \neg bottom\_or\_void\ ?\ (o,h)\ \wedge\ (b.List\ (reverse\ (As)))\ (o,h)\}$

Figure 3.4: In-place list reversal.

3.5.3 Other data structures

The linearised specification approach exemplified in the previous two sections can also be applied to other recursive data structures. Consider for example binary trees that store a value in each node:

$binary\_tree\ [\sigma] : \mathbf{TYPE} = \{leaf, node\ (v : \sigma\ l, r : binary\_tree)\}$

It is straightforward to define a predicate

$get\_node\ (bt : binary\_tree\ [\sigma], path : list\ [Name], v : \sigma) : bool$

that says whether by traversing *bt* in the order specified by *path*, we arrive at *v*. Basically *get_node* maps each constructor application to the corresponding field name. We can now describe a binary tree on the heap by mapping each of its nodes to a multidot field access:

$binary\_tree\_abstraction\ (bt : binary\_tree\ [Object])$
$\qquad\qquad (o : Object_{v\perp}, h : Heap_{\perp}) : bool =$

$$\forall\,(x : Object, path : list\,[Name]) :\\ get\_node\,(bt, path, x)\ \Rightarrow\\ multidot\,(path)\,(o, h) = x$$

From the properties about multidot expressions presented in Section 3.4 one can obtain separation and interference lemmas for binary trees.

The same ideas can be applied to other tree-like structures. First make a linearised abstraction of the data structure: obtain the path from the root to each of its elements and use that path to describe the pointer structure in terms of multidot expressions. Then use the properties of Section 3.4 when reasoning about assignments. Data structures with loops can also be specified, e.g., a circular list is just a path that starts and ends in the same object.

3.6 Evaluation and Future Work

A natural way to define abstractions is by means of recursion on the structure of the abstract model. We single out the work by Mehta and Nipkow that uses this approach to verify several pointer programs [MN05]. The advantage of using induction is that it is a familiar general-purpose method that is integrated in the theorem prover. Much work has been devoted to automate proofs by induction, in particular to heuristics to instantiate the inductive hypothesis, e.g. *rippling* [BBHI05]. In the inductive approach one still has to reason about the effect of the assignments to the data structure, whereas using the rules given in Section 3.4 the focus is on when to apply each rule and in finding the extrema of the K-sets, which requires an instantiation.

Our experience is that both approaches require a comparable amount of proof work. However, there is still work to be done on investigating specialised version of the assignment rules and on the integration with the theorem prover as tactics. For example, if we know that there is no loop on a multidot expression, as is the case in tree-like structures, then we also know that the K-sets are either empty or have only one element. This eliminates the need to find the minima or the maxima.

Because both approaches lead to definitions that are essentially equivalent, the same properties hold. Hence, our approach can be seen as a complement rather than a replacement of inductive reasoning.

Reasoning about assignments ultimately reduces to reasoning about object equality. Therefore, this framework would benefit from knowledge about the layout of the memory. The separation rules are used to provide local reasoning, but they are not a primitive of the logic as the star conjunct is in Separation Logic [Rey02] (see also Section 3.7). Hubert and Marché [HM07] propose a static separation analysis and show how it can be integrated in the component-as-array

modelling. They split the heap into regions that are inferred by the separation analysis and accordingly relabel the field names as a combination f_r of the old field name f and a region r. This could be integrated into our model, for example by redefining the heap as

$$Heap : \mathbf{TYPE} = [Region, Name \to [Object \to Object_v]]$$

When it is inferred that two objects x and y lie in separate regions, the comparison between them can be avoided and the separation lemmas can be applied automatically.

3.7 Related Work

A first version of some of the rules presented in Section 3.4 first appeared in Tamalet's Master's thesis [Tam06].

In the seminal work of Bornat [Bor00] and also in the work by Meyer on a semantics for EIFFEL [Mey03], pointer structures on the heap are related with abstract models via repeated composition of field requests. This has been a source of inspiration for this chapter. Bornat and Meyer both define a sequence closure operator that repeatedly requests a series of (the same) fields, yielding the list of objects that is traversed on the heap. This is essentially the same as our *Path* abstraction of Section 3.5.2. In this chapter we have given a complete and formalised overview of the effects of assignments to arbitrary multidot field expressions. A treatment of the sequential operator in the context of EIFFEL has been given in an unpublished work by Blanco and Castro [BP05], restricted to the case of lists.

A perhaps more natural way to define abstractions is by the use of recursion on the structure of the abstract model. Mehta and Nipkow [MN05] used this approach to verify the correctness of several pointer programs. We have compared the inductive approach and the linearised approach in Section 3.6.

Hoare and Jifeng [HJ99] introduce a framework for the formulation of assertions about objects and pointers based on trace model of graphs and process algebra. They use a graphical notation very similar to the one used in this chapter. However, their model uses graph transformations to describe the changes to the state whereas we use an operational semantics.

Our rules about an assignment followed by a multidot are meta-level properties of the language. To enable this meta-level reasoning we introduced a function *multidot* that maps a list of *Name*s to a suitable expression, which is essentially a deep embedding of multidot expressions. The rules about multidot expressions are a reflection of the properties 1 and 2. For an instructive paper on reflection with examples in PVS we refer to [vHPPR98].

3.7.1 Local reasoning

Local reasoning is the key to scalability in formal verification of programs. The way the heap is modelled in our framework is based on the component-as-array modelling idea of Burstall [Bur72]. Refinements of this modelling have been used as the core of weakest pre-condition calculus-based tools such as Krakatoa for the verification of Java programs, and Caduceus for the verification of C programs [FM07, MPM05]. A separation analysis tailored to integration with the component-as-array modelling has been proposed by Hubert and Marché [HM07]. Future work on the integration of this analysis with our work has been discussed in Section 3.6.

A well-studied approach to obtain local reasoning is that of Separation Logic, proposed by Reynolds [Rey02], which can be seen as a radical refinement of Burstall's idea. In Separation Logic disjointness of portions of the heap is made explicit in the logic. Its frame rule allows one to reason about just the relevant portion of the heap that a piece of code manipulates and later augment it with the rest of the heap. The separation logic assertion checking tools VERIFAST [JP08] and SMALLFOOT [BCO06] have a (partially) verified formal theory, see respectively [VJP12] and [Tue09].

A related line of research is Region Logic, whose goal it is to preserve the local reasoning of Separation Logic, but without using non-standard semantics of Hoare-triples. See [RBN10] for recent work.

3.8 Conclusions

In this chapter we have presented a novel approach to reason about assignments in recursive data structures. We have shown how recursive pointer structures can be described in terms of paths obtained by a series of field accesses. We have provided a formal model of these paths as multidot expressions and we have proved a set of rules that describe how an assignment can affect them. Using these rules we have derived separation and interference lemmas for lists and verified an in-place list reversal algorithm. A complete formalisation of the presented work has been carried out in the PVS theorem prover. We have also shown how to apply this approach to reason about other data structures and we have compared it with the standard inductive approach.

Acknowledgements. The author would like to thank Marko van Eekelen and Sjaak Smetsers for their comments on a draft version of the published paper.

CHAPTER 4

Formal Component-Based Semantics

Abstract. Component-Based Semantics is a solution proposed by Mosses [Mos09] aimed at improving the scalability of semantics of programming languages. It is expected that this framework can also be used effectively for modular meta-theoretic reasoning. This chapter presents a formalization of Component-Based Semantics in the theorem prover COQ. It is based on Modular SOS, a variant of SOS, and makes essential use of dependent types, while profiting from type-classes. This formalization constitutes a contribution towards modular meta-theoretic formalizations in theorem provers. As a small example, a modular proof of determinism of a mini-language is developed.

4.1 Introduction

Theorem prover formalization of programming language meta-theory and semantics receives a lot of attention. Most notably, the POPLMARK Challenge [ABF+05] calls for experiments on verifications of meta-theory and semantics using proof tools. One of the main issues that programming language formalizations have to cope with is the lack of reusability of existing work. Many programming languages have language constructs in common, but often have (slight) differences in their precise semantics (e.g. assignments in C versus assignments in JAVA).

Component-Based Semantics, introduced by Peter D. Mosses, aims to resolve this reusability issue by constructing language descriptions from combinations of

basic abstract constructs [Mos09]. Basic constructs are supposed to have a fixed meaning and be language-independent. As an example, the basic construct of conditional expressions should not depend on whether the expressions may have side-effects or not, terminate abruptly or even interact with other processes. One could even go as far as creating a repository of constructs that may be freely combined to build new languages. This repository is therefore necessarily open-ended, enabling users to add newly discovered basic constructs.

Modular Structural Operational Semantics (MSOS) [Mos04], a variant of SOS, provides an adequate framework for the independent description of language components [Mos09]. MSOS was designed to address the lack of reusability of SOS rules: *every* auxiliary entity used in a rule, such as an environment or a store, needs to be threaded through *all* rules of the language. MSOS provides a way to automatically propagate unmentioned entities between the premise(s) and conclusion of a rule, enabling the reuse of rules in different languages. SOS is very suitable for the formalization of languages and has therefore been widely adopted by the theorem prover community. MSOS has so far received less attention.

This chapter proposes a formalization of Component-Based Semantics based on MSOS in the theorem prover COQ [The12].[1] Our main contribution is a way to constructively formalize programming language semantics: basic constructs can be developed in separate COQ files, which may be verified independently. The formalization has been tested by building a small repository of constructs. Moreover, it is possible to equip the constructs with small proofs that can be used to construct larger proofs of properties holding for a full language. For this reason, we shall use the term *component* instead of *construct* in this chapter. Our formalization supports meta-theoretic reasoning about a programming language, but does not support reasoning about the format of MSOS rules.

The formalization follows the original design of MSOS in its use of arrows of a category for the auxiliary entities (encapsulated in labels) appearing in the transition rules. A very elementary level of knowledge about category theory and a modest amount of familiarity with theorem proving is required to read this chapter. Our formalization makes essential use of dependent types to formalize the labels in MSOS, and profits from COQ's support for type-classes. Each component is represented by a parametrized so-called COQ section. To define a full language, it is sufficient to enumerate its components. The correct instantiation of the corresponding parameters can in principle be performed automatically by COQ's powerful type system.

[1] The COQ files can be obtained via `http://www.cs.ru.nl/~kmadlene/fcbs.html`.

Syntactic Categories	
Cmd	commands
Exp	expressions
Dcl	declarations
Pcd	procedure abstractions
Prm	parameter patterns, encapsulating declarations

Constructs	
Cmd ::= seq (**Cmd**, ..., **Cmd**)	normal command sequencing
Cmd ::= skip	normal termination
Cmd ::= cond_loop (**Exp**, **Cmd**)	a simple while-loop, propagating abrupt termination
Cmd ::= catch (**Cmd**, **Pcd**)	tries to handle abrupt termination of **Cmd** by procedure abstraction **Pcd**
Cmd ::= throw **Exp**	terminates abruptly with the value of the **Exp**
Pcd ::= abs (**Prm**, **Cmd**)	a parametrized procedure abstraction (with static scoping)
Prm ::= eq **Exp**	a parameter that matches only the entity computed by the **Exp**.
Exp ::= block (**Dcl**, **Exp**)	locally binds **Dcl** in the **Exp**

Table 4.1: A basic repository.

4.2 Component-Based Semantics

We illustrate the description of programming languages in terms of basic abstract constructs by means of a while-loop example taken from [Mos09]. Depending on what concrete language is being analyzed, a standard command such as while may have different interpretations. For example, if the language includes a break command that abruptly terminates the program throwing a particular exception, then the description of while should include the handler for that exception. We assume that $Cmd[\![\_]\!]$ and $Exp[\![\_]\!]$ are functions mapping concrete expressions to abstract expressions of **Cmd** and **Exp**, respectively. Below, cond_loop is a simple while-loop that takes an expression and a command, and propagates abrupt termination. The other constructs involved can be found in Table 4.1. The description is then:

$$\begin{aligned} Cmd[\![\mathsf{while}\ (E)\ C]\!] &= \mathsf{catch}(\mathsf{cond\_loop}(Exp[\![E]\!],\ Cmd[\![C]\!]),\\ &\qquad\qquad \mathsf{abs}(\mathsf{eq}(\mathsf{breaking}),\ \mathsf{skip})) \\ Cmd[\![\mathsf{break}]\!] &= \mathsf{throw}(\mathsf{breaking}) \end{aligned}$$

A simple extension is a while-loop that handles continue commands. To describe such while-loops, all that is needed is to change the above example in such a way that $Cmd[\![C]\!]$ is encapsulated by a catch construct. Table 4.1 contains some possible constructs, which are used as examples throughout the rest of this chapter. See [Mos09] for an open-ended repository containing more constructs.

An important facet of Component-Based Semantics is that the construct repositories ideally contain no redundancy. If two basic constructs with different names have the exact same semantics, then one of them should be discarded. Moreover, if a construct can be expressed purely in terms of existing basic constructs, then this construct is generally also to be discarded. For instance, a while-loop testing for (non)zero corresponds to one with a boolean test combined with a predicate testing for (non)zero. However, one might well have both a while-do loop and a do-while loop as basic constructs, since neither is more basic than the other. A repository essentially describes a universal language that can be used to define the semantics of a concrete language in question. This universal language provides a fixed name for each basic construct, which in our formalization corresponds to the name of a COQ file.

In the rest this chapter we prefer to use the term *component* instead of *construct*, to emphasize we do not only refer to syntax when we use the term component, but also to its semantics and properties that it may be equipped with. For the semantics of each component to be language-independent, it is necessary that it does not depend on

- auxiliary entities that are not mentioned by the component;

- the transition relation of the full language;
- abstract syntax of the full language.

In our formalization we parametrize the components on these pieces of information. However, we first review MSOS, the framework our formalization is based on.

4.2.1 Modular SOS

In SOS, the operational semantics of a language with effects is modeled by a *labeled transition system (LTS)* $\langle \Gamma, A, \rightarrow \rangle$, where Γ is the set of configurations, A is the *set of actions*, and $\rightarrow \subseteq \Gamma \times A \times \Gamma$ is the *transition relation* (sometimes called *step relation*). It is possible to consider transition systems that also include terminal states, but these are only relevant if one desires to study terminating sequences of transitions, which is outside the scope of this chapter. A straightforward example of a set of configurations that we will use below is $\mathbf{Cmd} \times \rho \times \sigma$. We will call ρ and σ *auxiliary entities*, or simply *entities*.

A drawback of SOS is its lack of support for modularity. It is sometimes necessary to update existing rules by decorating the transitions with additional entities, e.g. a second store to model a separate part of memory. If we were to add an auxiliary entity to the configurations, then this entity needs to be threaded through *all* the rules that define the semantics. This prevents the rules from being reusable, and therefore plain SOS is not a suitable framework for Component-Based Semantics. One can get around this problem informally, by implicitly propagating the entities that are not mentioned, by using a convention (see also [CKS07]) such as:

$$\frac{\rho \vdash \langle c_1, \sigma \rangle \rightarrow \langle c_1', \sigma' \rangle}{\rho \vdash \langle \mathsf{seq}\ c_1\ c_2, \sigma \rangle \rightarrow \langle \mathsf{seq}\ c_1'\ c_2, \sigma' \rangle} \quad \rightsquigarrow \quad \frac{c_1 \rightarrow c_1'}{\mathsf{seq}\ c_1\ c_2 \rightarrow \mathsf{seq}\ c_1'\ c_2}$$

Normal command sequencing does not manipulate any of the entities and we can therefore assume that they are propagated. This informal description style enables formulation of rules independent of the auxiliary entities that may or may not be present and thereby provides reusability of the rules.

MSOS is a variant of SOS that has special support for the propagation of unmentioned entities. The key distinction is that it separates phrases of the language from entities by moving the entities into a label on the transition. That is, transitions are of the form $\gamma \xrightarrow{\alpha} \gamma'$, such that γ and γ' merely consist of abstract syntax (which may include computed values), and α is a label containing the auxiliary entities. Before we discuss the associated transition systems, let us consider some examples of rules specified in MSOS. Figures 4.1 and 4.2 provide examples of normal command sequencing and local bindings. The abstract

Label := {...}

$$\mathsf{seq}\ \mathsf{skip}\ c \to c \tag{4.1}$$

$$\frac{c_1 \xrightarrow{\{X\}} c_1'}{\mathsf{seq}\ c_1\ c_2 \xrightarrow{\{X\}} \mathsf{seq}\ c_1'\ c_2} \tag{4.2}$$

Figure 4.1: Normal command sequencing.

Label := {ρ : env, ...}

$$\frac{d \xrightarrow{\{X\}} d'}{\mathsf{block}\ d\ e \xrightarrow{\{X\}} \mathsf{block}\ d'\ e} \tag{4.3}$$

$$\frac{e \xrightarrow{\{\rho=\rho_0[\rho_1],X\}} e'}{\mathsf{block}\ \rho_1\ e \xrightarrow{\{\rho=\rho_0,X\}} \mathsf{block}\ \rho_1\ e'} \tag{4.4}$$

$$\mathsf{block}\ \rho_1\ v \to v \tag{4.5}$$

Figure 4.2: Local bindings.

syntax is standard, and the meta-variables c, d, e, ρ and v stand for commands, declarations, expressions, environments and values, respectively.

The meta-variable X plays an important rôle in the rules. It binds the unmentioned entities, allowing us to propagate them between the premise(s) and conclusion of each rule, without specifically describing what these entities are. Different occurrences of X in the same rule stand for the same entities. Note that the rules assume neither the presence or absence of particular auxiliary entities: the only entities that are mentioned are the ones used by the transitions in the rule in question. The Label box specifies what entities the label should at least include. Entities in labels can be matched in rules using notation such as "$\{\rho = \rho_0[\rho_1], X\}$", where $\rho_0[\rho_1]$ stands for updating ρ_0 by ρ_1.

Rules without labels on them are *unobservable*, meaning that they implicitly assume that the entities remain unchanged during the transition (e.g. in rule (4.1)). We remark that in this chapter skip too is a component: it has an empty label and an empty set of rules. A formal format for silent rules such as (4.1) has been introduced by Churchill and Mosses [CM13], which is based on the distinction between computations and values. See also Section 6.4 of this thesis.

Mosses [Mos04] recognized that the arrows of a category provide an adequate mathematical structure for labels. That is, two consecutive steps are only allowed to be made when their labels are composable, i.e., $\gamma \xrightarrow{p \longrightarrow q} \gamma' \xrightarrow{r \longrightarrow s} \gamma''$ is only allowed when $q = r$, and identity arrows correspond to the labels of unobservable transitions (as in rule 4.1). Hence, the associated transition systems are a triple $\langle \Gamma, A, \to \rangle$ similar to LTSs, with the difference that Γ strictly consists of abstract syntax, and the additional requirement that A are the arrows of a *label category* $\mathbb{A}$. The label category is a product of elementary categories that correspond to the entities, which we will discuss in Section 4.4. The values of the auxiliary entities

are the objects of $\mathbb{A}$. As an example, a simple step with rule (4.1) looks as follows, if the label contains an environment and a store:

$$\mathsf{seq}\ \mathsf{skip}\ c \xrightarrow{\langle \rho, \sigma \rangle \longrightarrow \langle \rho, \sigma \rangle} c \tag{4.6}$$

4.3 Formalization

In Component-Based MSOS, the source configuration γ of a transition $\gamma \xrightarrow{\alpha} \gamma'$ plays a special rôle. Namely, it determines to which component the rule permitting that particular transition belongs. The formalization defines for each component a so-called local transition relation, which describes the rules for source configurations that belong to that particular component.

Provided with the grammar of the full language, we construct the transition relation of the full language by combining the local transition relations. Components may optionally provide proof of a property that it satisfies, which can likewise be combined to build the proof of that property about the full language (if all components satisfy that property). This will be demonstrated in Section 4.5.

We make use of Coq's support for type-classes [SO08] to automatically "fill in the details", i.e. combining the components and filling in the parameters to construct the full language. Type-classes, however, are not strictly necessary for the formalization.

It is possible in our formalization to construct several full languages from the same repository. Extending an existing language with new constructs is not pursued in this chapter; this could potentially be done by leaving holes [DCB11] in the inductive datatype corresponding to the full language's grammar.

4.3.1 Types for transition relations

The transition relations of labeled transition systems (see Section 4.2.1) can be assigned the following type:

$$\mathit{Step}\ \Gamma\ A : \Gamma \to A \to \Gamma \to \mathbf{Prop}$$

In other words, a step is a predicate which takes arguments γ, α and γ' and returns an element of **Prop** (the built-in sort of propositional types in Coq). Just like the labeled transition systems associated with SOS specifications, there is no apparent distinction between syntax and the auxiliary entities.

Following the principles of MSOS, we update the type of *Step* to feature arrows of a category as labels on the transitions. *Step* now becomes parametric in the full label category $\mathbb{A}$ of the full language (which has a collection O of objects), resulting in the following type:

$$Step\ \Gamma\ O\ (\mathbb{A} : Category\ O) : \Gamma \to Arrows\ \mathbb{A} \to \Gamma \to \mathbf{Prop}$$

We have to remark that to avoid confusion, we are not following the exact syntax used in our formalization at this point. Moreover, we omit the definition of *Category* in this chapter, but we elaborate on *Arrows* in Section 4.4.

Component-Based MSOS requires both a modular way to specify the step relation and a modular way to specify the abstract syntax. The component seq of Figure 4.1 implicitly specifies its own signature, namely the production rule **Cmd** ::= seq **Cmd Cmd**, and specifies two new rules. It also assumes that a syntactical category **Cmd** exists, and to be able to define rule (4.2), it assumes that a transition relation on **Cmd** exists. We therefore parametrize the component (i.e. its local transition relation and lemmas) with Γ, representing the syntactic category, the full transition relation S on Γ, and the component's construct C (where P is a type that stands for its parameters, see the next section). Since the components always define the semantics for precisely one construct of the language, we restrict the input configuration to the phrases built by that construct. We call the transition relation of a component a *local step*, to emphasize the difference with a transition relation defined on a full syntactic category.

$$LocalStep\ \Gamma\ O\ (\mathbb{A} : Category\ O)\ (S : Step\ \Gamma\ O\ \mathbb{A})\ P\ (C : Construct\ P\ \Gamma) :$$
$$restr\ C \to Arrows\ \mathbb{A} \to \Gamma \to \mathbf{Prop}$$

To define the full language, it is sufficient to enumerate the components it is built of. This results in a transition relation of type *Step* for each syntactic category, which we call a *global step relation*. This is described later on in this section.

4.3.2 Grammar

As a running example, we define a programming language that consists of just the basic constructs skip and seq (see Figure 4.1). Although it is a fairly simple example, it allows us to explain the formalization without having to get ahead too much on labels, which are treated in Section 4.4. In component-based descriptions of actual programming languages, the semantics is to be defined by a (context-free) translation to basic constructs. Basic constructs are designed to be independent of a particular programming language and as a result are often too simple to be directly mapped to constructs of an actual programming language.

The grammar of our skip − seq language is straightforwardly encoded by the following inductive type:

$$\mathbf{Inductive}\ Cmd := skip \mid seq\ (c_1\ c_2 : Cmd)$$

Recall from Section 4.2 that each component is parametrized on its abstract construct. The arguments are passed on as an injection-projection pair which we will call *Construct*. Injection corresponds to applying a constructor and projection corresponds to pattern matching. *Construct* consists of two properties saying that i and p are (partial) inverses of each other. This is needed to prove properties about the component.

```
Class Inject P Γ := inject : P → Γ
Class Project P Γ := project : Γ → option P
Class Construct P Γ {i : Inject P Γ} {p : Project P Γ} := {
  H_i : ∀ x : P, p (i x) ≡ Some x;
  H_p : ∀ γ : Γ, match project γ with
                 | None   ⇒ True
                 | Some x ⇒ i x ≡ γ
                 end
}
```

For constructs that take several arguments, such as **Cmd** ::= seq **Cmd Cmd**, the arguments are tupled. The **Class** keyword declares the definitions to be type-classes. The convenience of type-classes is that class fields (such as *inject* or *project*) may be used without explicitly mentioning which instance of that class should be used. The curly brackets around i and p indicate that these arguments are implicit. In this case, these implicit arguments become class constraints, i.e., order to build an instance of *Construct*, instances of *Inject* and *Project* need to be present. For our example language, the corresponding instances are:

```
Instance : Inject unit Cmd := λ _, skip
Instance : Inject (Cmd × Cmd) Cmd :=
  λ p, let (c1, c2) := p in seq c1 c2
Instance : Project unit Cmd :=
  λ γ, match γ with
       | skip ⇒ Some tt
       | _    ⇒ None
       end
Instance : Project (Cmd × Cmd) Cmd :=
  λ γ, match γ with
       | seq c1 c2 ⇒ Some (c1, c2)
       | _         ⇒ None
       end
Instance : Construct unit Cmd
```

Instance : $\mathit{Construct}\ (\mathit{Cmd} \times \mathit{Cmd})\ \mathit{Cmd}$

The type-class mechanism can be seen at work here: we do not have to specify the arguments i and p, for they can be resolved from the signatures. In fact, the manual declaration of these type-class instances is straightforward and can be omitted by an augmentation of COQ's type-class resolution algorithm, but we skip the details here. The reader may have noted that when the full language has two constructs with the same signature, the type-class instance resolution algorithm may fill in the wrong *Construct* instance. This is solved in the formalization by adding an argument (i.e. a *string*) to *Construct*, enabling us to uniquely identify each instance.

Returning to the *LocalStep* type, the *Construct* argument is actually a class constraint (i.e. it is an implicit argument) in the formalization. In fact, the category and the *Step* relation are also class constraints. Some components require the presence of other components. For instance, the component seq "imports" the (very basic) component skip. To this end, the *Skip* construct becomes an additional constraint of seq. This does not interfere with modularity: all other details about the full language remain opaque.

4.3.3 Semantics

A straightforward way to encode transition relations in a theorem prover is by means of an inductive predicate [BHLP09]. Making the definition inductive guarantees that the only valid transitions are the ones that can be built by its constructors, which correspond to the rules. The encoding of rules is straightforward using nested implications, where universal quantifications are added for variables that occur in the rules. As an example, we give the transition relation for seq:

Inductive $\mathit{ls} : \mathit{restr}\ \mathit{Seq} \to \mathit{Arrows}\ \mathbb{A} \to \mathit{Cmd} \to \mathbf{Prop} :=$
$\mid \mathit{seq}_1 : \forall\ c_1\ c_2\ c_1'\ \mathit{ar}, \mathit{step}\ c_1\ \mathit{ar}\ c_1' \to \mathit{ls}\ (\mathit{Seq} \cdot (c_1, c_2))\ \mathit{ar}\ (i\ (c_1', c_2))$
$\mid \mathit{seq}_2 : \forall\ c_2\ \mathit{ar}, \mathit{unobs}\ \mathit{ar} \to \mathit{ls}\ (\mathit{Seq} \cdot (\mathit{skip}\ \mathit{tt}, c_2))\ \mathit{ar}\ c_2$

The premise *unobs ar* expresses unobservability of the label, i.e., it is an identity arrow. We have suppressed the class constraints here for readability. That is, *ls* requires suitable instances of *Category*, *Step*, *Construct* and *Label* (the latter is presented in Section 4.4). The type *restr C* is used to restrict phrases of the full language to ones built by constructor C. By means of an inductive type with a single constructor, we can ensure that the only way to build an instance of type *restr C* is by providing an object of P:

Inductive $\mathit{restr}\ `(C : \mathit{Construct}\ P\ \Gamma) := \mathit{restr\_cons}\ (\gamma : \Gamma)$
Notation $C \cdot \gamma := (\mathit{restr\_cons}\ C\ \gamma)$

The backtick performs implicit generalization: necessary variables to the argument C are automatically declared as implicit arguments of *restr*. Writing e.g. $Seq \cdot (c_1, c_2)$ is similar to applying the "real" constructor (e.g. *seq* c_1 c_2), but not exactly the same. One can obtain c_1, c_2 by straightforward pattern matching on *restr_cons*. In contrast, it is only possible obtain c_1, c_2 from *seq* c_1 c_2 by using the elimination principle of *Cmd*, which is not available inside the component.

The inductive predicate *ls* is made into a type-class instance to enable resolution:

Instance *LS_Seq* : *LocalStep O* := *ls*

The semantics of the full language is essentially defined by a case distinction on the constructors of the datatypes. The full step relation is defined as an inductive predicate *s* that combines the existing local step relations of the used components into one global step relation. This is done by means of an inductive predicate that has a single constructor. The constructor assumes a *localstep* of any of the local transition relations of the syntactic category in question (passing along *s* itself), and returns an object of *s* (as above, in *ls*). The reader interested in the details is referred to the source code. This construction satisfies equations such as:

$$\begin{aligned} \textit{localize Skip S Cmd} &= \textit{LS\_Skip} \\ \textit{localize Seq S Cmd} &= \textit{LS\_Seq} \end{aligned}$$

The operator *localize* maps the given *Step* instance (in this case *S_Cmd*) to the canonical *LocalStep* w.r.t. the provided construct. These equations are necessary to prove properties about the components. For example, consider the component seq, which imports the component skip. To be able to prove properties about seq, the local step relation of skip (which is empty) needs to be accessible. This is done by passing on the first equation as an argument. The equality is overloaded with the obvious meaning that the *Step* instances agree on all inputs (i.e. *ar*, γ and γ'). In conjunction with Coq's built-in support for setoid rewriting (rewriting modulo an equivalence relation), this enables us to perform short proofs for meta-theory (used in Section 4.5).

4.4 Labels

Auxiliary entities such as environments and stores in SOS are encapsulated in a label on the transitions in MSOS. In Section 4.2 we have explained that the labels on the transitions have the structure of arrows of a category: the labels of consecutive transitions should be composable. A subtle difference between MSOS and SOS is that the chosen label category may restrict the transition relation

specified by the rules, whereas in SOS it is solely the rules that determine this relation. This can be seen by assuming the label category to be a discrete category, i.e., the category with just identity arrows.

Mosses [Mos04] has shown that a suitable category is the product $\mathbb{A} \widehat{=} \prod_{i \in I} \mathbb{A}_i$ of elementary categories representing the auxiliary entities. The usual types of entities used in SOS rules are environments, stores and labels, which correspond to read-only, read-write or write-only permissions, respectively. In MSOS, each entity (with index i) has a corresponding set of objects S_i that, together with the permissions, determines its corresponding category $\mathbb{A}_i$:

- read-only: $\mathbb{A}_i$ is the discrete category with S_i as its objects;
- read-write: $\mathbb{A}_i$ is the pre-order category with S_i as its objects, and S_i^2 as its morphisms;
- write-only: $\mathbb{A}_i$ is the category with a single object $*$, and the free monoid on S_i as its morphisms.

A distinguishing feature of MSOS is its inherent support for write-only entities. For example, a transition in a system with a single write-only entity can be pictured as $\gamma \xrightarrow{* \xrightarrow{\alpha} *} \gamma'$, where α is the name of the arrow (there are usually infinitely many for write-only components). If it appears as the conclusion of a rule, then the premises of that rule cannot possibly depend on the value of that entity, because it is simply $*$. For this reason, we have adopted the use of arrows as labels in our formalization. Note that if one were to replace the arrows as labels with relations on product entities, then the notion of write-only labels would be lost (when S_i has at least one element in it).

Recall that the components are parametrized by a label category $\mathbb{A}$ on a collection of objects O. To build the product category, O is instantiated with the entity map $i \mapsto \mathbb{A}_i$. Inside the component, the label category is entirely opaque. In other words, it is impossible to learn anything from $\mathbb{A}$ except that it is a product category. The Label box in the component specification expresses what entities the full label should *at least* include. For example, Figure 4.2 requires that the full label includes an environment entity. This is reflected in our formalization by providing two functors P_M and P_U to each component, that project full labels to their mentioned entities and unmentioned entities, respectively:

$$\mathbb{A} \xrightarrow{P_M} \prod_{i \in M} \mathbb{A}_i, \qquad \mathbb{A} \xrightarrow{P_U} \mathbb{U}.$$

The idea is that the product of mentioned entities is transparent to the component, whereas $\mathbb{U}$ is opaque. We use the functor P_U to express unobservability, needed

e.g. in rule (4.1). Additionally, the component requires that (P_M, P_U) is an isomorphism, which is crucial to enable modular proof. Let us consider determinism as an illustration of this.

Property 10. *Assume configurations* $\gamma, \gamma', \gamma'' : \Gamma$ *and labels* $ar' : x \longrightarrow y$, $ar'' : x \longrightarrow z$. *The step relation on* Γ *is* deterministic *when both* $\gamma \xrightarrow{ar'} \gamma'$ *and* $\gamma \xrightarrow{ar''} \gamma''$ *imply that* $\gamma' = \gamma''$ *and* $ar' = ar''$.

The requirement that the arrows are equivalent ensures not only that the post configurations are equal, but also the outputs through the write-only components are equal. To prove that the component seq is deterministic, one proceeds by straightforward case analysis on the structure of the input configuration. In the case that it is seq skip c, we have two arrows ar', ar'' such that $P_U\ ar' = P_U\ ar'' = id$, and $P_M\ ar' = P_M\ ar'' = ()$ (the empty tuple). In other components that do have mentioned entities, these projections of P_M have to be equivalent. Using the isomorphism we can then conclude that $ar' = ar''$.

4.4.1 Formalization of labels

The category theory we have used in our formalization is provided by the MATHCLASSES library by van der Weegen and Spitters [SvdW11]. Their library makes extensive use of a technique called "unbundling", which boils down to separating the components of mathematical structures into separate type-classes. An example of this are categories. In Section 4.3.1, we have treated *Category* as a record structure containing *Arrows* as a field for presentation purposes. However, in the actual formalization, *Arrows* is a separate type-class:

Class *Arrows* $(O : \mathbf{Type}) : \mathbf{Type} := Arrow : O \to O \to \mathbf{Type}$
Infix $\longrightarrow$:= *Arrow*

To build a *Category*, among other components, an equivalence relation on the corresponding *Arrows* instance is necessary, to enable the comparison of arrows. We use this relation in our formalization to define the predicate *unobs* for unobservability. The following instances are used for the entity categories:

Instance *arrows_ro* : *Arrows O* $:= \lambda\, x\, y, x \equiv y$
Instance *arrows_rw* : *Arrows O* $:= \lambda\, x\, y, unit$
Instance *arrows_wo* : *Arrows unit* $:= \lambda\, x\, y, list\ O$

We now define the type-class *Label*, which is used to provide the projection functors. *Label* assumes the presence of the following objects:

$I\ M : \mathbf{Type}$
$O : I \to \mathbf{Type}$
$A : \forall\ i : I, Arrows\ (O\ i)$
$O\_M : M \to \mathbf{Type}$
$A\_M : \forall\ i : M, Arrows\ (O\_M\ i)$

In other words, for both index sets I and M it is required that a collection of arrows exists.

Class *Label* := {
  $cover\_O : \forall\ i : M, O\_M\ i \equiv O\ (to\_I\ i);$
  $cover\_A : \forall\ i : M, A\_M\ i \equiv \langle\ \lambda\, T, Arrows\ T \mid eq\_sym\ (cover\_O\ i)\ \rangle\ A\ (to\_I\ i)$
}

The *cover_O* property says that for every index of the mentioned entities, the objects have to correspond to the objects of the full category. Likewise, the arrows of the mentioned entities have to correspond. A cast operation [Hur10] on the objects (indicated by $\langle \_, \_ \rangle$) is needed to be able to express the latter, but we omit the details in this chapter. Given an instance of *Label*, we can derive the functors P_M and P_U together with the fact that they are isomorphic. Each component has a *Label* type-class constraint which leaves O and A parametric, but specifies O_M and A_M.

To illustrate how a rule is interpreted with help of the *Label* construction, we consider rule (4.4) of Figure 4.2. Let us first write it using informal notation. Assume that $ar : x \longrightarrow y$, $ar' : x' \longrightarrow y'$ and $proj_\rho = \pi_\rho \circ P_M$ is the projection of the component with index ρ.

$$\frac{proj_\rho\ x' = \rho_0[\rho_1] \qquad proj_\rho\ x = \rho_0 \qquad P_U\ ar = P_U\ ar' \qquad e \xrightarrow{ar'} e'}{\mathsf{block}\ \rho_1\ e \xrightarrow{ar} \mathsf{block}\ \rho_1\ e'}$$

In COQ-syntax, this rule is:

$rule4 :$
  $\forall\ (\rho_0\ \rho_1 : Env)\ (e\ e' : Exp)\ `(ar' : x' \longrightarrow y'),$
    $proj_\rho\ x' \equiv update\ \rho_0\ \rho_1 \to proj_\rho\ x \equiv \rho_0 \to$
    $fmap\_P_U\ ar = fmap\_P_U\ ar' \to step\ ar'\ e\ e' \to$
  (* − *)
    $ls\ (Block \cdot (\rho_1, e))\ ar\ (i\ (\rho_1, e'))$

Note that the use of equality in the above code is highly overloaded, which is made possible by the use of type-classes. Like the MATH-CLASSES library, we represent the functors by means of a function that maps the objects, which have the actual names P_M and P_U, and functions that map the arrows, which have the *fmap_* prefix.

4.5 Example of Modular Proof

Once the full language is declared, it is possible to combine proofs of the components to prove that a particular property holds for the full language. Like the local step relations, properties are parametrized by a global step relation S. We say that a property holds for a step relation if it holds for all the possible configurations, but we are a bit more general and allow the user to express that a property holds for a particular configuration.

Not all properties can be proved by induction, and likewise not all properties have a modular proof. We consider a class of admissible, well-behaved properties P such that $P\ S\ (I\ \gamma)$ does not depend on anything but the localized version of S w.r.t. C (here $I\ \gamma$ injects γ into Γ):

Definition *admissible* $\Gamma\ O\ (P : Step\ \Gamma\ O\ \mathbb{A} \to \Gamma \to \mathbf{Prop}) :=$
$\forall\ `(C : Construct\ A\ \Gamma)\ (S : Step\ \Gamma\ O\ \mathbb{A})\ (\gamma : restr\ C),$
$P\ (globalize\ (localize\ C\ S))\ (I\ \gamma) \to P\ S\ (I\ \gamma)$

The operator *globalize* is the reverse of *localize*: it takes a local step relation *ls* and makes it global, behaving like *ls* on phrases constructed by C and not permitting any steps to be made that start from other configurations. The idea of admissible properties is that they warrant that proof by induction is possible.

Lemma 1. *Determinism is admissible.*

We will demonstrate how this lemma is used to show that our $\mathsf{skip} - \mathsf{seq}$ language is deterministic by illustrating the seq case (skip is similar). Inside the Coq section of seq, we have proved the following lemma that says that the component is deterministic.

Lemma *det_Seq* $(c_1\ c_2 : Cmd)$:
$det\_global\ S\_Cmd\ c_1 \to det\_local\ LS\_Seq\ (Seq \cdot (c_1, c_2))$

Note that it assumes that the global step relation is deterministic on c_1, which is essentially the induction hypothesis.

Two *Step* or *LocalStep* instances are considered equivalent if they agree on the transitions between each triple consisting of two states and a label. Both *det_global* and *globalize* respect this notion of equivalence. Using Coq's built-in support for rewriting modulo equivalence relations (called setoid rewriting), it can be shown that:

$$
\begin{array}{lll}
 & \mathit{det\_global}\ \mathit{S\_Cmd}\ (\mathit{seq}\ c_1\ c_2) & (\text{fold } I) \\
= & \mathit{det\_global}\ \mathit{S\_Cmd}\ (I\ (\mathit{Seq} \cdot (c_1, c_2))) & (\text{Lemma 1}) \\
= & \begin{array}{l}\mathit{det\_global}\ \ (\mathit{globalize}\ (\mathit{localize}\ \mathit{Seq}\ \mathit{S\_Cmd})) \\ \qquad\qquad (I\ (\mathit{Seq} \cdot (c_1, c_2)))\end{array} & (\text{rewrite } \mathit{eq\_Seq}) \\
= & \mathit{det\_global}\ (\mathit{globalize}\ \mathit{LS\_Seq})\ (I\ (\mathit{Seq} \cdot (c_1, c_2))) & (\text{fold } \mathit{det\_local}) \\
= & \mathit{det\_local}\ \mathit{LS\_Seq}\ (\mathit{Seq} \cdot (c_1, c_2)) &
\end{array}
$$

Now, the latter holds because this is a property proved in the component seq. The proof for seq can therefore be completed by applying *det_Seq*, using the equation $\mathit{localize\ Skip\ S\_Cmd} = \mathit{LS\_Skip}$ and the induction hypothesis.

Other components follow the same prescription. In future work, we want to automate the weaving of local proofs by generalizing the above, and exploiting automated proof search with the help of the type-class mechanism in COQ. Experiments have already demonstrated that this is feasible, but fragile.

4.6 Related Work

A specification language for MSOS, called MSDF, short for MSOS Definition Formalism, has been developed by Mosses [Mos06] and revisited by Mosses and Chalub [CB07]. MSDF combines BNF notation with textual representation of MSOS transitions, and a large number of basic components have already been identified and specified in it. A tool that translates ML and (a part of) JAVA into this repository have been developed by Chalub and Braga [CB07], which can be executed in the MAUDE tool. MSDF provides its own specification language for datatypes, which can be constructed from primitives such as sequences, lists, maps, etc. In contrast, our formalization directly uses types defined in COQ.

Implicit-MSOS is an improvement of MSOS that reduces the amount of clutter in the rules even further by implicitly propagating unmentioned entities [MN09]. The interpretation of Implicit-MSOS is given in terms of MSOS, and we expect that it can be built on top of our formalization by clever use of type-classes.

The work of Delaware et al. focuses on the development of modular programming language meta-theory in COQ. They show how type-safety of Featherweight JAVA and its extensions can be proved in a modular fashion [DCB11]. In more recent work, they applied type-classes to compose proofs from modular components [DOS13], and support semantics with side-effects in their modular meta-theory [DKSO13].

The formalization of the operational semantics of $\text{OCAML}_{\text{light}}$ in HOL by Scott Owens makes use of labels to encode mutations to the store in them [Owe08]. These mutations are correlated to a reduction in the program. The labels explicitly

carry mutations and therefore simplify the notation, but do not enable a high degree of reusability of the rules.

In a theorem prover (and functional languages), abstract syntax and transition relations are typically encoded as inductive types, of which the constructors correspond to the grammar production rules and the constructors correspond to the rules of the step relation. The inductive definition ensures that those constructors are the only way to build instances of those types. This corresponds to the notions "initial algebra" and "least relation", sometimes used in this context (e.g. [MN09]). To facilitate Component-Based Semantics, we have to be able to build these inductive types from "partial versions" that define just the rules and production rules of the component in question. To our best knowledge, there is no theorem prover (or functional language) that supports (multiple) inheritance of inductive types natively.

4.7 Conclusions and Future Work

In this chapter we have presented a formalization of Component-Based Semantics in the theorem prover Coq. The formalization makes essential use of dependent types, and profits from Coq's support for type-classes. Our formalization is based on the ideas of MSOS, and makes use of the idea of labels as arrows in categories, as proposed by Mosses [Mos04]. Splitting the label category into a transparent part for the mentioned entities and an opaque part for the unmentioned entities enables modular proof. We have demonstrated this by crafting a proof of determinism of a mini-language from smaller local proofs provided by the components used.

In future work, the work in this chapter can be used to enable scalable verification of specific programs. Another direction of further research is to investigate whether the full generality of labels as arrows (which our formalization provides) can be exploited for entities of types other than read-only, read-write and write-only. We expect that by choosing a suitable category, it is possible to enforce information flow policies, which has applications to security.

Acknowledgments. The author would like to thank Peter D. Mosses for introducing him to the notion of Component-Based Semantics, and Bas Spitters for introducing him to type-classes in Coq. The author would also like to thank Peter D. Mosses, Julien Schmaltz and the anonymous reviewers for their comments on an earlier version of the published version of this chapter.

CHAPTER 5

GSOS Formalized in Coq

Abstract. Structural operational semantics provides a well-known framework to describe the semantics of programming languages, lending itself to formalization in theorem provers. The formalization of syntactic SOS rule formats, which enforce some form of well-behavedness, has so far received less attention. GSOS is a rule format that enjoys the property that the operational semantics and denotational semantics, both derived from the same set of GSOS rules, are consistent. This chapter formalizes the underlying theory in the theorem prover COQ, and proves the consistency property, also known as the adequacy theorem. The inspiration for our work has been drawn from the field of bialgebraic semantics.

5.1 Introduction

Operational and denotational semantics are two well-known approaches to assigning a meaning to programming languages and process algebras. Around fifteen years ago, Turi and Plotkin [TP97] developed a framework that unifies both these styles. Using the language of category theory, they managed to strip away language-specific details such as concrete syntax and behavior. Given a set of operational rules, they have shown how to derive both the operational and denotational semantics from a distributive law corresponding to a set of operational rules. Their result pertains to several syntactic operational rule formats [Bar04,

HJ11], but the GSOS format [BIM95] is the most prominent one of them [Bar04, Kli11]. For instance, classic languages such as basic process algebra and the language WHILE [BHLP09] can be described by the GSOS format. Although implementations based on Turi and Plotkin's work have previously been developed in HASKELL by Hutton [Hut98], Jaskelioff, Hutton and Ghani [JGH11], and Hinze and James [HJ11], formalized proofs in a theorem prover have not yet been provided.

The contributions of this chapter are the following.

- It provides an implementation of both an operational and a denotational semantics derived from a set of GSOS rules, and a proof of their consistency (called the "adequacy theorem"), fully developed in the theorem prover COQ [The12], using novel theorem proving techniques.
- A generic theory for syntactic terms, also fully developed in COQ, which is needed for the proof of the adequacy theorem.
- It lays out the foundations for further work on the formalization of bialgebraic semantics.

The advantage of a development in the constructive logic of COQ is that it enables both the execution of and formal proofs about the semantics at hand. The work in this chapter is heavily inspired by Turi and Plotkin's bialgebraic semantics. The presented formalization is a shallow embedding into COQ's **Type** (i.e. the type of types), and does not possess the full generality of Turi and Plotkin's category theoretic work. However, it is still far more general than previous work on programming language semantics in theorem provers which usually concentrates on the study of a concrete language, such as [BHLP09].

An important tenet of most theorem provers, including COQ, is that every function must terminate, otherwise the underlying logic would be inconsistent. This seemingly superficial difference with HASKELL has profound implications for the development presented in this chapter. In order to satisfy the syntactic checks which COQ performs on definitions to guarantee termination, the types representing the syntax and behavior of the language must be chosen carefully. As we will see, COQ's support for dependent types can be put to good use. The semantic domain potentially consists of infinite objects, as shown in the examples provided in this chapter. In contrast to HASKELL, there is a clear distinction between finite and infinite worlds in COQ. The standard syntactic equality of COQ is not general enough for serious proofs about infinite objects. We have based the COQ formalization on the use of setoids, i.e. a **Type** packaged with a user-defined notion of equality and a proof of well-behavedness of the equality.

To make the content in this chapter accessible to readers with limited exposure to the field of bialgebraic semantics and COQ, we explain the computational side

$$\frac{}{AS \xrightarrow{\mathsf{a}} AS} \qquad \frac{}{BS \xrightarrow{\mathsf{b}} BS}$$

$$\frac{x \xrightarrow{l} x' \qquad y \xrightarrow{m} y'}{Alt\ x\ y \xrightarrow{l} Alt\ y'\ x'} \qquad \frac{x \xrightarrow{l} x'}{Zip\ x\ y \xrightarrow{l} Zip\ y\ x'}$$

Figure 5.1: A simple language for streams.

of this work in Section 5.2, using a simple language for the construction of streams as our running example. Furthermore, we start off with a more limited rule format to sketch the main ideas, and then treat the GSOS format. The reader is expected to have a modest amount of familiarity with category theory. All definitions and theorems in this chapter have been formalized and proved in Coq.[1]

5.2 A Simple Stream Language

In this section we discuss the relation between operational and denotational semantics similar to the way they arise in the framework of Turi and Plotkin [TP97]. We use a simple language about streams (also used by Klin [Kli11] in his introduction to bialgebraic semantics) as our running example, allowing us to explain the basics and provide some hints towards the Coq implementation. An example of a more involved language featuring non-determinism will be given in Section 5.5. This section is almost exclusively limited to the computational side of our work, and the code presented in this section is perfectly executable within Coq.

Consider the simple stream language defined by the operational rules in Figure 5.1. For now we disregard the operation *Zip*. These rules inductively define a transition relation between terms, composed of the operations *AS*, *BS*, *Alt*, and a label, a or b. The operations *AS* and *BS* respectively generate the streams $\mathsf{aaaa}\cdots$ and $\mathsf{bbbb}\cdots$, and the operation *Alt* generates the alternation between its two provided streams.

The syntax of the language is specified by its signature: a set of function symbols each equipped with a fixed arity. Such a signature is encoded as a functor which we will call the *signature functor*. If Σ denotes the signature functor, then the terms are least fixpoint of Σ, usually denoted as $T := \mu X, \Sigma X$. The corresponding types are:

Inductive $\Sigma\, X := AS \mid BS \mid Alt\ (x\ y : X)$

[1] The Coq files can be obtained via `http://www.cs.ru.nl/~kmadlene/adequacy`.

Inductive $T := app\ (\sigma : \Sigma\ T)$

Here X is a **Type**, and *app* is a constructor of T with the type $\Sigma\ T \to T$. The separation of Σ from T will be important in the rest of the chapter. We provide the function map corresponding to the signature functor as well. This is an instance of the type-class *SFmap* (i.e. setoid function map), which will be discussed in more depth in Section 5.4.1.

Class $SFmap\ (M : \mathbf{Type} \to \mathbf{Type}) :=$
  $sfmap_M : \forall\ X\ Y\ (f : X \to Y), M\ X \to M\ Y$
Instance $: SFmap\ \Sigma :=$
  $\lambda\ X\ Y\ (f : X \to Y)\ x,$
    match x **with**
    $\mid AS \Rightarrow AS$
    $\mid BS \Rightarrow BS$
    $\mid Alt\ x\ y \Rightarrow Alt\ (f\ x)\ (f\ y)$
    end

5.2.1 Operational semantics

To encode the transition relation we also have to represent the behavior of the system. This again is done with a functor, which we call a *behavior functor*. The data type L corresponds to our label set.

Inductive $L := \mathsf{a} \mid \mathsf{b}$
Definition $B\ X := L \times X$
Instance $: SFmap\ B :=$
  $\lambda\ A\ B\ (f : A \to B)\ a,$
    let $(l, x) := a$ **in** $(l, f\ x)$

We can now define a transition system as a *B-coalgebra*, i.e. a pair consisting of a state-space, in this case T, and a structure map of type $T \to B\ T$. In this chapter we refer to the structure map as the coalgebra.

Fixpoint $op\ (t : T) : B\ T :=$
  $sfmap_B\ app\ ($
    match t **with**
    $\mid app\ AS \Rightarrow (\mathsf{a}, AS)$
    $\mid app\ BS \Rightarrow (\mathsf{b}, BS)$
    $\mid app\ (Alt\ x\ y) \Rightarrow$

$$\begin{array}{l}\quad\textbf{let}\ (l,x') := op\ x\ \textbf{in}\\ \quad\textbf{let}\ (m,y') := op\ y\ \textbf{in}\ (l, Alt\ y'\ x')\\ \textbf{end})\end{array}$$

One may think of *op* as a model of the operational semantics of the language in question (cf. the rules in Figure 5.1), as it specifies for each state what the next step would be. We can run a term by coiteratively unfolding *op*, resulting in a stream of labels. The streams are actually the greatest fixpoint of the present behavior functor [JR97], usually denoted as $\nu X, BX$. This leads to the following definitions:

$$\begin{array}{l}\textbf{CoInductive}\ Z_B := in_B\ (x : B\ Z_B)\\ \textbf{CoFixpoint}\ unfold_B\ `(c : X \to B\ X)\ (x : X) : Z_B :=\\ \quad in_B\ (sfmap_B\ (unfold_B\ c)\ (c\ x))\\ \textbf{Definition}\ run := unfold_B\ op\end{array}$$

Thus, in_B is the constructor of the streams Z_B, and has type $B\ Z_B \to Z_B$. We call *run* the *operational semantics*, which is derived from *op*, the *operational model*.

In this chapter we will only consider behavior functors B that have a final coalgebra. The existence of final coalgebras is a subtle matter; in HASKELL it is possible to define the greatest fixpoint of an arbitrary functor (which acts as the state-space of the final coalgebra), but in COQ the same definition is illegal due to the inability of COQ to guarantee the termination of every function in that case. The same goes for the least fixpoint operator, for the terms. The least and greatest fixpoint operators in HASKELL-code would be respectively (the argument f being the functor):

$$\begin{array}{ll}\textbf{data}\ \mu f & = \mu\ (f\ (\mu f))\\ \textbf{codata}\ \nu f & = \nu\ (f\ (\nu f))\end{array}$$

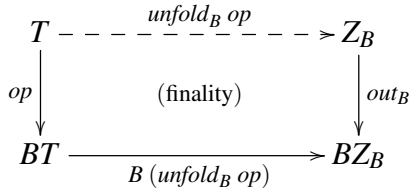
However, because there is no distinction in HASKELL between finite and infinite worlds, these operators are exactly the same.[2]

The structure map of the final coalgebra is called out_B:

$$\begin{array}{l}\textbf{Definition}\ out_B\ (z : Z_B) : B\ Z_B :=\\ \quad\textbf{match}\ z\ \textbf{with}\\ \quad\mid in_B\ x \Rightarrow x\\ \quad\textbf{end}\end{array}$$

[2] In HASKELL, the keywords **data** and **codata** are only used to indicate the *intended* use of the datatype, but can be interchanged.

An important property of the final coalgebra is that $unfold_B$ is the only function that makes the following diagram commute[3], and this function is *run* (= $unfold_B\ op$).



The intuition behind the above diagram for the concrete behavior functor B for streams, used in this section, is that splitting a label off the stream generated by unfolding *op* is the same as performing one step and unfolding *op* on the resulting term.

5.2.2 Denotational semantics

As in [TP97] we consider the denotational semantics as a dual version of the operational semantics. The underlying denotational model actually operates directly on elements of the semantic domain of our stream language, i.e. the streams Z_B. For the present example this means that it prescribes how the operations of the language, which receive streams as arguments, yield new streams. The semantic functions corresponding to each of the operations are:

```
CoFixpoint denAS : Z_B := in_B (a, denAS)
CoFixpoint denBS : Z_B := in_B (b, denBS)
CoFixpoint denAlt (x y : Z_B) : Z_B :=
  match (x, y) with
  | (in_B (l, x'), in_B (_, y')) ⇒ in_B (l, denAlt y' x')
  end
```

From the above functions we can define the full denotational model:

```
Definition den (σ : Σ Z_B) : Z_B :=
  match σ with
  | AS      ⇒ denAS
  | BS      ⇒ denBS
  | Alt x y ⇒ denAlt x y
  end
```

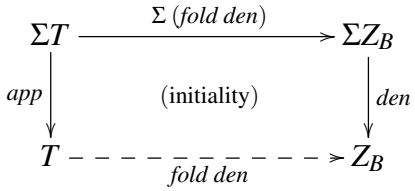
[3] In the diagrams of this chapter we will adopt the categorical notation for functors by writing F instead of $sfmap_F$, for some functor F, i.e., we explicitly indicate the instance type, and leave $sfmap_F$ itself implicit. Moreover, we omit parentheses in the notation of types.

In a fashion dual to the operational side, the denotational semantics evaluates a term, by folding the algebra *den* over that term.

Fixpoint *fold* '$(h : \Sigma\, X \to X)\ (t : T) : X :=$
  match t **with**
  $\mid app\ \sigma \Rightarrow h\ (sfmap_\Sigma\ (fold\ h)\ \sigma)$
  end

Definition $eval : T \to Z_B := fold\ den$

The terminology is that *den* is the *denotational model*, while *eval* is the *denotational semantics*. Likewise, there is a unique function making the following diagram commute, and this function is *eval* (= *fold den*).



5.3 Framework

The adequacy theorem in the case of the simple stream example says that executing *run* and *eval* on the same term yields the same stream. With the definitions as they stand, a proof of it would proceed by induction on the terms of the stream language, and would therefore be rather ad hoc. A more structured development will be laid out in the present section, based on the use of a distributive law to represent operational rules. From this distributive law we derive both operational and denotational models.

As we have detailed in the previous section, it is not possible to take arbitrary fixpoints in COQ due to limitations imposed by its logic. To develop our theory independent of a particular signature or type of behavior, we will assume the existence of least/greatest fixpoints with the appropriate properties. That is, the fixpoints and their properties are parameters of the theory we develop in this chapter. This is all possible in COQ as proofs and programs are in the same syntactic class, in true Curry-Howard style. The question is then how to realize these fixpoints in a fairly generic fashion. We do this for the terms in the present chapter.

5.3.1 Generic terms

A more general version of T that does not directly depend on a specific signature cannot be obtained by making T parametric in the signature in COQ, as we have

explained earlier. Instead, we fix T on the signature Σ, and Σ should have a certain shape. We will discuss the details of this in Section 5.4.2. To allow for open terms (used later on to represent meta-variables X in the operational rules) the constructor $var : X \to T\ X$ has been added.

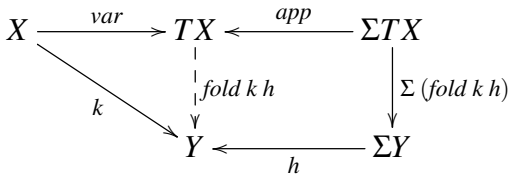
Inductive $T\ X := var\ (x : X) \mid app\ (\sigma : \Sigma\ (T\ X))$

T is also called the *free monad generated by* Σ. It is straightforward to generalize the *fold* provided earlier to the new version of T.

Fixpoint $fold\ (k : X \to Y)\ (h : \Sigma\ Y \to Y)\ (t : T\ X) : Y :=$
  match t **with**
  $\mid var\ x \Rightarrow k\ x$
  $\mid app\ \sigma \Rightarrow h\ (sfmap_\Sigma\ (fold\ k\ h)\ \sigma)$
  end

The *fold* operation provides a recursive definition principle that avoids explicit recursion (see e.g. [MFP91]): one only has to specify a mapping of the variables $k : X \to Y$, and an algebra $h : \Sigma\ Y \to Y$. This result is attributed to the following lemma.

Lemma 2. *Let $k : X \to Y$, and $h : \Sigma\ Y \to Y$. Then fold k h is the unique function making the following diagram commute:*



The equalities $fold\ k\ h \circ var = k$ and $fold\ k\ h \circ app = h \cdot \Sigma\ (fold\ k\ h)$ depicted in the above diagram should be interpreted extensionally. Through the use of type-classes we have overloaded the standard notion of equality in Coq to be the extensional equality, and moreover, using equality with respect to custom notions of the equality that might be defined on the types involved (setoids). See also Section 5.4.1.

If we choose the empty type (i.e. the type *False*) for X, then we obtain the set of closed terms (as in Section 5.2). In that case, the left part of the diagram can be ignored, and the remaining square says precisely that $app : \Sigma\ (T\ False) \to T\ False$ is the initial algebra for functor Σ. Observe that the diagram at the end of Section 5.2 can then be obtained by taking Z_B for Y, and *den* for h.

Finally, it is straightforward to provide a function mapping for T, to turn it into a functor:

Instance : $SFmap\ T :=$
  $\lambda\ X\ Y\ (f : X \to Y), fold\ (var\ Y \circ f)\ (app\ Y)$

5.3.2 Distributive laws

Before we treat the GSOS, we will first consider simple distributive laws, ones that distribute a functor over another functor. Distributive laws (in the simple format) are functions $\Lambda : \Sigma \circ B \Rightarrow B \circ \Sigma$ (i.e. of type $\forall X, \Sigma\ (B\ X) \to B\ (\Sigma\ X)$) that happen to be natural transformations (see Section 5.4.1).

We will replace the operational as well as the denotational model introduced in the previous section with models that are derived from the same distributive law Λ, which corresponds to the operational rules.

```
Definition Λ : Σ ∘ B ⇒ B ∘ Σ :=
  λ X σ,
    match σ with
    | AS ⇒ (a, AS)
    | BS ⇒ (b, BS)
    | Alt x y ⇒
      let (l, x') := x  in
      let (m, y') := y in (l, Alt y' x')
    end
```

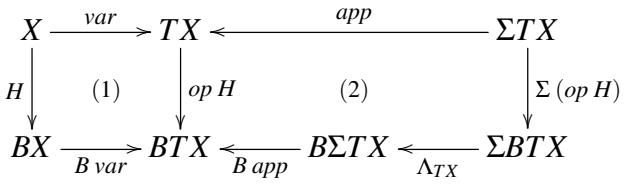
As a function, Λ takes an operation (an element from the signature) as argument. In the case of *Alt* this operation is applied to two arguments that both consist of a pairing of an action and a variable. This corresponds to the premise of a rule. The result is a pairing of an action and an operation applied to variables, corresponding to the target of the conclusion of each rule. The polymorphism in X ensures that Λ does not depend on a concrete choice of the set of variables. In summary, the type of Λ says that each operation in the language, as it is applied to behaviors on the variables, yields a behavior on an operation applied to variables.

5.3.3 Operational and denotational models

In the standard relational approach to operational semantics, the validity of a transition step is proved by the construction of a derivation tree. The nodes correspond to applications of the operational rules, and the leafs correspond applications of the hypotheses.

We can mimic this with the help of the definition principle for terms (the *fold* operation) combined with the semantic model. Suppose that we have a map $H : X \to B\ X$, representing the *behavior environment*: the hypotheses about the variables in the premises. If we encounter an application of an operation, then we

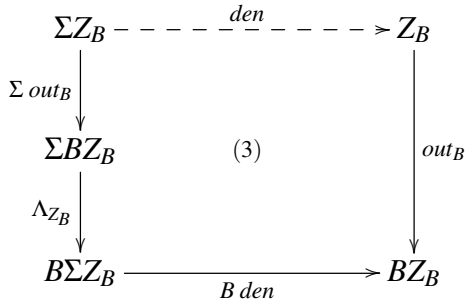
apply Λ, and if we encounter a variable we apply H. In a diagram,



which concretely is

Definition *op* '$(H : X \to B\,X) : T\,X \to B\,(T\,X) :=$
  $fold\ (sfmap_B\ (var\ X) \circ H)$
    $(sfmap_B\ (app\ X) \circ \Lambda\ (T\,X))$

The denotational model can be obtained in a dual fashion, by unfolding the semantic model. Assume the existence of a final coalgebra for the behavior B with state-space Z_B and structure map out_B.



Concretely,

Definition $den : \Sigma\,Z_B \to Z_B :=$
  $unfold_B\ (\Lambda\,Z_B \circ sfmap_\Sigma\ out_B)$

The denotational model operates directly on elements of the semantic domain. It tells how the operations of the language, applied to denotations, form new denotations. We remark that the hypotheses do not play a role in the denotational model, but will come into play when we construct the evaluation function *eval*. Running a term according to the operational model, and evaluating a term according to the denotational model is defined in same way as in Section 5.2:

Definition $run\ (H : X \to B\,X) : T\,X \to Z_B :=$
  $unfold_B\ (op\ H)$

Definition $eval\ (H : X \to B\,X) : T\,X \to Z_B :=$
  $fold\ (unfold_B\ H)\ den$

The distributivity property of Λ will be needed to prove the adequacy of *run* and *eval*.

There are many sensible rules that do not fit in the format of Λ of this section. For example, consider the operation *Zip* of our simple stream language, which zips two input streams together (see Figure 5.1). This operation differs slightly from *Alt* in the sense that at each transition it does not discard the head element of the second stream. If we try to encode this rule in our semantic model, for instance by adding the following alternative to Λ:

$$\mid \mathit{Zip}\ x\ y \Rightarrow \mathbf{let}\ (l,x') := x\ \mathbf{in}\ (l, \mathit{Zip}\ y\ x')$$

then the model does not type check anymore. The main problem is that variables used on both the left- and right-hand sides should receive a polymorphic type, which is not the case for the variable y. Also replacing y by a pattern match, as in the case for *Alt* will not work because then we have to reconstruct the first argument of *Zip* on the right-hand side out of the constituents. In Section 5.5 we discuss the more liberal *GSOS* rule format admitting operations like *Zip*.

5.4 COQ Formalization

In this section we discuss some details of the COQ formalization, and continue the development of the theory for terms.

5.4.1 Equational reasoning with setoids

Infinite objects, as in most theorem provers, live in a world separate from finite objects, and do not adhere to COQ's standard notion of equality. One often works instead with bisimulations, a weaker notion of equality on infinite objects. COQ does not support user-defined extensions of its standard notion of equality (i.e. quotient types) as it would endanger the decidability of type checking. To overcome this issue, it is common practice to work with *setoids*, **Type**s packaged with a user-defined notion of equality and a proof of well-behavedness of the equality. The commuting diagrams in Section 5.2 use bisimulation as the underlying notion of equality in the COQ formalization. Finally, *setoid morphisms* are functions whose domain and codomain are setoids and respect those equalities.

The recent addition of type-classes to COQ [SO08] enables the use of canonical names for standard mathematical notation. These type-classes are first-class as they are powered by proof search and implicit arguments. Declaring instances of the *Equiv*, *Setoid* and *Setoid_Morphism* type-classes enables fluent rewriting modulo setoid equality in proofs. In our development we have tacitly overloaded the canonical name "=" with setoid equality.

First, we introduce setoid counterparts for the standard categorical notions of functor and natural transformation. The *setoid functor* is taken from the MATHCLASSES library [SvdW11]. It consists of an object map M and two classes: a class *SFmap*, which is the function map, and a class *SFunctor* carrying proofs of the setoid functor axioms.[4]

Class *SFunctor* $(M : \mathbf{Type} \to \mathbf{Type})$
$\quad `\{\forall\ `\{Equiv\ X\}, Equiv\ (M\ X)\}\ `\{SFmap\ M\} := \{\ldots\}$

For the full definition of *SFunctor* we refer to the COQ code. The second argument lifts a notion of equality on X to a notion of equality on $M\ X$. Furthermore, it carries two sanity properties stating that the object map makes a setoid on X into a setoid on $M\ X$, and that the function map is a setoid morphism in its function argument (allowing us to rewrite equivalent functions with one another), and the following two familiar properties about the function map:

$$sfmap_M\ id = id$$
$$sfmap_M\ (f \circ g) = sfmap_M\ f \circ sfmap_M\ g$$

Given two object maps M and N, one can define a family of functions:

Notation $M \Rightarrow N := \forall\ X, M\ X \to N\ X$

The family of functions $\eta : M \Rightarrow N$ is a *setoid natural transformation* if η_X is a setoid morphism whenever X is a setoid, and if it satisfies a commutation law:

$$\eta_Y \circ sfmap_M\ f = sfmap_N\ f \circ \eta_X \tag{5.1}$$

Again for reasons of brevity, we no not include the corresponding type-class definition *SNatural*.

5.4.2 Dependent types for generic terms

Attempting to encode terms as the least fixpoint μ of a signature functor as is done in the HASKELL code in Section 5.2 results in an error in COQ, as such definitions violate COQ's syntactic check for positivity, which guarantees termination of structurally recursive functions.

We bypass this issue by exploiting the fact that signatures of binding-free languages have a fairly simple structure. That is, a term on such a signature is essentially a rose tree, in which each parent node has an arbitrary number of child

[4] Unlike HASKELL, COQ admits variable names starting with an uppercase letter. Furthermore, the backtick causes COQ to automatically generalize missing variables.

nodes, dictated by the arity of the operation corresponding to the parent node. A leaf of the tree is either a parent node with zero children nodes, or a variable, if we consider open terms.

The signature is nothing more than an assignment of an arity to each of the language's operations.

Variable $Operation : \mathbf{Type}$

Variable $ar : Operation \to nat$

Definition $\Sigma\, X := \{x : Operation \,\&\, vector\, X\ (ar\, x)\}$

A parent node in the tree is described by a dependent pair, consisting of the operation x and a vector of length the arity of x (*vector* is essentially a richly typed version of *list*). One can think of the notation "$\{\_ \,\&\, \_\}$" as a type-theoretic variant of set comprehension. Dependent pairs can be crafted using the notation "$(\_ \,\&\, \_)$", and *projT1*, *projT2* are the corresponding projections.

The function map for the signature functor is:

Instance : $SFmap\ \Sigma :=$
$\lambda\, X\ Y\ (f : X \to Y)\ (\sigma : \Sigma\, X)$,
match σ **with**
$|\ (s \,\&\, v) \Rightarrow (s \,\&\, map\, f\ v)$
end

The induction principle for T pushes the use of dependent types even further. It is the basis for the proofs of the Lemmas in Section 5.4.3.

Definition $T\_induction\ '(P : T\ X \to \mathbf{Type}) :$
$(\forall x : X, P\ (var\, x)) \to$
$(\forall x : \Sigma\ \{t : T\ X \,\&\, P\ t\}, P\ (app\ (sfmap_{\Sigma}\ projT1\ x))) \to \forall t, P\ t$

$T\_induction$ is essentially a dependently-typed version of *fold*: we can re-obtain *fold* by setting $P := \lambda\ \_, Y$:

Definition $fold\ '(k : X \to Y)\ (h : \Sigma\ Y \to Y) : T\ X \to Y :=$
$T\_induction\ (\lambda\ \_, Y)\ k\ (h \circ (sfmap_{\Sigma}\ projT2))$

5.4.3 Theory about terms

Now that we have the full definitions in place, we can continue our formalized treatment of the theory about terms.

Lemma 3. *Σ and T are setoid functors.*

The proof of Lemma 3 uses the full dependently-typed induction principle for T. The full principle has also been used to prove the properties in this section by induction on the structure of T.

In the remainder of this section we show that T is a monad in the categorical sense. To this end, we need to show that it has a unit, *var* in this case, and a multiplication, namely:

Definition $join\ X : T\ (T\ X) \to T\ X := fold\ id\ (app\ X)$

These satisfy the two standard coherence conditions of monads.

Lemma 4. *The terms form a monad, i.e. the following identities hold:*

$$join_X \circ sfmap_T\ join_X = join_X \circ join_{TX}$$
$$join_X \circ sfmap_T\ var_X = join_X \circ var_{TX} = id$$

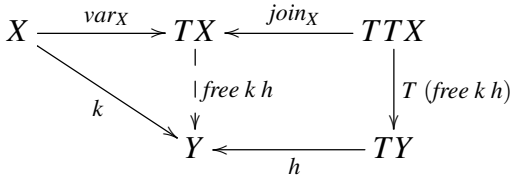
It is a well-known fact from category theory that the category of Σ-algebras is isomorphic to the category of *algebras for the term monad*. These are "plain" algebras h for the functor T, with two additional properties:

$$h \circ var_Y = id$$
$$h \circ sfmap_T\ h = h \circ join_Y$$

A T-algebra homomorphism is a homomorphism of the underlying algebra.

We conclude this section by providing an alternative proof principle for open terms, that differs from Lemma 2.

Lemma 5. *Let $k : X \to Y$ and $h : T\ Y \to Y$ such that h is an algebra for the term monad. Set free k h := fold k $(h \circ app_Y \circ sfmap_\Sigma\ (var_Y))$. Then free k h is unique in making the following diagram commute:*



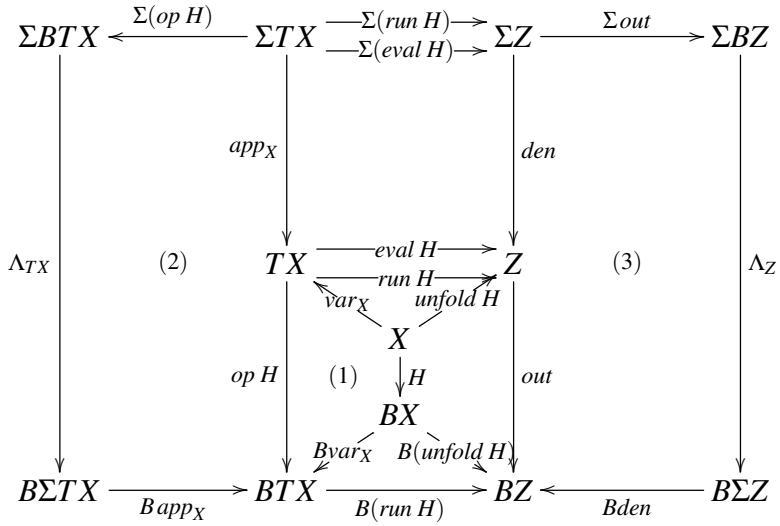
We have now set up a theory for syntax. Similarly, we could develop a theory for behavior. We do not pursue this goal for two reasons. First, since our presentation is essentially a deep embedding of SOS rules, most of it hinges on a structured encoding of the terms. Secondly, one may expect that more variation in the behavior is desired to model phenomena such as time or probability (see [Bar04]; see also [Kli11] for an overview). Moreover, finality proofs can be tricky to carry out in COQ due to guardedness restrictions that it puts on corecursive definitions.

5.5 Proving the Adequacy Theorem

Our approach to prove the adequacy theorem for the GSOS rules is to first carry it out for the simple rule format. The proof for GSOS follows the same proof skeleton, but requires some additional intermediate verifications.

5.5.1 Adequacy theorem for rules in simple format

Recall how the operational and denotational models are obtained from a single semantic model. We combine the diagrams of the operational and denotational models into the following diagram.

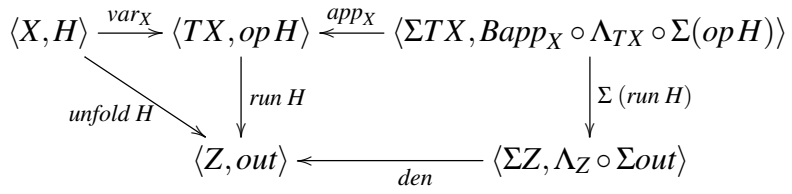


The following theorem holds for *open* terms, which is a mild generalization of what has been presented in the literature [TP97, Kli11, Bar04].

Theorem 1 (Adequacy)**.**

$$\forall H\, t, run\ H\ t = eval\ H\ t.$$

Proof. Consider the following diagram in the category of B-coalgebras. That is, the objects are pairs (consisting of the object and the structure map) and the arrows are coalgebra homomorphisms.



$$\frac{}{!a \xrightarrow{\mathsf{a}} \mathit{Done}} \qquad \frac{x \not\xrightarrow{} \qquad y \xrightarrow{l} y'}{x \bullet y \xrightarrow{l} y'} \qquad \frac{x \xrightarrow{l} x'}{x \bullet y \xrightarrow{l} x' \bullet y}$$

$$\frac{x \xrightarrow{l} x'}{x \sqcup y \xrightarrow{l} x'} \qquad \frac{y \xrightarrow{l} y'}{x \sqcup y \xrightarrow{l} y'} \qquad \frac{}{\mathit{Done} \not\xrightarrow{}}$$

Figure 5.2: Basic process algebra.

Except for the arrow $\Sigma\ (\mathit{run}\ H)$, it is trivial to see that each of the arrows are in fact coalgebra homomorphisms, as it can be directly read off the complete diagram above. To show this for $\Sigma\ (\mathit{run}\ H)$, one uses naturality of Λ and the fact that *run* is a coalgebra homomorphism. Commutativity of the diagram follows from the finality of *out*. The theorem then follows from applying the forgetful functor to the above diagram and the definition principle used to define *eval H*. □

5.5.2 The GSOS format

Consider the process algebra language of Figure 5.2. The parallel composition operation "⊔" and the possibility of a state not having any outgoing transitions clearly make this a non-deterministic language. We use the *fin* type family to define the behavior functor; *fin n* consists of the first *n* natural numbers.

Definition $B\ X := \{\, n : \mathit{nat}\ \&\ \mathit{fin}\ n \to A \times X \,\}$
CoInductive $\mathit{cotree} := \mathit{node} : B\ \mathit{cotree} \to \mathit{cotree}$

The object of the corresponding final coalgebra consists of the cotrees.

Lemma 6. *cotree is the object of the final coalgebra for the functor B.*

As an aside, it may seem more natural to set $B\ X := \mathit{list}\ (A \times X)$, but one runs into problems trying to provide a guarded corecursive definition of *unfold* for that choice of *B*.

Note that the sequencing operation "•" does not fit in the simple rule format for the same reasons as *Zip* in Section 5.3.3. Moreover, the second rule in Figure 5.2 has a transition to a variable in its conclusion, which is not supported in the simple format.

The operational rules of basic process algebra can however be encoded in the GSOS format. Let *B* be arbitrary and Σ be constrained to the conditions set out in

Section 5.4.2, and set $D\,X := X \times B\,X$. The *abstract GSOS format* entails natural transformations of the following type:[5]

$$\rho : \Sigma \circ D \Rightarrow B \circ T$$

The difference with the simple format is that the arguments of each operation are now pairings consisting of both the variable and the behavior on that variable, and each rule yields a term.

For reference we include the encoding of the rules of Figure 5.2:

Definition $\rho : \Sigma \circ D \Rightarrow B \circ T :=$
$\lambda\, X\, \sigma,$
match σ **with**
$\mid !\, a \Rightarrow (1 \,\&\, (\lambda\, \_, (\mathsf{a}, app\ Done)))$
$\mid x \sqcup y \Rightarrow (\_ \,\&\, merge\ ([id, var] \circ projT2\ (\pi_2\ x))$
$\qquad ([id, var] \circ projT2\ (\pi_2\ y)))$
$\mid (\_, (0 \,\&\, \_)) \bullet (\_, b) \Rightarrow sfmap_B\ var\ b$
$\mid (\_, b) \bullet (y, \_) \Rightarrow sfmap_B\ (\lambda\, x', app\ (var\ x' \bullet var\ y))\ b$
$\mid Done \Rightarrow (0 \,\&\, case0\ \_)$
end

Here *merge* and *case0* have the following types:

Definition *merge* $\text{`}(f : fin\ n \to X)\ \text{`}(g : fin\ m \to X) : fin\ (n+m) \to X$
Definition *case0* $(f : fin\ 0 \to \mathbf{Type})\ (i : fin\ 0) : f\ i$

5.5.3 From GSOS to distributive laws

Recall that the symmetry of the (co)domains was vital to the proof of the adequacy theorem. The rules ρ need to undergo a two-step transformation to obtain a distributive law of T over D. First expand ρ's codomain:

Definition $\widetilde{\rho} : \Sigma \circ D \Rightarrow D \circ T :=$
$\lambda\, X, \langle app\ X \circ sfmap_\Sigma\ (var\ X \circ \pi_1), \rho \rangle$

To obtain Λ we apply *fold*.

Definition $\Lambda : T \circ D \Rightarrow D \circ T :=$
$\lambda\, X, fold\ (sfmap_D\ (var\ X))$
$\qquad (sfmap_D\ (join\ X) \circ \widetilde{\rho}\ (T\ X))$

[5] Literature on process algebra often considers just the case where B is the finite power set and calls that the GSOS format.

A general proof (included in the COQ development) shows that the definition principle for terms yields natural transformations. From this fact, together with the assumption that ρ is a natural transformation, it is straightforward to show that Λ is natural as well. The obtained distributive law, which distributes the term monad over the copointed functor D, enjoys more structure than the plain distributive laws, as we will prove in Proposition 1.

Proposition 1. *The following two identities hold:*

$$\Lambda_X \circ var_{DX} = sfmap_D\ var_X$$
$$\Lambda_X \circ join_{DX} = sfmap_D\ join_X \circ \Lambda_{TX} \circ sfmap_T\ \Lambda_X$$

The first identity says that the law should behave trivially on variables. The second identity characterizes compositionality of the semantics. Proposition 1 is a key ingredient in the proof of Lemma 7 and Lemma 8. Lenisa, Power and Watanabe [LPW04] prove that the GSOS format corresponds precisely to a distributive law of a monad over a copointed functor, but for the adequacy theorem it is sufficient to show that a GSOS rule implies such a law.

5.5.4 Adequacy theorem for the GSOS format

The proof of the adequacy theorem for GSOS is analogous to the situation in Theorem 1, but B should be replaced by D, Σ by T, and *app* by *join* and entails more proof obligations due to the richer structure on Λ. We adapt the definitions of the operational/denotational models and semantics to this new situation. Recall that we used the definition principle of terms to obtain an operational model from the plain distributive laws. We repeat this construction for distributive laws that stem from GSOS rules, with the difference that we use the alternative definition principle.

Definition $op_{GSOS}\ `(H : X \to D\ X) : T\ X \to D\ (T\ X) :=$
$\quad free\ (sfmap_D\ (var\ X) \circ H)$
$\qquad (sfmap_D\ (join\ X) \circ \Lambda\ (T\ X))$

As in the proof, we need to verify that each of the relevant arrows are coalgebra homomorphisms, but for the functor D instead of B. For the arrow corresponding to *join* this follows from the following fact:

Lemma 7. *$sfmap_D\ join_X \circ \Lambda_{TX}$, used in op_{GSOS}, is an algebra for the term monad.*

The D-coalgebras are isomorphic to the B-coalgebras, and it is straightforward to verify that if *out* is a final B-coalgebra, then $\langle id, out \rangle$ is a final D-coalgebra (with

the same state-space). Hence, the denotational model for GSOS rules and run_{GSOS} can be obtained by finality, analogous to Section 5.3.2. We obtain $eval_{GSOS}$ by making use of the alternative definition principle for terms.

Definition $run_{GSOS}\ (H : X \to D\ X) : T\ X \to D\ (T\ X) :=$
$unfold_D\ (op_{GSOS}\ H)$
Definition $den_{GSOS} : T\ Z_D \to Z_D :=$
$unfold_D\ (\Lambda\ Z_D \circ sfmap_T\ out_D)$
Definition $eval_{GSOS}\ (H : X \to D\ X) : T\ X \to D\ (T\ X) :=$
$free\ (unfold_D\ H)\ den_{GSOS}$

Recalling Lemma 5, to ensure the uniqueness of $eval_{GSOS}$, we need to verify the following fact:

Lemma 8. *den_{GSOS} is an algebra for the term monad.*

We can now conclude the following:

Theorem 2 (Adequacy for GSOS rules)**.**

$$\forall H\ t, run_{GSOS}\ H\ t = eval_{GSOS}\ H\ t.$$

5.6 Related Work

The work in this chapter is part of a line of research called *bialgebraic semantics*, initiated by the work of Turi and Plotkin [TP97]. Bialgebras appear in the present chapter in the form of diagrams (2) and (3). Hinze and James [HJ11] give a pen and paper proof of the adequacy theorem based on HASKELL definitions for several rule formats, using proof techniques similar to ours. The most powerful rules distribute a monad over a comonad and also appear in [TP97, Bar04]. Although these laws provide the most abstract perspective of well-behaved rules, they have not yet been applied in concrete studies of rule formats [Kli11].

An implementation of Turi and Plotkin's work has been developed by Hutton [Hut98] in HASKELL and extended for modularity by Jaskelioff, Ghani and Hutton [JGH11]. Both papers define the terms and and the final coalgebra as the greatest fixpoint of a functor. Direct translations to COQ are not possible; in this chapter we have presented an alternative approach based on dependent types.

Niqui [Niq09] extends the class of productive specifications definable in COQ by developing the λ-coiteration scheme in COQ, based on Bartels' work [Bar04]. In the further work section of his paper he mentions that adding monadic, pointed or cofree structure on the bialgebraic nature of λ-coiteration can help to build even more powerful schemes.

Aceto et al. [ACGI11] have developed a tool, called the PREG AXIOMATIZER, to prove the bisimilarity of two ground terms written in a language specified in GSOS extended with predicates. It derives a sound set of axioms from the GSOS rules, and uses that to prove the bisimulation.

5.7 Conclusions

We have shown how operational and denotational semantics can be obtained from operational rules in the GSOS format in the theorem prover COQ. Moreover, we have formally proved the theorem that says that these forms of semantics are consistent. Our formalization facilitates both formal reasoning about and the execution of programming language semantics.

Directions of further work would be to add support for variable binding, which requires the use of a different base category [FPT99] (the present formalization is based on **Type**), and further generalization to support different rule formats, as in [Bar04, HJ11].

CHAPTER 6

Modular Bialgebraic Semantics and Algebraic Laws

Abstract. The ability to independently describe operational rules is indispensable for a modular description of programming languages. This chapter introduces a format for open-ended rules and proves that conservatively adding new rules results in well-behaved translations between the models of the operational semantics. Silent transitions in our operational model are truly unobservable, which enables one to prove the validity of algebraic laws between programs. We also show that algebraic laws are preserved by extensions of the language and that they can be instantiated. The work presented in this chapter is developed within the framework of bialgebraic semantics.

6.1 Introduction

In order to scale to the complexity of real-world programming languages, a modular way of describing semantics is highly desirable. When dealing with incrementally constructed languages, one should anticipate future extensions or changes to the language. Moreover, a concrete program seldomly uses all the constructs provided by the language. When reasoning about a program it is convenient to narrow the semantics down to the part of the language which is actually used. True modularity offers the possibility to build an ad hoc semantics, easing the construction of correctness proofs.

Mosses [Mos09] advocates to define higher-level language constructs out of so-called "funcons", language-independent fundamental programming constructs. It is highly desirable that algebraic equations between programs are preserved under the addition of new funcons, since this avoids the repetition of proofs.

The present chapter provides a fundamental perspective on this issue, built on the framework of Turi and Plotkin's bialgebraic semantics [TP97] (see also Chapter 5). One of the advantages of this work is that it can be implemented in a functional language such as HASKELL as well as in a theorem prover like COQ. In fact, part of this work has already been formalized within COQ, based on [MS13].

Each operation corresponding to a funcon has a number of defining operational rules, which may manipulate the state, or invoke an external operation. For example, the rule for a condition-less loop would be $\mathsf{loop}\ x \Longrightarrow \mathsf{seq}\ x\ (\mathsf{loop}\ x)$. The double arrow indicates that the transition is deemed silent, it does not generate an observable side-effect. To handle two subsequential commands, loop invokes the external operation seq. This mechanism is comparable to interfaces in object-oriented languages. Thus, we consider the operational rules corresponding to some construct as open-ended, empowering true modularity in language descriptions. By commencing with an empty language and then incrementally extending this with new constructs, a full language is obtained.

Silent transitions are indispensable in providing independent descriptions of the operations. An alternative version of the above silent transition loop-rule, which avoids the use of a silent transition, can be defined by performing a "look-ahead", i.e.:

$$\begin{array}{lll} x \xrightarrow{a} x' & \vdash & \mathsf{seq}\ x\ y \xrightarrow{a} \mathsf{seq}\ x'\ y \\ x \xrightarrow{a} x' & \vdash & \mathsf{loop}\ x \xrightarrow{a} \mathsf{seq}\ x'\ (\mathsf{loop}\ x). \end{array}$$

However, a change in the rule for seq would impact the rule for loop and vice versa, e.g.:

$$\begin{array}{lll} x \xrightarrow{a} x' & \vdash & \mathsf{seq}\ x\ y \xrightarrow{\overline{a}} \mathsf{seq}\ x'\ y \\ x \xrightarrow{a} x' & \vdash & \mathsf{loop}\ x \xrightarrow{\overline{a}} \mathsf{seq}\ x'\ (\mathsf{loop}\ x), \end{array}$$

hence these rules depend on each other; this is not a modular semantics. Another way in which the modularity problem of the look-ahead version of loop manifests itself is that it requires a separate rule in the case that x is skip, because skip can make no transition at all.

A distinguished label, '$\bot$' say, could be used to mimic silent transitions, but replacing silent transitions by $\bot$-transitions does not make them truly unobservable. For example, $\mathsf{loop}\ x$, with $\bot$-transitions, would not be behaviorally equivalent to $\mathsf{seq}\ x\ (\mathsf{loop}\ x)$, unless one resorts to the more complex notion of weak bisimulations. Finally, rules for silent transitions are often not purely structural.

The rule seq skip $x \Longrightarrow x$ inspects the first argument of the head operation before it can be applied.

In this chapter we treat structural operational rules and rules for silent transitions as separate classes. A generalization of the categorical interpretation by Turi and Plotkin [TP97] of the GSOS rule format accommodates the structural rules. We apply an altered construction of Klin [Kli04], who extends Turi and Plotkin's work with silent transitions.

The standard notion of bisimulation between computations expresses that both computations exhibit the same observational behavior. Unfortunately, standard bisimulation is not preserved by language extensions [MMR10]. Formal-Hypothesis bisimulations, introduced by De Simone [DS85], take into account that variables in terms being evaluated may exhibit arbitrary behavior. A pair of FH-bisimilar (open) terms is called an algebraic law. We prove that our notion of language extension preserves algebraic laws. Moreover, we show that algebraic laws can be instantiated, in the sense of [Ren00]. This property eases reasoning about programs, since it allows program fragments to be replaced by other simpler fragments, provided these are FH-bisimilar.

In summary, the contributions of this chapter are threefold:

- We introduce a rule format, called "open GSOS", which enables the modular description of operational semantics. Moreover, we provide a definition for conservative extensions of open GSOS rules, and show that there exists a well-behaved translation between the operational semantics described by open GSOS rules and the operational semantics described by conservative extensions of these rules.

- We add support for rules with silent transitions to open GSOS, in such a manner that silent transitions are truly unobservable while well-behavedness of the translation between the base and extended operational semantics remains intact.

- We formalize the notion of algebraic laws within the bialgebraic framework, prove that these laws are preserved through conservative language extensions, and prove that they can be instantiated. This transfers some results from [MMR10] and [Ren00] to the setting of bialgebraic semantics.

Basic definitions are provided in Section 6.2, followed by three sections (Section 6.3, 6.4, and 6.5) corresponding to the above points. Section 6.6 shows how the resulting operational models can be executed. Related work is discussed in Section 6.7 and conclusions are drawn in Section 6.8.

6.2 Preliminaries

This section recalls some basic definitions. A good introduction to the field of bialgebraic semantics is provided in [Kli11]; further background can be found in [Jac12].

We shall use the symbols $X, Y, \ldots$ and $f, g, \ldots$ to respectively denote objects and morphisms in the underlying category, and Greek symbols $\psi, \phi, \ldots : F \Rightarrow G$ to denote natural transformations between functors F and G. Because the work in this chapter is intended to be verifiable in Coq, we use the category of Coq types (in which the morphisms are Coq functions) as the underlying category. However, the work in this chapter is general enough for it to be applicable to other categories (such as **Set**). The Coq-like notation $x, y : X$ means that x and y are instances of type X.

The (open) terms, generated by an endofunctor F, where X acts as the variables, are the initial solution F^*X to the equation $Y \cong X + FY$. This means that there is an isomorphism $\kappa_X : F^*X \xrightarrow{\cong} X + FF^*X$, whose inverse is

$$\kappa_X^{-1} = [\eta_X, \psi_X] : X + FF^*X \xrightarrow{\cong} F^*X,$$

i.e. formally $\eta_X := \kappa_X^{-1} \circ \kappa_1$ and $\psi_X := \kappa_X^{-1} \circ \kappa_2$, where κ_1 and κ_2 stand for the left and right injections, respectively, into the coproduct. Thus, terms are either a variable (and η_X transforms a variable into a term), or an operation over a number of terms (and ψ_X transforms that into a term). We will also use an auxiliary morphism to transform operations into terms:

$$\phi : F \Rightarrow F^* := \psi \circ F\eta$$

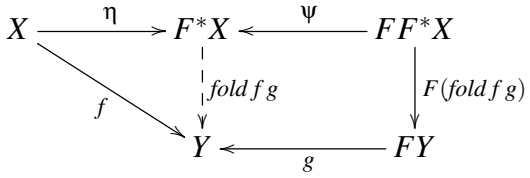
The functor F, called the *signature functor*, stands for the grammar of the language, and is specified by cases, e.g. $FX := X^2 + X$, which we will also denote as $FX := \mathsf{seq}\ (x\ y : X) \mid \mathsf{loop}\ (x : X)$, to directly label each of the cases in the coproduct. To enhance readability, we will loosely use square brackets $[-]_F$ to indicate that a term should be interpreted as an instance of F^*X. For example, by $[\mathsf{seq}\ x\ y]_F$ we formally mean $\phi_X\,(\mathsf{seq}\ (\eta_X\, x)\ (\eta_X\, y))$.

One can show that the functor F^* is a monad. Recall from Chapter 5 that this means that F^* has a unit, $\eta : Id \Rightarrow F^*$, and a join operation $\mu : F^*F^* \Rightarrow F^*$ (which flattens terms over terms), which are both natural transformations (see Chapter 5 for their definitions Coq), and that η and μ are subject to the following conditions:

$$\mu \circ F^*\mu = \mu \circ \mu_{F^*} \tag{6.1}$$

$$\mu \circ \eta_{F^*} = \mu \circ F^*\eta = id. \tag{6.2}$$

Formally, the terms F^*X are the free monad generated by F, and come with a principle which says that there is a unique morphism satisfying the following diagram, for any morphism $f : X \to Y$ and algebra $g : FY \to Y$:

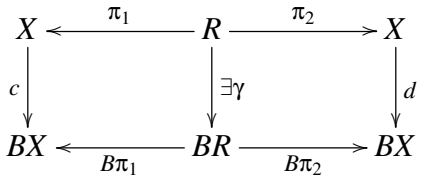


(6.3)

We have called this unique morphism *fold f g*, to emphasize that this corresponds to folding over terms, a concept familiar from functional programming (see Chapter 5 for examples in COQ).

F-algebras are morphisms $FX \to X$ for some X and functor F. F-algebras such as ψ_X play a crucial role in the syntax. Dually coalgebras play a crucial role for the behavior. A *B-coalgebra* is a pair $\langle X, c \rangle$, where X is an object, representing the states of the system and $c : X \to BX$ is a morphism capturing the dynamics of the system. B is the type functor, also called the *behavior functor*. Sometimes we will also call c itself a coalgebra, if X is understood from the context.

A *relation* is a triple $\langle R, \pi_1 : R \to X, \pi_2 : R \to X \rangle$, where R and X are objects in the underlying category (recall that in the present chapter we consider this to be the category of COQ types). The notation xRy for $x, y : X$ means that there exists an instance $r : R$ such that $\pi_1 r = x$ and $\pi_2 r = y$. The relation $\langle R, \pi_1, \pi_2 \rangle$ is an *Aczel-Mendler bisimulation relation* [AM89, Jac12] between two coalgebras c, d if there exists a morphism γ such that the following diagram commutes:



6.3 Rule Format

In [TP97] it was shown that the operational rules in the GSOS format can be understood as a natural transformation $\rho : FD \Rightarrow BF^*$, where $D := Id \times B$. From Chapter 5 we recall that the left-hand side of the arrow in the type of ρ represents an operation (from the signature functor), which for each of its argument positions takes a pair consisting of a variable and the behavior of that variable. The right-hand side is the result of executing the rule on that operation, which is a behavior and the resulting term. From ρ one can derive subsequently a distributive law Λ^ρ :

$$x \xrightarrow{a} x' \;\vdash\; \mathsf{seq}\ x\ y \xrightarrow{a} \mathsf{seq}\ x'\ y$$

$$\mathsf{seq}\ \mathsf{skip}\ y \Longrightarrow y$$

$$\mathsf{seq}\ x\ \mathsf{skip} \Longrightarrow x$$

$$\mathsf{loop}\ x \Longrightarrow \mathsf{seq}\ x\ (\mathsf{loop}\ x)$$

$$x \xrightarrow{a} x' \;\vdash\; \mathsf{if}\ x\ y\ z \xrightarrow{a} \mathsf{if}\ x'\ y\ z$$

$$\mathsf{if}\ \mathsf{true}\ y\ z \Longrightarrow y$$

$$\mathsf{if}\ \mathsf{false}\ y\ z \Longrightarrow z$$

Figure 6.1: Example operational rules.

$F^*D \Rightarrow DF^*$ and a coalgebra $op_{\Lambda\rho} : F^*X \to BF^*X$, which acts as the operational model.

In order to develop programming language semantics in an incremental fashion, it is useful to be able to define the operations used in the language in isolation, so that an existing language can be extended with new operations. This also promotes reuse of pre-defined operations. In order to provide a set of rules in the GSOS format, according to the above type, one needs to define each of the operations in the language. Instead, in this section, we provide a generalization of the GSOS format, which we call "Open GSOS", enabling independent description of the semantics of (groups of) operations, which can later be combined in order to obtain the full language.

6.3.1 Example

As an example, we will explain, in terms of bialgebraics, the semantics of a very basic programming language, which consists of an if-then-else operation and an operation to sequence statements. The rules can be found in Figure 6.1 (the reader should ignore the loop rule for now). The double arrows '$\Longrightarrow$' stand for silent transitions, and we will ignore these rules until Section 6.4, as they require special treatment.

We consider labels to be a simplified version of MSOS of Chapter 4. Suppose that A is a set consisting of labels and set $BX := A \times X$. More specifically, A could represent all possible state transitions $S \times S$, for some set of states S. The operations in the rules Figure 6.1 are if and seq. We do not consider skip, true and false to be operations, instead, we consider these to be *values*, as they produce no behavioral meaning (see also [CM13]). When values occur in argument positions of operations, silent transitions may be triggered, as it is witnessed by the silent transition rules of seq and if. We will discuss silent transitions in more depth in Section 6.4. When an operation only has variables on its argument positions (ruling out e.g. $\mathsf{seq}\,\mathsf{skip}\,y$), a non-silent transition is made, which can be specified in terms of GSOS rules $\rho : FD \Rightarrow BF^*$.

To provide a modular semantics, we define the rules for each operation seperately, starting with seq. We define a signature functor $FX := \mathsf{seq}\ (xy : X) = X^2$, and then define $(\rho_F)_X$ (for any object X) as follows (recall that F^* is the free monad generated by F):

$$\begin{aligned}(\rho_F)_X : FDX &\longrightarrow BF^*X \\ \mathsf{seq}\ \langle x, \langle a, x' \rangle\rangle\ \langle y, \langle b, y' \rangle\rangle &\longmapsto \langle a, [\mathsf{seq}\ x'\ y]_F \rangle\end{aligned}$$

We can also define the non-silent rule for if in a similar way, for the signature functor $GX := \mathsf{if}\ (xyz : X) = X^3$, yielding a rule ρ_G:

$$\begin{aligned}(\rho_G)_X : GDX &\longrightarrow BG^*X \\ \mathsf{if}\ \langle x, \langle a, x' \rangle\rangle\ \langle y, \langle b, y' \rangle\rangle\ \langle z, \langle c, z' \rangle\rangle &\longmapsto \langle a, [\mathsf{if}\ x'\ y\ z]_G \rangle\end{aligned}$$

Starting with a language just consisting of seq, we can extend it with the if operation by taking the coproduct of the rules as follows, yielding:

$$\begin{aligned}(\rho_{F+G})_X : (F+G)DX &\longrightarrow B(F+G)^*X \\ \kappa_1\,(\mathsf{seq}\ x\ y) &\longmapsto B\,(\iota_{F^*,(F+G)^*})_X\,((\rho_F)_X\,(\mathsf{seq}\ x\ y)) \\ \kappa_2\,(\mathsf{if}\ x\ y\ z) &\longmapsto B\,(\iota_{G^*,(F+G^*)})_X\,((\rho_G)_X\,(\mathsf{if}\ x\ y\ z))\end{aligned}$$

In the specification above, κ_1, κ_2 stand for the left and right injections, respectively, into the coproduct, and $\iota_{F^*,(F+G)^*}$ and $\iota_{G^*,(F+G)^*}$ stand for the "inclusion" of the terms generated by signature functors F, G respectively, into the terms generated by the coproduct $F+G$. We will discuss this inclusion natural transformation in more depth in the next section.

The above construction with coproducts can be used when each of the operations, which are defined in separate rules, do not refer to operations outside their range of definition. If we wish to define a looping operation (i.e. an operation that causes its argument to be executed indefinitely), using the rule

$$x \xrightarrow{a} x' \vdash \mathsf{loop}\ x \xrightarrow{a} \mathsf{seq}\ x'\ (\mathsf{loop}\ x),$$

then we run into a problem. Say the corresponding bialgebraic GSOS rule would be $\rho_H : HD \Rightarrow BH^*$, where $HX := \mathsf{loop}\ (x : X) = X$. In that case the rule has a type-mismatch, because seq is not included in H, although the rule returns a term which has seq in it. On the other hand, if seq were to be included in H, then there is a redundant specification of seq in ρ_H and ρ_F, which violates modularity. This situation arises often in modular language design, as operations tend to interact with one another. As we have explained in the introduction of this chapter, the way the looping operation is defined above is not modular, we merely use the

above as an example to illustrate a limitation of the standard type $FD \Rightarrow BF^*$ of the GSOS format.

6.3.2 The Open GSOS format

We introduce a generalization of the GSOS rule format which makes a distinction between the signature whose operations are being defined, the *ingoing signature functor* F, and the signature whose operations can be the result of the rule, the *outgoing signature functor* G. Recall that K^* denotes the free monad generated by some functor K. When F is a signature functor, then F^*X essentially represents the terms generated by that signature, with variables from X.

Definition 1 (Open GSOS Rules)**.** *Suppose that we have functors F, G, B, such that there exists a natural transformation $\iota_{F,G} : F \Rightarrow G$, called the* inclusion of F in G*. Furthermore, suppose that G^* is the free monad generated by G. A rule in* open GSOS format *is a natural transformation $\rho : FD \Rightarrow BG^*$, where $D := Id \times B$.*

It is possible to slightly generalize the above definition by stipulating that D is any functor that comes with a natural transformation $\pi : D \Rightarrow Id$. The pair $\langle D, \pi \rangle$ is then also called a *copointed endofunctor*. However, most of the results in this chapter depend on the assumption that $D := Id \times B$ specifically.

As an example, the ingoing signature of the loop operation would be $FX := \mathsf{loop}\,(x : X)$, and its outgoing signature would be $GX := \mathsf{loop}\,(x : X) \mid \mathsf{seq}\ (xy : X)$ (ignoring previous use of the names F, G).

The intuition behind the natural transformation $\iota_{F,G}$ is that it corresponds to the set inclusion of the operations (function symbols) corresponding to each of the signatures, and thus any operation in the ingoing signature should be part of the outgoing signature. Although in most of the examples the inclusion functor will be pointwise injective, we do not formally require it to be so in this chapter. We can extend $\iota_{F,G}$ to terms, by folding:

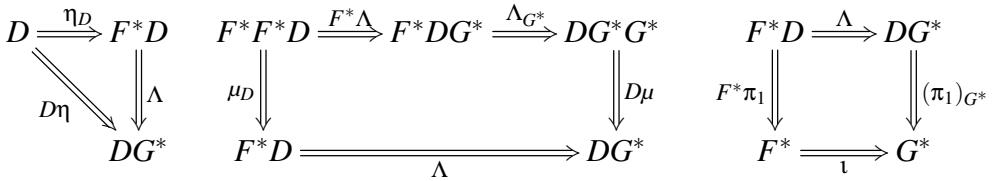
$$(\iota_{F^*,G^*})_X : F^*X \to G^*X := \mathit{fold}\ (\eta_G)_X\ ((\psi_G)_X \circ \iota_{G^*X}),$$

which yields a monad morphism $\iota_{F^*,G^*} : F^* \Rightarrow G^*$ (verifying that this is indeed a monad morphism is left as an exercise for the reader). Likewise we have an inclusion for the behavior functors and the obvious extension to the copointed behavior functors. When the types are obvious, we will omit the subscripts belonging to ι (and likewise for η, μ and ψ).

The idea is that one defines rules where the outgoing signature functor has the minimal set of operations required to define the rule (e.g. for the seq operation the minimal outgoing signature functor is $FX := \mathsf{seq}\ (xy : X)$, and $FX := \mathsf{seq}\ (xy : X) \mid \mathsf{if}\ (xyz : X)$ is not minimal). Then the rule can be included into the full

language by composing it with the inclusion functor (as in the ρ_{F+G} example of the previous section), which is in fact also an Open GSOS rule. The definition of Open GSOS does not prevent one from not using the minimal set of operations in the outgoing signature, however, using it that way is unnecessary and non-modular. Repeatedly extending existing Open GSOS rules with new operations (by taking coproducts as in the ρ_{F+G} example) results in new rules in the Open GSOS format. Eventually, one should end up with a closed system where the ingoing and outgoing signature functors are equal, in which case it is in the original GSOS format.

Definition 2. *Suppose that there exists a natural transformation $\iota : F \Rightarrow G$ for signature functors F, G. An* open distributive law of F^*, G^* over the functor $D := Id \times B$ *is a natural transformation $\Lambda : F^*D \Rightarrow DG^*$, subject to the following three coherence conditions:*



From left to right, the first condition says that the law should behave trivially on variables, the second condition characterizes the compositionality of the semantics, and the third condition says that the first component of the result is essentially the input, included into G^*.

Proposition 2. *There exists a map $\rho \mapsto \Lambda^\rho$, which is a one-to-one correspondence between natural transformations $\rho : FD \Rightarrow BG^*$ and open distributive laws $\Lambda^\rho : F^*D \Rightarrow DG^*$.*

Proof. First, define the auxiliary morphisms

$$\widetilde{\Psi}\ \frac{}{FD \overset{F(\eta\circ\pi_1)}{\Longrightarrow} FF^* \overset{\Psi}{\Rightarrow} F^*}$$

and

$$\widetilde{\rho}\ \frac{FD \overset{\rho}{\Rightarrow} BG^*}{FD \overset{\langle\widetilde{\Psi},\rho\rangle}{\Longrightarrow} F^* \times BG^* \overset{\iota_{F^*,G^*}\times id}{\Longrightarrow} DG^*}.$$

From an open GSOS rule ρ_X we obtain a morphism, which we will later prove to be an open distributive law, by folding:

$$\Lambda^\rho_X : F^*DX \to DG^*X := fold\ (D(\eta_G)_X)\ (D(\mu_G)_X \circ \widetilde{\rho}_{G^*X}).$$

We can show that $B(\mu_G)_X \circ (\pi_2)_{G^*X} \circ \Lambda^{\rho}_X \circ \phi_{DX} \circ FD(\eta_G)_X$ is the inverse of the above. First, we calculate:

$$\begin{aligned}
\Lambda^{\rho}_X \circ \phi_{DX} &= \Lambda^{\rho}_X \circ (\psi_F)_{DX} \circ F(\eta_F)_{DX} \\
&= D(\mu_G)_X \circ \widetilde{\rho}_{G^*X} \circ F\Lambda^{\rho}_X \circ F(\eta_F)_{DX} \\
&= D(\mu_G)_X \circ \widetilde{\rho}_{G^*X} \circ F(\Lambda^{\rho}_X \circ (\eta_F)_{DX}) \\
&= D(\mu_G)_X \circ \widetilde{\rho}_{G^*X} \circ FD(\eta_G)_X \\
&= D(\mu_G)_X \circ DG^*(\eta_G)_X \circ \widetilde{\rho}_{G^*X} \\
&= D((\mu_G)_X \circ G^*(\eta_G)_X) \circ \widetilde{\rho}_{G^*X} \\
&= D(id) \circ \widetilde{\rho}_{G^*X} \\
&= \widetilde{\rho}_{G^*X}.
\end{aligned}$$

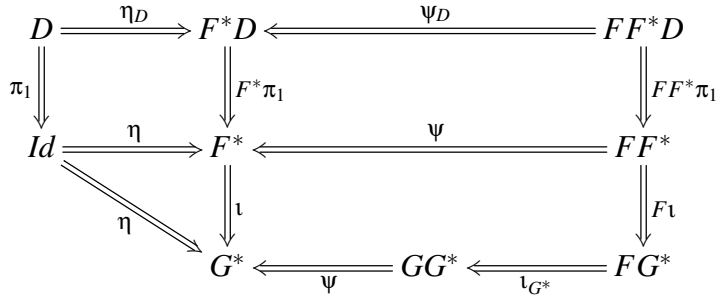
This leaves us with:

$$\begin{aligned}
B(\mu_G)_X \circ (\pi_2)_{G^*X} \circ \widetilde{\rho}_{G^*X} \circ FD(\eta_G)_X &= B(\mu_G)_X \circ \rho_{G^*X} \circ FD(\eta_G)_X \\
&= B(\mu_G)_X \circ BG(\eta_G)_X \circ \rho_X \\
&= B((\mu_G)_X \circ (\eta_G)_X) \circ \rho_X \\
&= B(id) \circ \rho_X \\
&= \rho_X,
\end{aligned}$$

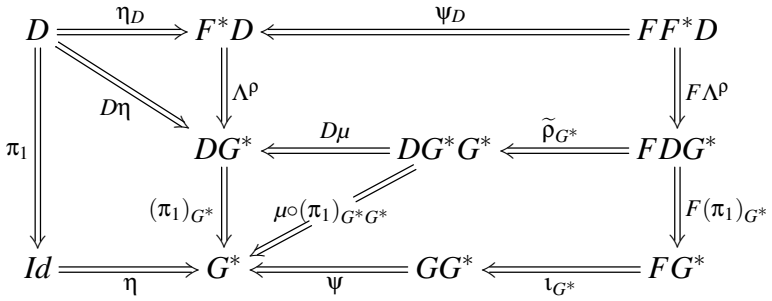
which proves our claim that inverse operation indeed results in ρ_X.

Now, we verify that Λ^{ρ} is an open distributive law. The first coherence condition holds by the definition of Λ^{ρ}. The verification of the second condition is entirely analogous to Lemma 3.5.2i in [Bar04]. In order to verify the third coherence condition for Λ^{ρ}, we show that the next two diagrams commute. We do so by applying induction on the terms, i.e. (6.3).

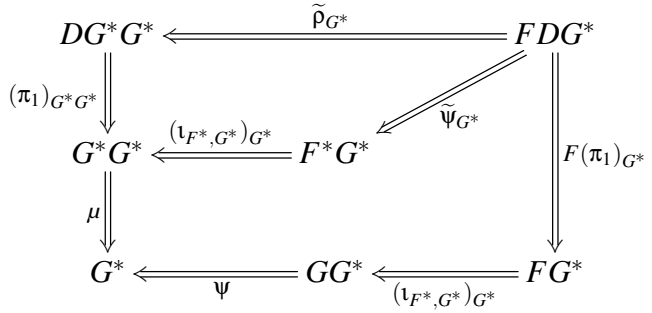
First, consider the following commuting diagram:



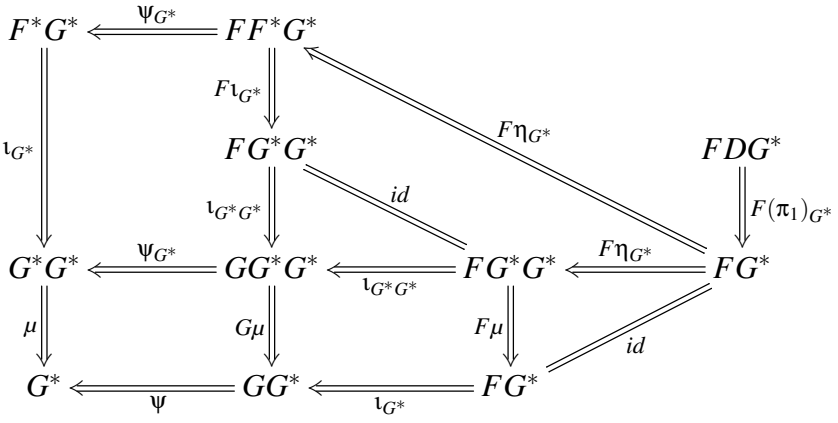
Now, consider the following diagram:



Everything but the bottom right pentagon commutes trivially (the top right pentagon holds per definition of Λ^ρ). Note that by using the definition of $\widetilde{\rho}$, we can split up the pentagon as follows:



The bottom region follows from the commutativity of the following diagram, in which we have unfolded the definition of $\widetilde{\psi} := \psi \circ F(\eta \circ \pi_1)$:



This completes the proof. □

In the case that $D := Id \times B$, then any open distributive law Λ (regardless of whether it is obtained from an open GSOS rule following the above procedure or not) induces an *operational model*, which can be defined as follows:

$$op_\Lambda \ \frac{X \xrightarrow{h} BX}{F^*X \xrightarrow{F^*\langle id,h\rangle} F^*DX \xrightarrow{\Lambda_X} DG^*X \xrightarrow{(\pi_2)_{G^*X}} BG^*X}.$$

Thus, the operational model takes an environment h (hypotheses about the behavior of variables) and maps it over the terms, and then applies the distributive law. The projection π_2 leaves us with the resulting behavior. Lemma 11 below provides an alternative way to define the operational model which does not require the specific assumption that $D := Id \times B$.

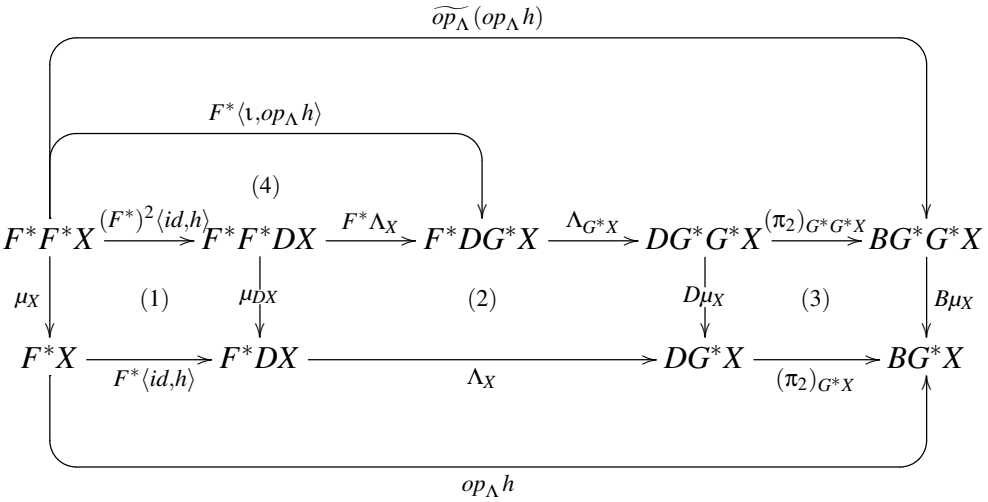
We will now prove compositionality of the operational model, a well-known property of bialgebraic semantics. The proof makes use of the second coherence condition of distributive laws as a key property. First, we will need an auxiliary, slightly modified version of op_Λ. This is a necessity due to the possible difference between ingoing and outgoing signature functors; when $F = G$ and $(\iota_{F^*,G^*})_X = id$, the definition below coincides with op_Λ.

$$\widetilde{op_\Lambda} \ \frac{F^*X \xrightarrow{h} BG^*X}{F^*F^*X \xrightarrow{F^*\langle(\iota_{F^*,G^*})_X,h\rangle} F^*DG^*X \xrightarrow{\Lambda_{G^*X}} DG^*G^*X \xrightarrow{(\pi_2)_{G^*G^*X}} BG^*G^*X}.$$

Lemma 9 (Compositionality)**.** *The following diagram commutes:*

$$\begin{array}{ccc} F^*F^*X & \xrightarrow{\widetilde{op_\Lambda}(op_\Lambda h)} & BG^*G^*X \\ {\scriptstyle \mu_X}\downarrow & & \downarrow{\scriptstyle B\mu_X} \\ F^*X & \xrightarrow[op_\Lambda h]{} & BG^*X \end{array}$$

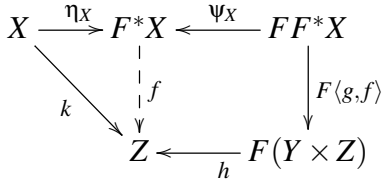
Proof. We prove that the following diagram commutes:



Region (1) follows by naturality of μ, region (2) by the second condition of open distributive laws, region (3) by naturality of π, region (4) follows by the third coherence condition of Λ (for the first projection of Λ) together with the definition of op_Λ (for the second projection of Λ). The remaining two regions follow by definition. □

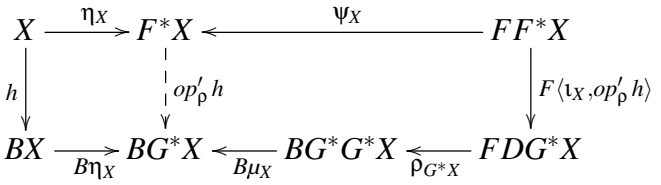
In light of Proposition 2, it is also possible to derive the operational model directly from a GSOS rule by using fold. We will need the following lemma in Section 6.5.2.

Lemma 10. *Given morphisms $k : X \to Z, h : F(Y \times Z) \to Y$ and $g : F^*X \to Y$ for some X, Y, Z, there exists a unique morphism f that makes the diagram commute:*

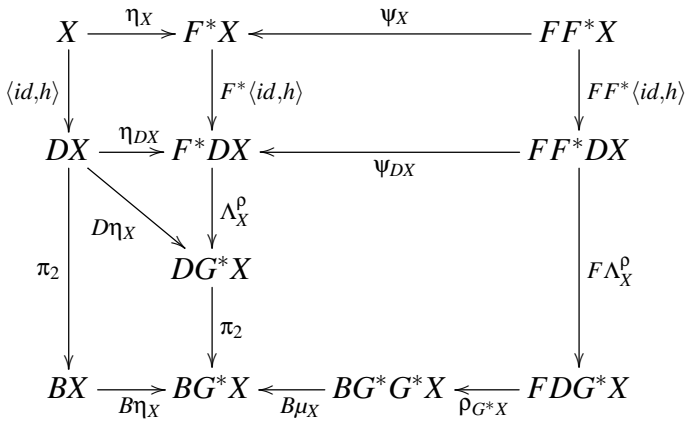


Proof. Remark that $h \circ F\langle g, f\rangle = h \circ F(g \times id) \circ F\langle id, f\rangle$. Then apply the structural recursion theorem with accumulators, i.e. Theorem 5.1 in [TP97]. □

Lemma 11. *The operational model $op_{\Lambda^\rho} h$, where the distributive law is obtained from a GSOS rule ρ, is equivalent to the unique morphism $op'_\rho h$ in following the diagram:*



Proof. Consider the following expansion of op_{Λ^ρ}:



The triangle commutes by the first coherence condition of Λ^{ρ}, the rest follows by definition. Since the above diagram commutes, we can conclude by Lemma 10 that the operational models are equivalent. □

6.3.3 Operational conservative extensions

In this section we will consider two Open GSOS rules, the *base language*, and the *extended language*. As a convention, we will write F_+, G_+ for the in- and outgoing signatures (terms) of the extended language, respectively, and B_+ for the behavior functor of the extension. Going back to the example in Section 6.3.1, B_+ could be $A_+ \times X$, where A_+ is a larger set of labels, extended with new side-effects. An example of this would be an extension for catching exceptions, in which case the extended label set needs to encode changes to the exception state.

Assumption 1. *We will assume in the rest of this chapter that the signature, behavior and copointed behavior functors of the base and extended language are chosen in such a way that there exist natural transformations* $\iota_{F,F_+}, \iota_{G,G_+}$, *and* ι_{B,B_+}. *Moreover, we require that the signature inclusion natural transformations commute as follows:*

$$\begin{array}{ccc} F & \xRightarrow{\iota} & F_+ \\ {\scriptstyle\iota}\Downarrow & & \Downarrow{\scriptstyle\iota} \\ G & \xRightarrow[\iota]{} & G_+ \end{array}$$

It is easy to prove that the above diagram extends to the free monads generated by the signature functors. We also have the obvious inclusion $\iota_{D,D_+} : D \Rightarrow D_+ := id \times \iota_{B,B_+}$. Finally, we remark that G and F_+ are not necessarily the same, as G may refer to operations that are not (yet) defined by the extension.

We call an extension *conservative* when the base language, as included in the extended language, retains its original behavior (see also [AFV99] for background information):

Definition 3. *Let* $\rho : FD \Rightarrow BG^*$ *and* $\rho_+ : F_+D_+ \Rightarrow B_+G_+^*$ *be Open GSOS rules. Then* ρ_+ is a conservative rule extension of ρ *if the diagram below holds.*

$$\begin{array}{ccccc} FD & \xRightarrow{\iota_D} & F_+D & \xRightarrow{F_+\iota} & F_+D_+ \\ {\scriptstyle\rho}\Downarrow & & & & \Downarrow{\scriptstyle\rho_+} \\ BG^* & \xRightarrow[B\iota]{} & BG_+^* & \xRightarrow[\iota_{G_+^*}]{} & B_+G_+^* \end{array}$$

Let $\Lambda : F^*D \Rightarrow DG^*$ *and* $\Lambda_+ : F^*_+D \Rightarrow D_+F^*_+$ *be natural transformations. Then* Λ_+ *is a* conservative (open) distributive law extension *of* Λ *if the diagram below holds.*

$$\begin{array}{ccccc} F^*D & \xRightarrow{\iota_D} & F^*_+D & \xRightarrow{F^*_+\iota} & F^*_+D_+ \\ \Lambda \Downarrow & & & & \Downarrow \Lambda_+ \\ DG^* & \xRightarrow[D\iota]{} & DG^*_+ & \xRightarrow[\iota_{G^*_+}]{} & D_+G^*_+ \end{array}$$

In the rest of this chapter we will omit the word "conservative", as everything we do in this chapter is in this spirit.

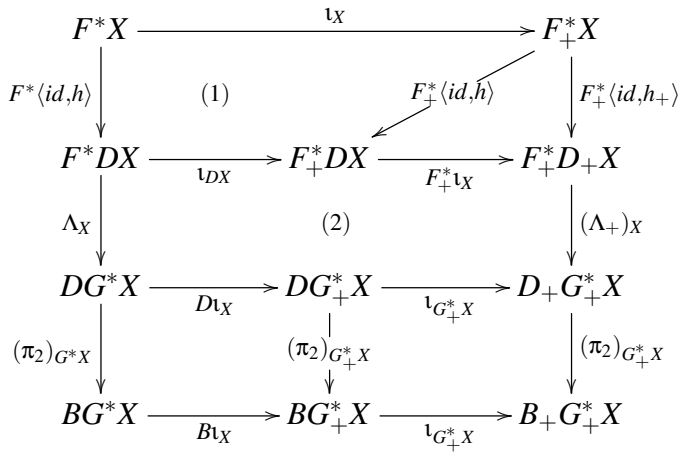
Proposition 3. *If* ρ_+ *is an extension of* ρ*, then* Λ^{ρ_+} *is an extension of* Λ^{ρ}.

Proof. The proof proceeds by induction on the terms. At the core of the induction proof, one is required to show that $\widetilde{\rho'}_{U'}$ is an extension of $\widetilde{\rho}_{G^*}$ (recall from Proposition 2 that $\widetilde{\rho}$ is used as an intermediate step to obtain the distributive law Λ^{ρ} from a rule ρ). This can be proved by considering the two cases where we post-compose the equality to be proved with the projections π_1 and π_2. In the first case, the commutativity can be proved making use of Assumption 1. In the second case, one makes use of the assumption that ρ' is an extension of ρ. The full proof has been carried out in Coq. □

Proposition 4. *Suppose that* Λ_+ *is an extension of* Λ*. Let* $h : X \to BX$ *be arbitrary, and fix* $h_+ := (\iota_{B,B_+})_X \circ h$*. Then it holds that* op_{Λ_+} *is an extension of* op_{Λ}*, i.e.:*

$$\begin{array}{ccccc} F^*X & & \xrightarrow{\iota_X} & & F^*_+X \\ {\scriptstyle op_\Lambda h}\downarrow & & & & \downarrow{\scriptstyle op_{\Lambda_+} h_+} \\ BG^*X & \xrightarrow[B\iota_X]{} & BG^*_+X & \xrightarrow[\iota_{G^*_+X}]{} & B_+G^*_+X \end{array}$$

Proof. We prove that the following diagram commutes.



Region (1) follows by naturality of ι_{F^*,F^*_+}, the triangle follows using the definition of ι_{D,D_+}, region (2) follows from Proposition 3, and the two squares at the bottom follow from the naturality of π_2. □

6.4 Silent Transitions

Perhaps the most straightforward way to represent silent transitions is by making them an option in the behavior, i.e. $Id + B$, so that left side means that the program made a silent transition, and the right side means that a step was made with "regular" behavior. However, the problem with this approach is that silent transitions are made explicit, which can lead to some undesirable results. For example, the terms seq skip x and x, although intuitively representing the same thing, have different semantics. With the above choice for the behavior functor, silent transitions are not *truly unobservable*. This section introduces a modified construction of Klin [Kli04], which develops distributive laws and operational models with truly unobservable silent transitions.

The approach we take in this section is to merge the rules of silent transition rules with open distributive laws, yielding a new open distributive law, from which an operational model with truly unobservable silent transitions is obtained. The approach makes use of the existence of a least fixpoint, and we will therefore need to assume that the underlying category is **CPPO**-enriched and that all morphisms are strict (in addition to the usual properties of the underlying category as required by the bialgebraic framework). A category is **CPPO**-enriched when its homsets, say H, are CPPOs (these are (small) posets with a least element, which are closed under LUBs '$\bigsqcup$' and omega chains), and that composition is continuous in both arguments. A morphism f is strict when both operations $f \circ \_$ and $\_ \circ f$ preserve

bottom elements. When all morphisms in the underlying category are strict, then a natural transformation $\bot$ between any two functors exists which consists of the collection of bottom elements of the homsets.

An important fact that will be used in this section is Tarski's theorem, which asserts that for every continuous map $\Psi : H \to H$, if $\Psi f \geqslant f$, then the least fixpoint of Ψ above f exists, and it is given by $\Psi^* f := \bigsqcup_{n \in \mathbb{N}} \Psi^n f$.

The examples in this section are aimed at the category $\mathbf{CPPO}_\bot$, in which the objects are CPPOs and the morphisms are continuous strict functions.

6.4.1 Silent transitions as unfolding rules

In this section we consider rules for silent transitions to be a natural transformation $r^0 : H \Rightarrow F^*H^*$ (where F and H are signature functors), which sends each operation to the term corresponding to the right-hand side of the silent transition.

Let us provide some examples. Set $FX := \mathsf{seq}\,(x\,y : X)$, and set $HX := \mathsf{loop}\,(x : X)$, the signature functor for the looping operation. For the looping operation, we define r^0_H as follows (the H subscript indicates that it represents the silent transitions belonging to the operations in H):

$$\begin{aligned} (r^0_H)_X : HX &\longrightarrow F^*H^*X \\ \mathsf{loop}\ x &\longmapsto [\mathsf{seq}\ x\ [\mathsf{loop}\ x]_H]_F \end{aligned}$$

Thus, a silent transition made by $\mathsf{loop}\ x$ is turned into a sequencing of its argument x with $\mathsf{loop}\ x$.

The mapping for seq can be defined with the help of an auxiliary functor $F_V := F(V + Id)$, where $V := \mathsf{skip}\ (= 1)$ stands for the coproduct of all values that occur on the left-hand side of silent transition rules for seq, which is only skip in this case. We can define $r^0_{F_V}$ for all objects X as follows:

$$\begin{aligned} (r^0_{F_V})_X : F_V X &\longrightarrow F^* F_V^* X \\ \mathsf{seq}\ (\kappa_2\, x)\ (\kappa_2\, y) &\longmapsto [\mathsf{seq}\ [x]_{F_V}\ [y]_{F_V}]_F \\ \mathsf{seq}\ (\kappa_1\, \mathsf{skip})\ (\kappa_2\, y) &\longmapsto [[y]_{F_V}]_F \\ \mathsf{seq}\ (\kappa_2\, x)\ (\kappa_1\, \mathsf{skip}) &\longmapsto [[x]_{F_V}]_F \\ \mathsf{seq}\ (\kappa_1\, \mathsf{skip})\ (\kappa_1\, \mathsf{skip}) &\longmapsto \bot \end{aligned}$$

Note that the free monad generated by F_V consists of the terms in which each value is preceded by an operation. We have deliberately avoided using the coproduct $F + V$ as the signature functor, as that would have required us to consider skip to have computational behavior (however, it is easy to prove that $F_V^* + V \cong (F + V)^*$).

The following definitions show how, by making use of Tarski's least fixpoint construction, the above rules can be (infintely) unfolded. We start off by obtaining a single-step unfolding of a term H^* from r^0 (recall from Section 6.2 that $\phi_F := \psi_F \circ F\eta_F$).

Definition 4. *Let F, H be arbitrary signature functors and assume a natural transformation $r^0 : H \Rightarrow F^*H^*$ such that (recall that $\phi_F : F \Rightarrow F^* := \psi_F \circ F\eta_F$):*

$$r^0 \circ \iota_{F,H} = F^*\eta_H \circ \phi_F.$$

A regular unfolding rule *is a natural transformation induced by r^0 as follows:*

$$r \;\frac{H \overset{r^0}{\Longrightarrow} F^*H^*}{H^* \overset{\kappa_H}{\Longrightarrow} Id + HH^* \overset{\eta_F + (r^0)_{H^*}}{\Longrightarrow} F^* + F^*H^*H^* \overset{[F^*\eta_H, F^*\mu_H]}{\Longrightarrow} F^*H^*}.$$

The condition on r^0 essentially means that any operations that have no defined silent transitions should be left untouched. Let us verify that this holds for seq as an example:

$$\begin{array}{ccc} \mathsf{seq}\ x\ y & \overset{(\phi_F)_X}{\longmapsto} & [\mathsf{seq}\ x\ y]_F \\ \Big\downarrow{\scriptstyle (\iota_{F,F_V})_X} & & \Big\downarrow{\scriptstyle F^*(\eta_{F_V})_X} \\ \mathsf{seq}\ (\kappa_2 x)\ (\kappa_2 y) & \underset{(r^0_{F_V})_X}{\longmapsto} & [\mathsf{seq}\ [x]_{F_V}\ [y]_{F_V}]_F \end{array}$$

The next step is to define the (infinite) unfolding $\bar{r}$ from r (recall that $\bot$ below is the collection of bottom elements of all homsets).

Definition 5. *The natural transformation $v : H^* \Rightarrow Id := [id, \bot] \circ \kappa_H$ is called the* variable classifier. *The* infinite unfolding *of r is defined (for any object X) as:*

$$\bar{r}_X : H^*X \to F^*X := \Phi^*_X(\eta_X \circ v_X),$$

which is a least fixpoint of the auxiliary function Φ:

$$\Phi \;\frac{H^* \overset{f}{\Rightarrow} F^*}{H^* \overset{r}{\Rightarrow} F^*H^* \overset{F^*f}{\Longrightarrow} F^*F^* \overset{\mu}{\Rightarrow} F^*}.$$

The variable classifier is used to query whether a given term is a variable. One can show that the above fixpoint $\bar{r}_X$ is well-defined in a **CPPO**-enriched category, and that $\bar{r}$ is a natural transformation, see [Kli04].

We compute a few examples based on the silent rules earlier in this section:

- $\overline{r}\ [\mathsf{seq}\ \mathsf{skip}\ (\mathsf{seq}\ \mathsf{skip}\ x)]_{F_V} = [x]_F$;
- $\overline{r}\ [\mathsf{seq}\ \mathsf{skip}\ \mathsf{skip}]_{F_V} = \bot$ (because seq skip skip has no rule);
- $\overline{r}\ [\mathsf{loop}\ x]_H = [\mathsf{seq}\ x\ (\mathsf{seq}\ x\ (\mathsf{seq}\ x\ \ldots))]_F$

Note that $\overline{r}$ cannot be applied directly to a value, e.g. $\overline{r}\ (\mathsf{skip})$ is not well-defined, because $F_V = F(V + Id)$ enforces that values are always preceded by an operation.

Lemma 12 (Klin [Kli04])**.** *Let F and H be signature functors and $\overline{r} : H^* \Rightarrow F^*$ be an infinite unfolding rule as in Definition 5. Then $\overline{r} \circ \eta_H = \eta_F$.*

Lemma 13 (Klin [Kli04])**.** *Let F and H be signature functors. Any infinite unfolding $\overline{r} : H^* \Rightarrow F^*$ obtained, using Definition 5, from a regular unfolding rule $r : F^* \Rightarrow F^*H^*$ is a monad morphism.*

We incorporate the infinite unfolding into a distributive law by setting:

$$\Lambda^r \ \frac{}{H^*D \overset{\overline{r}_D}{\Longrightarrow} F^*D \overset{\Lambda}{\Longrightarrow} DG^*}.$$

Proposition 5. *Let r be as in Lemma 13. Then Λ^r is an open distributive law.*

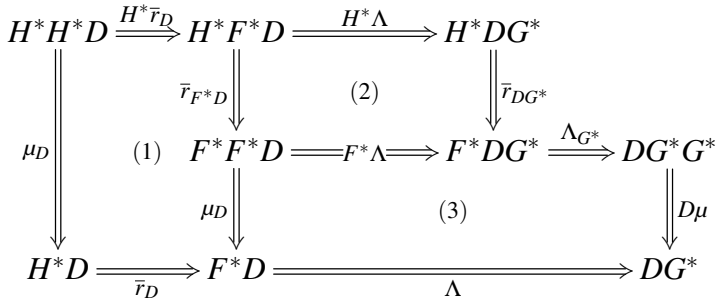
Proof. It is obvious that Λ^r is natural, as it is a composition of two natural tranformations. We will verify the three conditions of open distributive laws.

The first condition follows from

$$\begin{aligned}\Lambda^r \circ (\eta_H)_D &= \Lambda \circ \overline{r}_D \circ (\eta_H)_D \\ &= \Lambda \circ (\eta_F)_D \\ &= D\eta_G,\end{aligned}$$

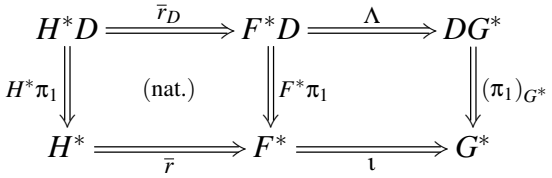
by making use of Lemma 12, and the first coherence condition of Λ.

For the second condition, consider the following diagram, while expanding the definition of Λ^r:



Region (1) commutes due to $\bar{r}$ being a monad morphism, region (2) commutes by the naturality of $\bar{r}$, and region (3) commutes by the second coherence condition of Λ.

We now prove the third condition. We claim that $\iota_{F^*,G^*} \circ \bar{r}$ is a valid inclusion morphism. Making use of naturality of $\bar{r}$ and the third coherence condition of Λ we can see that the following diagram commutes:



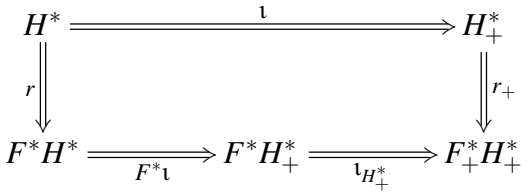
This proves the third condition and thereby completes the proof. □

6.4.2 Unfolding rule extensions

In this section we define what extensions of unfolding rules are and prove an extension theorem for operational models based on unfolding rules, a generalization of Proposition 4. A corollary of this theorem is that the operational model op_{Λ^r} is an extension of the operational model op_Λ, i.e. adding rules for silent transitions is a proper extension of a pre-existing set of GSOS rules.

Throughout this entire section, H, H_+, F, F_+, G and G_+ are signature functors, and B and B_+ are behavior functors. As in Assumption 1, we assume the existence of the usual inclusion functors.

Definition 6. *Suppose that we have unfolding rules $r : H^* \Rightarrow F^*H^*$ and $r_+ : H_+^* \Rightarrow F_+^*H_+^*$. Then r_+ is an* unfolding rule extension *of r if the following condition is satisfied:*



Lemma 14. *Let $v_F : F^* \Rightarrow Id$ and $v_H : H^* \Rightarrow Id$ be variable classifiers. Then $v_H \circ \iota_{F^*,H^*} = v_F$.*

Proof. The proof follows from the following derivation:

$$\begin{aligned} v_H \circ \iota_{F^*,H^*} &= [id, \bot] \circ \kappa_H \circ \iota_{F^*,H^*} \\ &= [id, \bot] \circ (id + (\iota_{F,H} \circ F\iota_{F^*,H^*})) \circ \kappa_F \\ &= [id, \bot] \circ \kappa_F \\ &= v_F. \end{aligned}$$

Note that the second step follows by the definition of ι_{F^*,H^*}. □

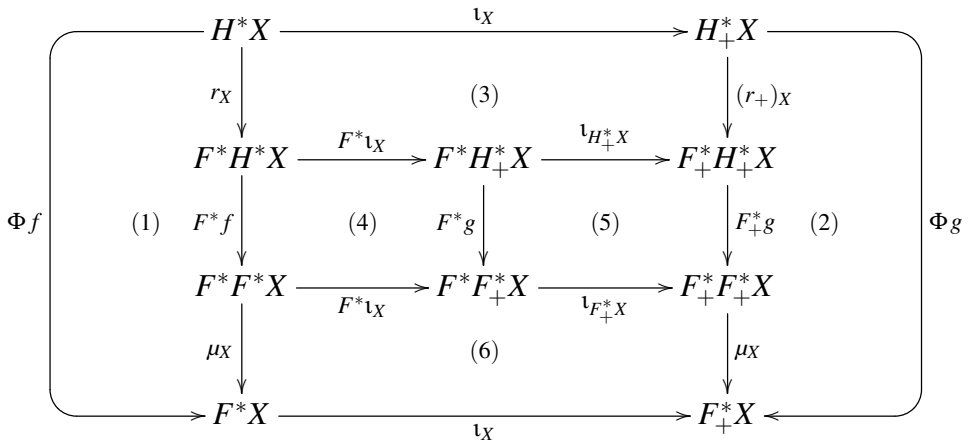
Lemma 15. *Suppose that $\overline{r} : H^* \Rightarrow F^*$ and $\overline{r_+} : H^*_+ \Rightarrow F^*_+$ are infinite unfolding rules, obtained from the unfolding rules respectively $r : H^* \Rightarrow F^*H^*$ and $r_+ : H^*_+ \Rightarrow F^*_+H^*_+$, and suppose that r_+ is an extension of r. Then $\overline{r_+} \circ \iota_{H^*,H^*_+} = \iota_{F^*,F^*_+} \circ \overline{r}$.*

Proof. We proceed by fixpoint induction on Φ (recall Definition 5).

Base case. Making use of Lemma 14 in the second step:

$$\begin{aligned}\eta_{F_+} \circ \nu_{H_+} \circ \iota_{H^*,H^*_+} &= \eta_{F_+} \circ \nu_H \\ &= \iota_{F^*,F^*_+} \circ \eta_F \circ \nu_H.\end{aligned}$$

Induction step. Assume that $(\iota_{F^*,F^*_+})_X \circ f = g \circ (\iota_{H^*,H^*_+})_X$ for some $f : H^*X \to F^*X$ and $g : H^*_+X \to F^*_+X$. Now consider the following diagram:

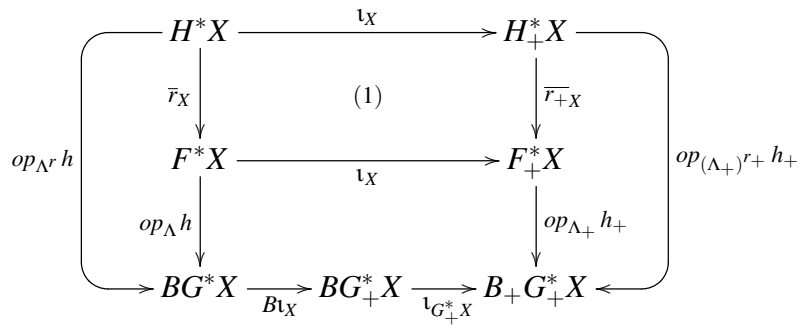


Regions (1) and (2) follow from the definition of Φ, region (3) is the assumption of this lemma, region (4) is the induction hypothesis, region (5) follows from the naturality of ι_{F^*,F^*_+}, and region (6) follows from the fact that ι_{F^*,F^*_+} is a monad morphism.

This completes the proof. □

Theorem 3 (Extension)**.** *Let $\Lambda_+ : F^*_+D_+ \Rightarrow D_+G^*_+$ be an open distributive law which is an extension of $\Lambda : F^*D \Rightarrow DG^*$, and suppose that the unfolding rule $r_+ : H^*_+ \Rightarrow F^*_+H^*_+$ is an extension of $r : H^* \Rightarrow F^*H^*$. Then $op_{(\Lambda_+)^{r_+}}$ is an extension of op_{Λ^r}.*

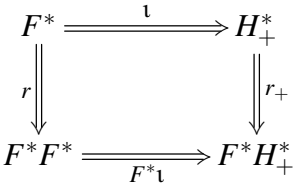
Proof. As in Proposition 4 assume arbitrary $h : X \to BX$, and set $h_+ := (\iota_{B,B_+})_X \circ h$. We prove the theorem by showing that everything in the following diagram commutes:



Region (1) commutes by Lemma 15; the other three regions commute by Proposition 4. Note that for the two side regions we instantiate ι_{H^*,F^*} with $\overline{r}$ and $\overline{r_+}$. This is permitted, as the only requirement about ι_{H^*,F^*} by Proposition 4 is that it is a natural transformation. □

Corollary 1. *Suppose that $\Lambda : F^*D \Rightarrow DG^*$ is an open distributive law and that $r_+ : H^*_+ \Rightarrow F^*H^*_+$ is a regular unfolding rule. Then $op_{\Lambda^{r_+}}$ is an extension of op_Λ.*

Proof. Set $r : F^* \Rightarrow F^*F^*$ to be $[\eta_{F^*} \circ \eta, \phi_{F^*}] \circ \kappa$. It then follows that $\overline{r} = id$, see Klin [Kli04, p.29], and therefore $op_{\Lambda^r} h = op_\Lambda h$ for any $h : X \to BX$. Because r_+ is regular, from Lemma 9 in [Kli04] it follows that r_+ is an extension of r, i.e.:



Using the above two observations, from Theorem 3 it then follows that $op_{\Lambda^{r_+}}$ is an extension of op_Λ (instantiating $H = F$, $F_+ = F$, $G_+ = G$, and $B_+ = B$). □

6.5 Algebraic Laws

In favor of scalability, it is desirable that bisimulation relations that were established in a base language also hold in any extension of the base language, because this prevents one from having to redo the arduous task of re-verifying each bisimulation relation in the extended language. However, whether this holds depends on the precise definition of bisimulation one chooses to work with.

A straightforward variation of the standard notion of bisimulation would be to demand that all substitutions of variables with closed terms are again bisimulations, this is also called Closed-Instance bisimulation (CI-bisimulation). It turns out that CI-bisimulations are not preserved by language extensions. Counterexamples pointing out this fact can be found in [MMR10].

FH-bisimulations, introduced by De Simone [DS85], are more general than CI-bisimulations and do take into account that the variables of the terms in question may exhibit arbitrary behavior. FH-bisimulations have been studied in the context of transition systems, but not in the context of coalgebras. We introduce FH-bisimulations here, adapted to our coalgebraic setting. An important way in which FH-bisimulation differs from the standard notion of bisimulation is that FH-bisimulations require the state-space to consist of terms, and that the coalgebras are operational models. It was proved by Mosses et al. [MMR10] that FH-bisimulations are indeed preserved by conservative language extensions; we will prove the same result in our coalgebraic setting in Section 6.5.1.

Definition 7 (Extended FH-bisimulation)**.** *Let $\Lambda : FD \Rightarrow DG$ be an open distributive law. A relation $\langle R, \pi_1, \pi_2 \rangle$, where $\pi_1, \pi_2 : R \to F^*X$, is an* extended FH-bisimulation relation *if for every environment $h : X \to BX$, there exists $\gamma_h : R \to BR$, called the B-*structure*, such that the following diagram commutes:*

$$\begin{array}{ccccccc}
F^*X & \xleftarrow{\pi_1} & R & \xrightarrow{\pi_2} & F^*X \\
{\scriptstyle op_\Lambda h}\downarrow & & \downarrow{\scriptstyle \gamma_h} & & \downarrow{\scriptstyle op_\Lambda h} \\
BG^*X & \xleftarrow[B\iota_X]{} BF^*X & \xleftarrow[B\pi_1]{} BR & \xrightarrow[B\pi_2]{} BF^*X & \xrightarrow[B\iota_X]{} BG^*X
\end{array}$$

When the set of rules is closed, i.e. $F = G$, and $(\iota_{F^*,G^*})_X = id$, then the above diagram can be read as an Aczel-Mendler bisimulation. If additionally we consider the terms to be closed, then the above definition coincides with the standard definition of a bisimulation relation R on the coalgebra op_Λ. Since it is possible that $F = G$, but $(\iota_{F^*,G^*})_X \neq id$, we introduce a technical variation:

Definition 8 (FH-bisimulation)**.** *Let $\Lambda : FD \Rightarrow DF$ be a(n) (open) distributive law. A relation $\langle R, \pi_1, \pi_2 \rangle$ with $\pi_1, \pi_2 : R \to F^*X$ is an* FH-bisimulation *when for all $h : X \to BX$, $\langle R, \pi_1, \pi_2 \rangle$ is an Aczel-Mendler bisimulation for the coalgebra $op_\Lambda h$. We say that the terms $t, u : F^*X$ are* FH-bisimilar *if there exists an FH-bisimulation relation R such that $t \, R \, u$.*

Only Theorem 4 in this section will involve extended FH-bisimulations, the other results are about FH-bisimulations.

Since FH-bisimulations relate terms, we call a pair of such terms an *algebraic law*. Examples of algebraic laws can be found by unfolding silent transition rules:

$\mathsf{seq}\ \mathsf{skip}\ x = x$ and $\mathsf{loop}\ x = \mathsf{seq}\ x\ (\mathsf{seq}\ x\ ...)$. The following proposition formalizes this.

Proposition 6. *Let $\bar{r} : H^* \Rightarrow F^*$ be an infinite unfolding rule (with H and F as in Section 6.4), and $\Lambda : F^*D \Rightarrow DH^*$ be an open distributive law. Then for any $t : H^*X$, it holds that t and $(\iota_{F^*,H^*})_X\,(\bar{r}_X\,t)$ are FH-bisimilar, i.e. every term is FH-bisimilar to its infinite unfolding.*

Proof. Set

$$R := \{\langle t, (\iota_{F^*,H^*})_X\,(\bar{r}_X\,t)\rangle \mid t : H^*X\}.$$

We claim that $\langle R + H^*X, [\pi_1, id], [\pi_2, id]\rangle$ is an FH-bisimulation (in the category **Set**, the corresponding relation would be the union of R with the reflexive relation). This claim implies the present proposition.

The proof proceeds by considering each side of the coproduct as a separate case. The right-hand side is easy: for any $h : X \to BX$, take $\gamma_h = B\kappa_2 \circ op_\Lambda\, h$ to be the B-structure. This leaves us with the left-hand side of the coproduct.

We claim that

$$\gamma_h \;\; \frac{}{R \xrightarrow{\pi_1} H^*X \xrightarrow{op_{\Lambda^r}\,h} BH^*X \xrightarrow{B\kappa_2} B(R + H^*X)}$$

is a valid B-structure for the R-case. We verify the left-hand side of the FH-bisimulation diagram:

$$\begin{aligned} B[\pi_1, id] \circ \gamma_h &= B[\pi_1, id] \circ B\kappa_2 \circ op_{\Lambda^r}\, h \circ \pi_1 \\ &= B([\pi_1, id] \circ \kappa_2) \circ op_{\Lambda^r}\, h \circ \pi_1 \\ &= B(id) \circ op_{\Lambda^r}\, h \circ \pi_1 \\ &= op_{\Lambda^r}\, h \circ \pi_1. \end{aligned}$$

For the right-hand side we first recall two facts:

- $op_{\Lambda^r}\, h = op_\Lambda\, h \circ \bar{r}_X$ (by Proposition 4, see also Theorem 3);
- $\bar{r}_X \circ (\iota_{F^*,H^*})_X = id$ (Lemma 13 in [Kli04]).

Using these facts, we can prove:

$$\begin{aligned}
B[\pi_2, id] \circ \gamma_h &= B[\pi_2, id] \circ B\kappa_2 \circ op_{\Lambda^r} h \circ \pi_1 \\
&= B([\pi_2, id] \circ \kappa_2) \circ op_{\Lambda^r} h \circ \pi_1 \\
&= B(id) \circ op_{\Lambda^r} h \circ \pi_1 \\
&= op_{\Lambda^r} h \circ \pi_1 \\
&= op_{\Lambda} h \circ \bar{r}_X \circ \pi_1 \\
&= op_{\Lambda} h \circ \bar{r}_X \circ (\iota_{F^*, H^*X})_X \circ \bar{r}_X \circ \pi_1 \\
&= op_{\Lambda} h \circ \bar{r}_X \circ \pi_2 \\
&= op_{\Lambda^r} h \circ \pi_2.
\end{aligned}$$

This validates the claim and therefore the proof is completed. □

6.5.1 Preservation of algebraic laws

We now prove that algebraic laws are preserved by conservative language extensions. This result makes use of Theorem 3, which says that operational behavior is preserved by conservative extensions. As in Mosses et al. [MMR10], we will need the assumption that the extension adds no new behavior, meaning that the behavior functors of the base and extended language are the same. Counterexamples that show why the theorem without this assumption fails can be found in [MMR10].

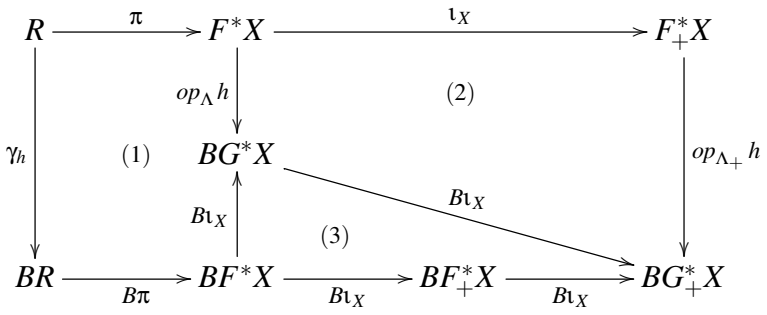
Theorem 4 (Preservation)**.** *Suppose that $\Lambda_+ : FD \Rightarrow DG$ is a conservative extension of $\Lambda : F_+D \Rightarrow DG_+$. Furthermore, suppose that $\langle R, \pi_1, \pi_2 \rangle$ is an extended FH-bisimulation relation on op_Λ, where $\pi_1, \pi_2 : R \to F^*X$. Then*

$$\langle R, (\iota_{F^*, F_+^*})_X \circ \pi_1, (\iota_{F^*, F_+^*})_X \circ \pi_2 \rangle$$

is an extended FH-bisimulation on op_{Λ_+}.

Proof. Let $h : X \to BX$ be arbitrary. The theorem assumes that $\langle R, \pi_1, \pi_2 \rangle$ is an extended FH-bisimulation, which means that for any h, there exists a morphism $\gamma_h : R \to BR$ satisfying the diagram of Definition 7. We claim that γ_h is also a valid B-structure for the extended FH-bisimulation relation to be proved in this theorem.

Due to its symmetry, we verify the extended FH-bisimulation diagram for $\pi = \pi_1, \pi_2$:



Region (1) commutes by the assumption that R is an extended FH-bisimulation on op_Λ, and region (2) commutes by Theorem 3. Region (3) commutes by Assumption 1. This finishes the proof. □

6.5.2 Combining algebraic laws

Similar to standard bisimulations, there exists a greatest FH-bisimulation relation, notation $\overset{\text{FH}}{\leftrightarrow}$, which is the union of all FH-bisimulations, i.e for $t,u : F^*X$,

$$t \overset{\text{FH}}{\leftrightarrow} u \iff \exists \langle R, \pi_1, \pi_2 \rangle \text{ such that } R \text{ is an FH-bisimulation, and } t\,R\,u.$$

The relation $\overset{\text{FH}}{\leftrightarrow}$ itself is an FH-bisimulation relation, in fact, it is an equivalence relation.

We shall use the notation $t[f] := \mu_X(F^* f\, t)$ for the substitution of a map $f : X \to F^*X$ in a term t.

Proposition 7 (Instantiation)**.** *For any $t : F^*X$ and $f : X \to \overset{\text{FH}}{\leftrightarrow}$ (a map assigning FH-bisimilar pairs of terms to variables), $t[\pi_1 \circ f]$ and $t[\pi_2 \circ f]$ are FH-bisimilar.*

Proof. We claim that

$$\langle F^* \overset{\text{FH}}{\leftrightarrow}, \mu_X \circ F^*\pi_1, \mu_X \circ F^*\pi_2 \rangle$$

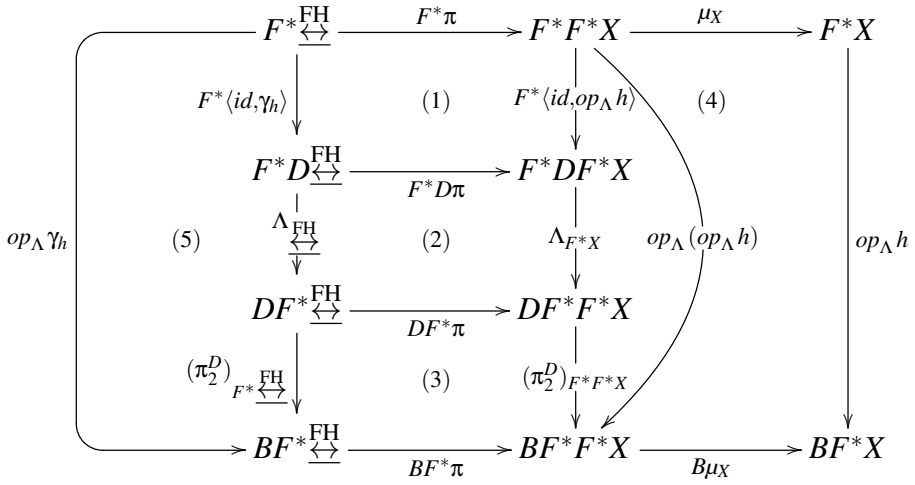
is an FH-bisimulation relation. The FH-bisimilarity in the present proposition is implied by this claim, because $(F^* f)t : F^* \overset{\text{FH}}{\leftrightarrow}$, and $\mu_X(F^*\pi((F^* f)t)) = t[\pi \circ f]$, for $\pi = \pi_1, \pi_2$.

Suppose that we are provided with some $h : X \to BX$. The diagram below shows that

$$op_\Lambda \gamma_h : F^* \overset{\text{FH}}{\leftrightarrow} \longrightarrow BF^* \overset{\text{FH}}{\leftrightarrow}$$

is a B-structure that satisfies the FH-bisimulation diagram for $F^* \overset{\text{FH}}{\leftrightarrow}$, in which γ_h is a B-structure derived from the fact that $\overset{\text{FH}}{\leftrightarrow}$ itself is an FH-bisimulation. Due to its

symmetry, we show the diagram for $\pi = \pi_1, \pi_2$. To avoid confusion between the names $\pi_2 : \underline{\overset{\text{FH}}{\leftrightarrow}} \to F^*X$ and $\pi_2 : D \Rightarrow B$, we shall call the latter π_2^D in the diagram below.



Regions (1-3) follow respectively from the fact that $\underline{\overset{\text{FH}}{\leftrightarrow}}$ is an FH-bisimulation, and that Λ and π_2^D are natural transformations. Region (4) follows from compositionality of the operational model, i.e. Lemma 9, and note that because $F = G$ and $(\iota_{F^*,G^*})_X = id$, we have $\widetilde{op_\Lambda}\,(op_\Lambda h) = op_\Lambda\,(op_\Lambda h)$. Finally, region (5) follows by definition. □

We leave proving that for any $t, u : F^*X$ such that $t \underline{\overset{\text{FH}}{\leftrightarrow}} u$ implies $t[\pi_2 \circ f] \underline{\overset{\text{FH}}{\leftrightarrow}} u[\pi_2 \circ f]$ as an open problem. Together with the Instantiation proposition above, one could then derive, by making use of the fact that $\underline{\overset{\text{FH}}{\leftrightarrow}}$ is transitive, that $t[\pi_1 \circ f] \underline{\overset{\text{FH}}{\leftrightarrow}} u[\pi_2 \circ f]$, which would show that $\underline{\overset{\text{FH}}{\leftrightarrow}}$ is substitutive [Ren00]. In [MSvE13] the author has proved this under the rather strong assumption that X and F^*X are isomorphic.

6.6 Running the operational semantics

This section highlights another application of Theorem 3.

We first recall an important notion of the theory of coalgebra. *Final coalgebras* are pairs $\langle Z, \zeta : Z \to BZ \rangle$ which enjoy the property that there exists an operator $unfold : (X \to BX) \to X \to Z$, which maps a coalgebra c to the unique coalgebra homomorphism from c to ζ. Here Z is the greatest solution to the equation $X \cong BX$. In case of the leading example $BX := A \times X$, the object of the final coalgebra

would be an infinite stream of A's. We will assume that B and B_+ have been chosen such that they indeed have final coalgebras.

When the rules are closed, then $op_\Lambda h$ is a coalgebra and we can "run" the operational semantics by unfolding it, i.e for any final coalgebra $\langle Z, \zeta \rangle$, signature functor F and (open) distributive law $\Lambda : FD \Rightarrow DF$, define:

$$run_\Lambda h : F^*X \to Z := unfold\,(op_\Lambda h).$$

Set $\iota_{Z,Z_+} := unfold\,((\iota_{B,B_+})_Z \circ \zeta)$. We can prove that running the extended operational model is faithful to running the base model.

Proposition 8. *Suppose that $\Lambda : F^*D \Rightarrow DF^*$ and $\Lambda_+ : F_+^*D \Rightarrow DF_+^*$ are (open) distributive laws, and that Λ_+ is an extension of Λ. Then for all $h : X \to BX$ it holds that*

$$\begin{array}{ccc} F^*X & \xrightarrow{run_\Lambda h} & Z \\ {\scriptstyle \iota_X}\downarrow & & \downarrow{\scriptstyle \iota} \\ F_+^*X & \xrightarrow[run_{\Lambda_+} h_+]{} & Z_+ \end{array}$$

Proof. Let $h : X \to BX$ be arbitrary, and consider the following diagram of B_+-coalgebras:

$$\begin{array}{ccc} \langle F^*X, (\iota_{B,B_+})_X \circ op_\Lambda h \rangle & \xrightarrow{run_\Lambda h} & \langle Z, (\iota_{B,B_+})_Z \circ \zeta \rangle \\ {\scriptstyle (\iota_{F^*,F_+^*})_X}\downarrow & & \downarrow{\scriptstyle \iota_{Z,Z_+}} \\ \langle F_+^*X, op_{\Lambda_+} h_+ \rangle & \xrightarrow[run_{\Lambda_+} h_+]{} & \langle Z_+, \zeta_+ \rangle \end{array}$$

By definition, $run_{\Lambda_+} h_+$ and ι_{Z,Z_+} are B_+-coalgebra homomorphisms, $run_\Lambda h$ is a B_+-coalgebra homomorphism by making use of its definition together with naturality of ι_{B,B_+}, and $(\iota_{F^*,F_+^*})_X$ is a B_+-coalgebra homomorphism by Theorem 3 and naturality of ι_{B,B_+}. Because Z_+ is final, the above diagram commutes, and hence the diagram in the statement of the present proposition commutes. □

6.7 Related Work

The results in this chapter were developed within the bialgebraic semantics framework, a body of research initiated by Turi and Plotkin [TP97].

FH-bisimulations were originally introduced by De Simone [DS85]. The theorems in Section 6.5, which prove that FH-bisimulations are preserved by conservative extensions, transfer the original results, obtained in the more traditional set-theoretic approach to SOS by Mosses et al. [MMR10], to Turi and Plotkin's bialgebraic framework [TP97]. There are a number of differences between the work of Mosses et al. and the present chapter. First, in this chapter we were forced to be quite explicit about the various signature functors involved in the operational rules and their extensions, due to the way we applied the bialgebraic framework. Second, by developing the theory in the present chapter for arbitrary behavior functors, we avoided being specific about side-effects of each language. For example, we expect that our work is general enough to support MSOS-style descriptions (see Chapter 4), although verifying this is left for future work. Finally, the bialgebraic framework, which is rooted in category theory, makes it easy to approach proofs in a very structured manner. For example, in the proof of the Preservation Theorem (Theorem 4), the use of the Extension Theorem (for unfolding rules) (Theorem 3) can be seen as a drop-in replacement of Proposition 4. This replacement generalizes the Preservation Theorem so that it holds that e.g. the algebraic law $\mathsf{seq}\ \mathsf{skip}\ x = x$ is preserved by any conservative extension (in which the behavior functor is the same), and vice versa, that adding an algebraic law (through the addition of a silent rule) preserves any pre-existing algebraic laws.

The dichotomy between value terms and computational terms was emphasized by Churchill and Mosses [CM13], who introduce a rule format built on the tyft format, which has built-in rules to deal with silent transitions. They provide a variant of bisimilarity, and prove that it is a congruence in the resulting transition system. The distributive law Λ^r of Section 6.4 has similar characteristics, through the infinite unfolding of silent transitions. This law is a variant of the one introduced by Klin [Kli04].

An alternative to considering only free monads as in the present chapter, is to quotient the term monad by the algebraic laws. Bonsangue et al. [BHKR13] prove that if Λ respects the algebraic laws, then there is a unique distributive law Λ' such that the quotient map is a well-behaved translation from Λ to Λ'. After the work in this chapter was developed, several authors have proposed frameworks to handle silent steps in a coalgebraic setting [SW13, BMSZ14]. The focus of both papers was to recover known language semantics of automata and transition systems, but they have not explored at all the connections with bialgebraic semantics.

A modular variant of GSOS has been provided by Jaskelioff et al. [JGH11] as part of a HASKELL implementation of the bialgebraic framework. They distinguish ingoing from outgoing signatures, as in the present chapter, but consider the outgoing signature as an abstract parameter of each modular rule, and add type-class constraints to ensure the inclusion of certain operations in the outgoing

signature.

Watanabe [Wat02] provides an interpretation of operational conservative extensions [AFV99] in terms of distributive laws, and proves a variant of Theorem 4, but does not treat the difference between ingoing and outgoing signatures.

6.8 Conclusions

We have provided an operational rule format, tailored to the modular description of programming languages. The semantics supports truly unobservable transitions, as generated by rules for silent transitions. We have proved that algebraic laws are preserved by conservative extensions of the operational semantics, and that algebraic laws are substitutive. Our work has been developed within the bialgebraic framework [TP97], making it amenable to implementation in a theorem prover [MS13].

In future work we wish to ease the condition in Section 6.5.2 on the distributive law, enabling the substitutivity of algebraic laws for a wider range of silent transition rules. We would also like to explore applications to software verification. In particular, one can view the operational rules of programming languages as pointwise extensions in the sense of [HK11]. We expect that this will lead to a structured way to obtain sound Hoare logics for trace-based semantics such as [NU10].

Acknowledgments. The inspiration for this work arose from consultation with Peter D. Mosses by the author. Without his support this work would not have been possible. The author is also grateful to Alexandra Silva whose sharp comments helped improve this chapter.

Bibliography

[ABF+05] Brian E. Aydemir, Aaron Bohannon, Matthew Fairbairn, J. Nathan Foster, Benjamin C. Pierce, Peter Sewell, Dimitrios Vytiniotis, Geoffrey Washburn, Stephanie Weirich, and Steve Zdancewic. Mechanized metatheory for the masses: The PoplMark challenge. In Joe Hurd and Tom Melham, editors, *Proceedings of the 18th International Conference on Theorem Proving in Higher Order Logics (TPHOLs'05)*, volume 3603 of *LNCS*, pages 50–65. Springer, 2005. Cited on page 49.

[ACGI11] Luca Aceto, Georgiana Caltais, Eugen-Ioan Goriac, and Anna Ingolfsdottir. PREG Axiomatizer – A ground bisimilarity checker for GSOS with predicates. In Andrea Corradini, Bartek Klin, and Corina Cîrstea, editors, *Proceedings of the 4th Conference on Algebra and Coalgebra in Computer Science (CALCO'11)*, volume 6859 of *LNCS*, pages 378–385. Springer, 2011. Cited on page 86.

[AFV99] Luca Aceto, Wan Fokkink, and Chris Verhoef. Structural operational semantics. In J.A. Bergstra, A. Ponse, and S.A. Smolka, editors, *Handbook of Process Algebra*, pages 197–292. Elsevier, 1999. Cited on pages 100 and 116.

[AM89] Peter Aczel and Nax Mendler. A final coalgebra theorem. In *Category Theory and Computer Science*, pages 357–365. Springer, 1989. Cited on page 91.

[Bar03] John Barnes. *High Integrity Software: The Spark Approach to Safety and Security*. Addison-Wesley, 2003. Cited on page 7.

[Bar04] Falk Bartels. *On generalised coinduction and probabilistic specification formats*. PhD thesis, Vrije Universiteit Amsterdam, 2004. Cited on pages 68, 80, 81, 85, 86, and 96.

[BBHI05] Alan Bundy, David Basin, Dieter Hutter, and Andrew Ireland. *Rippling: meta-level guidance for mathematical reasoning*. Cambridge University Press, 2005. Cited on page 45.

[BCO06] Josh Berdine, Cristiano Calcagno, and Peter O'Hearn. Smallfoot: Modular automatic assertion checking with separation logic. In Frank de Boer, Marcello Bonsangue, Susanne Graf, and Willem-Paul de Roever, editors, *4th International Symposium on Formal Methods for Components and Objects (FMCO'05)*, volume 4111 of *LNCS*, pages 115–137. Springer, 2006. Cited on page 47.

[BHKR13] Marcello M. Bonsangue, Helle H. Hansen, Alexander Kurz, and Jurriaan Rot. Presenting distributive laws. In *Proceedings of the 5th Conference on Algebra and Coalgebra in Computer Science (CALCO'13)*, volume 8089 of *LNCS*, pages 95–109. Springer, 2013. Cited on page 115.

[BHLP09] Yves Bertot, Gérard Huet, Jean-Jacques Lévy, and Gordon Plotkin, editors. *Theorem proving support in programming language semantics*, chapter 15, pages 337–361. Cambridge University Press, 2009. Cited on pages 58 and 68.

[BHS07] Bernhard Beckert, Reiner Hähnle, and Peter H. Schmitt, editors. *Verification of Object-Oriented Software: The KeY Approach*, volume 4334 of *LNCS*. Springer, 2007. Cited on page 5.

[BIM95] Bard Bloom, Sorin Istrail, and Albert R. Meyer. Bisimulation can't be traced. *Journal of the ACM*, 42(1):232–268, 1995. Cited on pages 3 and 68.

[BMSZ14] Filippo Bonchi, Stefan Milius, Alexandra Silva, and Fabio Zanasi. How to kill epsilons with a dagger – a coalgebraic take on systems with algebraic label structure. *CoRR*, abs/1402.4062, 2014. Cited on page 115.

[Bor00] Richard Bornat. Proving pointer programs in Hoare logic. In Roland Backhouse and José Oliveira, editors, *5th International*

Conference on Mathematics of Program Construction (MPC'00), volume 1837 of *LNCS*, pages 102–126. Springer, 2000. Cited on pages 34 and 46.

[BP05] Javier Blanco and Castro Pablo. A semantics for proving class correctness. unpublished, 2005. Cited on page 46.

[Bur72] Rodney Burstall. Some techniques for proving correctness of programs which alter data structures. In *Machine Intelligence 7*, pages 22–50. Edinburgh University Press, 1972. Cited on pages 34 and 47.

[CB07] Fabricio Chalub and Christiano Braga. Maude MSOS Tool. *Electronic Notes in Theoretical Computer Science*, 176(4):133–146, 2007. Cited on page 64.

[CKS07] Pietro Cenciarelli, Alexander Knapp, and Eleonora Sibilio. The Java memory model: Operationally, denotationally, axiomatically. In Rocco De Nicola, editor, *Proceedings of the 16th European Symposium on Programming (ESOP'07)*, volume 4421 of *LNCS*, pages 331–346. Springer, 2007. Cited on page 53.

[CM13] Martin Churchill and Peter D. Mosses. Modular bisimulation theory for computations and values. In Frank Pfenning, editor, *Proceedings of the 16th International Conference on Foundations of Software Science and Computation Structures (FoSSaCS'13)*, volume 7794 of *LNCS*, pages 97–112. Springer, 2013. Cited on pages 54, 92, and 115.

[DCB11] Benjamin Delaware, William Cook, and Don Batory. Product lines of theorems. *ACM SIGPLAN Notices*, 46(10):595–608, 2011. Cited on pages 55 and 64.

[DDH72] Ole-Johan Dahl, Edsger W. Dijkstra, and C.A.R. Hoare. *Structured programming*. Academic Press, 1972. Cited on page 1.

[DKSO13] Benjamin Delaware, Steven Keuchel, Tom Schrijvers, and Bruno C.d.S. Oliveira. Modular monadic meta-theory. In *Proceedings of the 18th ACM SIGPLAN international conference on Functional programming (ICFP'13)*, pages 319–330. ACM, 2013. Cited on page 64.

[DMvEP08] Maarten De Mol, Marko van Eekelen, and Rinus Plasmeijer. Proving properties of lazy functional programs with Sparkle. In

Zoltán Horváth, editor, *Revised and selected lectures of the 2nd Central European Functional Programming School (CEFP'08)*, volume 5161 of *LNCS*, pages 41–86. Springer, 2008. Cited on page 6.

[DOS13] Benjamin Delaware, Bruno C.d.S. Oliveira, and Tom Schrijvers. Meta-theory à la carte. *ACM SIGPLAN Notices*, 48(1):207–218, 2013. Cited on page 64.

[DS85] Robert De Simone. Higher-level synchronising devices in Meije-SCCS. *Theoretical Computer Science*, 37:245–267, 1985. Cited on pages 89, 109, and 115.

[FM07] Jean Christophe Filliâtre and Claude Marché. The Why/Krakatoa/Caduceus platform for deductive program verification. In Werner Damm and Holger Hermanns, editors, *Proceedings of the 19th International Conference on Computer Aided Verification (CAV'07)*, volume 4590 of *LNCS*, pages 173–177. Springer, 2007. Cited on pages 34 and 47.

[FPT99] Marcelo Fiore, Gordon D. Plotkin, and Daniele Turi. Abstract syntax and variable binding. In *Proceedings of the 14th Symposium on Logic in Computer Science (LICS'99)*, pages 193–202. IEEE, 1999. Cited on page 86.

[HHM+10] Thomas C. Hales, John Harrison, Sean McLaughlin, Tobias Nipkow, Steven Obua, and Roland Zumkeller. A revision of the proof of the Kepler conjecture. *Discrete and Computational Geometry*, 44:1–34, 2010. Cited on page 6.

[HJ99] C.A.R. Hoare and He Jifeng. A trace model for pointers and objects. In Rachid Guerraoui, editor, *Proceedings of the 13th European Conference on Object-Oriented Programming (ECOOP'99)*, volume 1628 of *LNCS*, pages 1–17. Springer, 1999. Cited on page 46.

[HJ11] Ralf Hinze and Daniel W.H. James. Proving the unique fixed-point principle correct: An adventure with category theory. In *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming (ICFP'11)*, pages 359–371. ACM, 2011. Cited on pages 68, 85, and 86.

[HK11] Helle Hvid Hansen and Bartek Klin. Pointwise extensions of GSOS-defined operations. *Mathematical Structures in Computer Science*, 21(2):321–361, 2011. Cited on page 116.

[HM07] Thierry Hubert and Claude Marché. Separation analysis for deductive verification. In *Workshop on Heap Analysis and Verification (HAV'07)*, pages 81–93, 2007. Cited on pages 34, 45, and 47.

[Hol97] Gerard J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, 1997. Cited on pages 14, 23, and 27.

[Hol06] Gerard J. Holzmann. The power of 10: Rules for developing safety-critical code. *IEEE Computer*, 39(6):95–97, 2006. Cited on pages 18 and 28.

[Hui01] Marieke Huisman. *Reasoning about Java Programs in Higher Order Logic with PVS and Isabelle*. PhD thesis, University of Nijmegen, 2001. Cited on page 6.

[Hur10] Chung-Kil Hur. Heq: A Coq library for heterogeneous equality. Informal presentation at the 2nd Coq workshop, 2010. Cited on page 62.

[Hut98] Graham Hutton. Fold and unfold for program semantics. In *Proceedings of the 3rd ACM SIGPLAN International Conference on Functional Programming (ICFP'98)*, pages 280–288. ACM, 1998. Cited on pages 68 and 85.

[IEC96] IEC. Functional safety: Safety related systems, International Standard IEC 61508, International Electrotechnical Commission, Geneva, Switzerland, 1996. Cited on page 14.

[Jac12] Bart Jacobs. *Introduction to coalgebra: Towards mathematics of states and observations*. In preparation, version 2.0. 2012. `http://www.cs.ru.nl/B.Jacobs/CLG/JacobsCoalgebraIntro.pdf`. Cited on pages 90 and 91.

[JGH11] Mauro Jaskelioff, Neil Ghani, and Graham Hutton. Modularity and implementation of mathematical operational semantics. *Electronic Notes in Theoretical Computer Science*, 229(5):75–95, 2011. Cited on pages 68, 85, and 115.

[Jia94] Xiaoping Jia. ZTC: A Type Checker for Z – User's Guide. *Institute for Software Engineering, Department of Computer Science and Information Systems, DePaul University, Chicago, USA*, 1994. Cited on page 14.

[Joy10] Jeff Joyce. Use of machine-assisted theorem-proving as a means of verifying critical software in the context of RTCA DO-178C. In Workshop on Theorem Proving in Certification, December 6–7, 2010, Cambridge, UK, 2010. Cited on page 7.

[JP08] Bart Jacobs and Frank Piessens. The VeriFast program verifier. Technical Report CW-520, Katholieke Universiteit Leuven, Belgium, 2008. Cited on page 47.

[JR97] Bart Jacobs and Jan Rutten. A tutorial on (co)algebras and (co)induction. *EATCS Bulletin*, 62:62–222, 1997. Cited on page 71.

[JSGB98] Bill Joy, Guy Steele, James Gosling, and Gilad Bracha. *The Java Language Specification*. Addison-Wesley, 1998. Cited on page 3.

[JVDBH+98] Bart Jacobs, Joachim Van Den Berg, Marieke Huisman, Martijn van Berkum, Ulrich Hensel, and Hendrik Tews. Reasoning about Java classes (preliminary report). *ACM SIGPLAN Notices*, 33(10):329–340, 1998. Cited on pages 5, 6, and 36.

[Kar96] Pim Kars. The application of Promela and SPIN in the BOS project. In Jean-Charles Grégoire, Gerard J. Holzmann, and Doron A. Peled, editors, *Proceedings of the 2nd Workshop on the SPIN Verification System (SPIN'96)*, volume 32 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science, Rutgers University*, 1996. Cited on page 27.

[Kli04] Bartek Klin. Adding recursive constructs to bialgebraic semantics. *Journal of Logic and Algebraic Programming*, 60:259–286, 2004. Cited on pages 89, 102, 104, 105, 108, 110, and 115.

[Kli11] Bartek Klin. Bialgebras for structural operational semantics: An introduction. *Theoretical Computer Science*, 412(38):5043–5069, 2011. Cited on pages 68, 69, 80, 81, 85, and 90.

[KSM13] Matt Kaufmann and J. Strother Moore. *ACL2 Version 6.1*, 2013. `http://www.cs.utexas.edu/users/moore/acl2/`. Cited on page 5.

[Ler09] Xavier Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, 2009. Cited on pages 6 and 28.

[LHJ95] Sheng Liang, Paul Hudak, and Mark Jones. Monad transformers and modular interpreters. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'95)*, pages 333–343. ACM, 1995. Cited on page 4.

[LPW00] Marina Lenisa, John Power, and Hiroshi Watanabe. Distributivity for endofunctors, pointed and co-pointed endofunctors, monads and comonads. *Electronic Notes in Theoretical Computer Science*, 33:230–260, 2000. Not cited.

[LPW04] Marina Lenisa, John Power, and Hiroshi Watanabe. Category theory for operational semantics. *Theoretical Computer Science*, 327(1-2):135–154, 2004. Cited on page 84.

[LSVE08] Leonard Lensink, Sjaak Smetsers, and Marko Van Eekelen. Machine checked formal proof of a scheduling protocol for smartcard personalization. In Stefan Leue and Pedro Merino, editors, *Proceedings of the 12th International Workshop on Formal Methods for Industrial Critical Systems (FMICS'07)*, volume 4916 of *LNCS*, pages 115–132. Springer, 2008. Cited on page 2.

[Mey03] Bertrand Meyer. Towards practical proofs of class correctness. In Didier Bert, Jonathan P. Bowen, Steve King, and Marina Waldén, editors, *Proceedings of the 3rd International Conference of B and Z Users (ZB'03)*, volume 2651 of *LNCS*, pages 359–387. Springer, 2003. Cited on pages 9, 31, 32, 34, 35, and 46.

[MFP91] Erik Meijer, Maarten Fokkinga, and Ross Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In John Hughes, editor, *Proceedings of the 5th ACM Conference on Functional Programming Languages and Computer Architecture (FPCA'91)*, volume 523 of *LNCS*, pages 124–144. Springer, 1991. Cited on page 74.

[MMR10] Peter D. Mosses, Mohammad Reza Mousavi, and Michel A. Reniers. Robustness of equations under operational extensions. In Sibylle Fröschle and Frank D. Valencia, editors, *Proceedings of the 17th International Workshop on Expressiveness in Concurrency*

(EXPRESS'10), EPTCS, pages 106–120, 2010. Cited on pages 89, 109, 111, and 115.

[MN05] Farhad Mehta and Tobias Nipkow. Proving pointer programs in higher-order logic. *Information and Computation*, 199:200–227, 2005. Cited on pages 45 and 46.

[MN09] Peter D. Mosses and Mark J. New. Implicit propagation in structural operational semantics. *Electronic Notes in Theoretical Computer Science*, 229(4):49–66, 2009. Cited on pages 64 and 65.

[Mog89] Eugenio Moggi. An abstract view of programming languages. Technical Report ECS-LFCS-90-113, University of Edinburgh, 1989. Cited on page 4.

[Mos04] Peter D. Mosses. Modular structural operational semantics. *Journal of Logic and Algebraic Programming*, 60:195–228, 2004. Cited on pages 4, 50, 54, 60, and 65.

[Mos06] Peter D. Mosses. Teaching semantics of programming languages with Modular SOS. In *Teaching Formal Methods: Practice and Experience*, Electronic Workshops in Computing. BCS, 2006. Cited on page 64.

[Mos09] Peter D. Mosses. Component-based semantics. In *Proceedings of the 8th International Workshop on Specification and Verification of Component-Based Systems (SAVCBS'09)*, pages 3–10. ACM, 2009. Cited on pages 9, 49, 50, 52, and 88.

[MPM05] Claude Marché and Christine Paulin-Mohring. Reasoning about Java programs with aliasing and frame conditions. In Joe Hurd and Tom Melham, editors, *Proceedings of the 18th International Conference on Theorem Proving in Higher Order Logics (TPHOLs'05)*, volume 3603 of *LNCS*, pages 179–194. Springer, 2005. Cited on page 47.

[MRG07] Mohammad Reza Mousavi, Michel A. Reniers, and Jan Friso Groote. SOS formats and meta-theory: 20 years after. *Theoretical Computer Science*, 373(3):238–272, 2007. Cited on page 3.

[MS13] Ken Madlener and Sjaak Smetsers. GSOS formalized in Coq. In *Proceedings of the 7th International Symposium on Theoretical Aspects of Software Engineering (TASE'13)*, pages 199–206. IEEE, 2013. Cited on pages 10, 88, and 116.

[MSvE10] Ken Madlener, Sjaak Smetsers, and Marko van Eekelen. A formal verification study on the Rotterdam storm surge barrier. In Jin Song Dong and Huibiao Zhu, editors, *Proceedings of the 12th International Conference on Formal Engineering Methods (IFCEM'10)*, volume 6447 of *LNCS*, pages 287–302. Springer, 2010. Cited on page 8.

[MSvE11] Ken Madlener, Sjaak Smetsers, and Marko van Eekelen. Formal component-based semantics. In Michel A. Reniers and Pawel Sobocinski, editors, *Proceedings of the 8th Workshop on Structural Operational Semantics (SOS'11)*, volume 62 of *Electronic Proceedings in Theoretical Computer Science*, pages 17–29, 2011. Cited on page 9.

[MSvE13] Ken Madlener, Sjaak Smetsers, and Marko van Eekelen. Modular bialgebraic semantics and algebraic laws. In *Proceedings of the 17th Brazilian Symposium on Programming Languages (SBLP'13)*, volume 8129 of *LNCS*, pages 46–60. Springer, 2013. Cited on pages 11 and 113.

[Niq09] Milad Niqui. Coalgebraic reasoning in Coq: Bisimulation and the λ-coiteration scheme. In *Proceedings of TYPES'08*, volume 5497 of *LNCS*, pages 272–288. Springer, 2009. Cited on page 85.

[NU10] Keiko Nakata and Tarmo Uustalu. A Hoare logic for the coinductive trace-based big-step semantics of While. In Andrew D. Gordon, editor, *Proceedings of the 19th European Symposium on Programming (ESOP'10)*, volume 6012 of *LNCS*, pages 488–506. Springer, 2010. Cited on page 116.

[Ohe01] David von Oheimb. *Analyzing Java in Isabelle/HOL: Formalization, Type Safety and Hoare Logic*. PhD thesis, Technische Universität München, 2001. Cited on page 6.

[ORS92] Sam Owre, John M. Rushby, and Natarajan Shankar. PVS: A prototype verification system. In Deepak Kapur, editor, *Proceedings of the 11th International Conference on Automated Deduction (CADE'92)*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752. Springer, 1992. Cited on pages 5 and 15.

[OSRS01] Sam Owre, Natarajan Shankar, John M. Rushby, and Dave Stringer-Calvert. PVS language reference (version 2.4). Technical

report, Computer Science Laboratory, SRI International, 2001. Cited on page 32.

[Owe08] Scott Owens. A sound semantics for OCaml light. In Sophia Drossopoulou, editor, *Proceedings of the 17th European Symposium on Programming (ESOP'08)*, volume 4960 of *LNCS*, pages 1–15. Springer, 2008. Cited on page 64.

[Owr06] Sam Owre. Random testing in PVS. In *Workshop on Automated Formal Methods*, Seattle, USA, 2006. `http://fm.csl.sri.com/AFM06/papers/5-Owre.pdf`. Cited on page 25.

[Pau94] Lawrence C. Paulson. *Isabelle: a generic theorem prover*, volume 828 of *LNCS*. Springer, 1994. Cited on page 6.

[Plo81] Gordon D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Aarhus University, 1981. Reprinted in *Journal Logic and Algebraic Programming* 60–61:17–139, 2004. Cited on page 2.

[PP02] Gordon Plotkin and John Power. Notions of computation determine monads. In Mogens Nielsen and Uffe Engberg, editors, *Proceedings of the 5th International Conference on Foundations of Software Science and Computation Structures (FOSSACS'02)*, volume 2303 of *LNCS*, pages 342–356. Springer, 2002. Cited on page 4.

[RBN10] Stan Rosenberg, Anindya Banerjee, and David A. Naumann. Local reasoning and dynamic framing for the composite pattern and its clients. In Gary T. Leavens, Peter O'Hearn, and Sriram K. Rajamani, editors, *Proceedings of the 3rd international Conference on Verified software: Theories, Tools, Experiments (VSTTE'10)*, volume 6217 of *LNCS*, pages 183–198. Springer, 2010. Cited on pages 32 and 47.

[Ren00] Arend Rensink. Bisimilarity of open terms. *Information and Computation*, 156(1):345–385, 2000. Cited on pages 89 and 113.

[Rey02] John Reynolds. Separation logic: A logic for shared mutable data structures. In *Proceedings of the 17th Symposium on Logic in Computer Science (LICS'02)*, pages 55–74. IEEE, 2002. Cited on pages 32, 45, and 47.

[Ruy01] Theo Ruys. *Towards Effective Model Checking*. PhD thesis, University of Twente, 2001. Cited on page 27.

[Saa97] Mark Saaltink. The Z/Eves system. In Jonathan P. Bowen, Michael G. Hinchey, and David Till, editors, *Proceedings of the 10th International Conference of Z Users (ZUM'97)*, volume 1212 of *LNCS*, pages 72–88. Springer, 1997. Cited on page 19.

[Sco70] Dana S. Scott. Outline of a mathematical theory of computation. Technical Report PRG-2, Oxford University Computing Laboratory, 1970. Cited on page 3.

[Slo99] Oscar Slotosch. Overview over the project quest. In *Proceedings of the International Workshop on Current Trends in Applied Formal Methods (FM-Trends'98)*, volume 1641 of *LNCS*, pages 346–350. Springer, 1999. Cited on page 27.

[SO08] Matthieu Sozeau and Nicolas Oury. First-class type classes. In *Proceedings of the 21st International Conference on Theorem Proving in Higher Order Logics (TPHOLs'08)*, volume 5170 of *LNCS*, pages 278–293. Springer, 2008. Cited on pages 55 and 77.

[Spi89] J. M. Spivey. The Z notation: a reference manual. *Prentice-Hall International Series In Computer Science*, 1989. Cited on page 14.

[SS71] Dana S. Scott and Christopher Strachey. Toward a mathematical semantics for computer languages. Technical Report PRG-6, Oxford University Computing Laboratory, 1971. Cited on page 3.

[STT+09] Erik Schierboom, Alejandro N. Tamalet, Hendrik Tews, Marko van Eekelen, and Sjaak Smetsers. Preemption abstraction. In María Alpuente, Byron Cook, and Christophe Joubert, editors, *Proceedings of the 15th International Workshop on Formal Methods for Industrial Critical Systems (FMICS'09)*, volume 5825 of *LNCS*, pages 149–164. Springer, 2009. Cited on pages 2 and 27.

[SvdW11] Bas Spitters and Eelis van der Weegen. Type classes for mathematics in type theory. *Mathematical Structures in Computer Science*, 21:1–31, 2011. Cited on pages 61 and 78.

[SW13] Alexandra Silva and Bram Westerbaan. A coalgebraic view of ε-transitions. In Reiko Heckel and Stefan Milius, editors,

Proceedings of the 5th Conference on Algebra and Coalgebra in Computer Science (CALCO'13), volume 8089 of *LNCS*, pages 267–281. Springer, 2013. Cited on page 115.

[Tam06] Alejandro N. Tamalet. Yet another semantics for proving class correctness. Master's thesis, Universidad Nacional de Rosario, Argentina, 2006. Cited on pages 9 and 46.

[The12] The Coq Development Team. *The Coq Proof Assistant Reference Manual for Version 8.4*, 2012. `http://coq.inria.fr`. Cited on pages 5, 50, and 68.

[TM10] Alejandro N. Tamalet and Ken Madlener. Reasoning about assignments in recursive data structures. In Jim Davies, Leila Silva, and Adenilso da Silva Simão, editors, *Proceedings of the 13th Brazilian Symposium on Formal Methods (SBMF'10)*, volume 6527 of *LNCS*, pages 161–176. Springer, 2010. Cited on page 9.

[TP97] Daniele Turi and Gordon D. Plotkin. Towards a mathematical operational semantics. In *Proceedings of the 12nd Symposium on Logic in Computer Science (LICS'97)*, pages 280–291. IEEE, 1997. Cited on pages 3, 67, 69, 72, 81, 85, 88, 89, 91, 99, 114, 115, and 116.

[Tue09] Thomas Tuerk. A formalisation of Smallfoot in HOL. In Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel, editors, *Proceedings of the 22nd International Conference on Theorem Proving in Higher Order Logics (TPHOLs'09)*, volume 5674 of *LNCS*, pages 469–484. Springer, 2009. Cited on page 47.

[TWC01] Jan Tretmans, Klaas Wijbrans, and Michel Chaudron. Software engineering with formal methods: The development of a storm surge barrier control system revisiting seven myths of formal methods. *Formal Methods in System Design*, 19(2):195–215, 2001. Cited on pages 14 and 27.

[vdBJ01] Joachim van den Berg and Bart Jacobs. The LOOP compiler for Java and JML. In Tiziana Margaria and Wang Yi, editors, *Proceedings of the 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'01)*, volume 2031 of *LNCS*, pages 299–312. Springer, 2001. Cited on page 7.

[vETHSU08] Marko van Eekelen, Stefan Ten Hoedt, René Schreurs, and Yaroslav S. Usenko. Analysis of a session-layer protocol in

mCRL2. In Stefan Leue and Pedro Merino, editors, *Proceedings of the 13th International Workshop on Formal Methods for Industrial Critical Systems (FMICS'08)*, volume 4916 of *LNCS*, pages 182–199. Springer, 2008. Cited on page 2.

[vHPPR98] Friedrich W. von Henke, Stephan Pfab, Holger Pfeifer, and Harald Rueß. Case studies in meta-level theorem proving. In Jim Grundy and Malcolm C. Newey, editors, *Proceedings of the 11th International Conference on Theorem Proving in Higher Order Logics (TPHOLs'98)*, volume 1479 of *LNCS*, pages 461–478. Springer, 1998. Cited on page 46.

[VJP12] Frédéric Vogels, Bart Jacobs, and Frank Piessens. Featherweight VeriFast: Extended version. Technical Report CW-614, Katholieke Universiteit Leuven, Belgium, 2012. Cited on page 47.

[Wat02] Hiroshi Watanabe. Well-behaved translations between structural operational semantics. *Electronic Notes in Theoretical Computer Science*, 65(1):337–357, 2002. Cited on page 116.

[WBRG08] Klaas Wijbrans, Franc Buve, Robin Rijkers, and Wouter Geurts. Software engineering with formal methods: Experiences with the development of a storm surge barrier control system. In Jorge Cuellar, Tom Maibaum, and Kaisa Sere, editors, *Proceedings of the 15th International Symposium on Formal Methods (FM'08)*, volume 5014 of *LNCS*, pages 419–424. Springer, 2008. Cited on page 27.

[Win93] Glynn Winskel. *The formal semantics of programming languages: an introduction*. MIT Press, 1993. Cited on page 3.

Summary

This thesis contributes a number of techniques in formal reasoning about the correctness of computer software, a foundational subject in computer science. It does so in the context of theorem provers: software which assists with logical reasoning on a computer.

Before one can reason about the correctness of a piece of software, a formal description of its meaning is needed, also called its semantics. It is widely agreed that providing descriptions in well-known styles of semantics such as operational or denotational semantics is a difficult task when it concerns programs written in mainstream programming languages such as C, C++ or JAVA. Such languages involve a mixture of different types of computational behavior and other details which prevents concise description. One may opt to leave out some detail and thereby reason about a lightweight version of the full semantics, if the program under verification does not use all features of the language. Verification based on lightweight models may not be conclusive about the correctness of software, but can help to find issues that have escaped manual code review or a traditional software testing process.

In Chapter 2 a crucial component of BOS, a large safety-critical system that decides when to close and open the Maeslantkering, a storm surge barrier near the city of Rotterdam in the Netherlands is verified. This case-study is based on a lightweight semantics, formalized and verified in the theorem prover PVS. It uncovers several mismatches between its specification and implementation (which have now been fixed in the source code), even though the code has been subjected to a thorough review process.

Chapter 3 develops a framework to reason about how an assignment may affect a recursive data structure. A number of rules are provided that describe when and how a path in the heap is (or is not) modified by an assignment. These rules can

be used to prove the correctness of programs. A key aspect of this approach is that by applying these rules one does not need to reason inductively: the induction is encapsulated in the rules.

An active research topic is enabling scalability of programming language semantics through modularity, i.e. by obtaining it from a combination of language-independent pieces. Conversely, a modular semantics allows one to scale down to a lightweight version, which has a formal correspondence to the full semantics.

Chapter 4 proposes a technique to encode semantics in a modular fashion in theorem provers, based on a variant of operational semantics. An essential ingredient is that the rules that define the operational semantics can be described in a way independent of other rules that may or may not exist. This way, a base language can be extended with new rules to provide additional features.

Chapter 5 provides a partial formalization of bialgebraic semantics, originally proposed by Turi and Plotkin, in the theorem prover COQ. An important aspect of formalization is that it unifies operational and denotational semantics within the logic of COQ, under the condition that the models of both semantics are derived from a rule format called GSOS. This allows one to interchange these styles of semantics within COQ, thereby profiting from either of their characteristics.

Modularity in the bialgebraic framework is treated in Chapter 6. A variant of the GSOS rule format is proposed, which supports the language-independent description of operations. It is proved in this chapter that conservatively adding new operations to a language results in well-behaved translations from the base language to the extended language, that algebraic laws are preserved by language extensions, and that algebraic laws can be instantiated.

Samenvatting

Dit proefschrift contribueert een aantal technieken om formeel te redeneren over de correctheid van computer software, een fundamenteel onderwerp in de informatica. Het doet dit in de context van bewijsassistenten: computer programma's die de gebruiker in staat stellen logisch te redeneren op een computer.

Alvorens men in staat is te redeneren over de correctheid van een programma, is een formele beschrijven van het programma nodig, wat ook wel de semantiek van het programma wordt genoemd. Men is het er over eens dat het geven van beschrijvingen in gangbare stijlen zoals operationele en denotationele semantiek een complexe taak is, als het gaat om programma's geschreven in veelgebruikte talen zoals C, C++ of JAVA. Zulk soort talen maken gebruik van combinaties van verschillende soorten computationeel gedrag of er zijn andere details die het lastig maken een elegante, beknopte beschrijving te geven. Men kan er voor kiezen de beschrijving te vereenvoudigen en daarmee te redeneren over een lichtgewicht versie van de volledige semantiek, indien het programma dat wordt geverifieerd niet gebruik maakt van alle attributen van de programmeertaal. Verificatie gebaseerd op lichtgewicht modellen stelt de gebruiker niet altijd in staat het bestaan van fouten volledig uit te sluiten, maar kan wel helpen fouten te vinden die aan een handmatige revisie van de programmacode of een traditioneel software testproces zijn ontglipt.

In hoofdstuk 2 wordt een cruciaal component van BOS (Beslis en Ondersteunend Systeem) geverifieerd. Dit is een groot kritiek systeem dat beslist wanneer de Maeslantkering moet worden geopend of gesloten. De Maeslantkering is een stormvloedkering gebouwd in de omgeving van Rotterdam. Deze case-study is gebaseerd op een lichtgewicht semantiek die is geformaliseerd in de bewijsassistent PVS. In de case-study zijn een aantal punten ontdekt waarop de specificatie verschilt van de implementatie (deze verschillen zijn nu gerepareerd in

de broncode). Opmerkelijk is dat de broncode was onderworpen aan een grondige herziening.

Hoofdstuk 3 ontwikkelt een raamwerk om te redeneren over hoe toekenningen (van bijvoorbeeld waarden aan variabelen) invloed kunnen hebben op een recursieve datastructuur. Een aantal regels worden voorgesteld die beschrijven hoe een pad dat wordt doorlopen van pointers in het geheugen al dan niet wordt veranderd door een toekenning. Deze regels kunnen worden gebruikt om de correctheid van een programma te bewijzen. Een belangrijk aspect van de gebruikte benadering is dat bij het toepassen niet wordt geredeneerd met behulp van inductie; de inductie is ingekapseld in de regels zelf.

Een actief onderzoeksonderwerp is de schaalbaarheid van semantiek van programmeertalen door middel van modulariteit. In andere woorden, het creëren van schaalbaarheid door de semantiek als een combinatie van taal-onafhankelijke delen te beschrijven wordt actief onderzocht. Andersom kan een modulaire semantiek worden gebruikt om terug te schalen naar een lichtgewicht semantiek, welke een formele correspondentie heeft met het volledige model.

In hoofdstuk 4 wordt een techniek beschreven die men in staat stelt semantiek op een modulaire manier te formaliseren in een bewijsassistent, gebaseerd op een variant van operationele semantiek. Een belangrijk ingrediënt is dat de regels die de operationele semantiek definiëren, onafhankelijk van welke andere regels al dan niet bestaan, kunnen worden beschreven. Dit stelt de gebruiker in staat een programmeertaal uit te breiden.

Een partiële formalisatie van bialgebraïsche semantiek in de bewijsassistent COQ wordt ontwikkeld in hoofdstuk 5. Bialgebraïsche semantiek zelf is voorgesteld door Turi en Plotkin. Een belangrijk aspect van de formalisatie is dat het operationele en denotationele semantiek verenigt binnen de logica van COQ, onder de voorwaarde dat de modellen van beide stijlen van semantiek zijn afgeleid van een verzameling regels die zijn gecodeerd in het GSOS formaat. Deze formalisatie stelt de gebruiker in staat beide stijlen onderling uit te wisselen binnen COQ, en daarmee voordeel te doen van de karakteristieken die beide stijlen bevatten.

Modulariteit in het bialgebraïsche raamwerk wordt behandeld in hoofdstuk 6. Een variant van het GSOS regel-formaat wordt voorgesteld die ondersteuning biedt voor het taal-onafhankelijk beschrijven van operaties. In dit hoofdstuk wordt bewezen dat het toevoegen van conservatieve nieuwe operaties aan een taal resulteert in afbeeldingen tussen de basis-taal en zijn uitbreiding die zich goed gedragen. Daarnaast wordt bewezen dat algebraïsche wetten worden behouden door taal uitbreidingen en dat algebraïsche wetten instantieerbaar zijn.

Curriculum Vitae

Ken Madlener was born on the 24th of March in Zevenaar, the Netherlands. He started his studies in informatics in 2000 at the Avans University of Applied Sciences in 's-Hertogenbosch. After obtaining his bachelor's degree in 2004, he continued his studies at the Radboud University Nijmegen in computer science, and obtained a master's degree in 2008. From 2006 to 2008 he also studied mathematics at the Radboud University Nijmegen, for which he obtained a bachelor's degree in 2008. In September 2008, Ken started as a Ph.D. student at the Institute for Computing and Information Sciences, at the Radboud University Nijmegen, where he was supervised by prof. dr. Marko van Eekelen and dr. Sjaak Smetsers. The topic of his Ph.D. thesis is formal reasoning about the correctness of computer software.