Reasoning about Functional Programs

*Sparkle, a proof assistant for Clean*

Maarten de Mol
Copyright © 2009 Maarten de Mol
All rights reserved

IPA dissertation series 2009-02
NUR-code: 993

Typeset with \LaTeX\textsuperscript{2ε}
Printed by PrintPartners Ipskamp

The work in this thesis has been carried out under the auspices of the research school IPA (Institute for Programming research and Algorithmics).
Reasoning about Functional Programs

Sparkle, a proof assistant for Clean

Een wetenschappelijke proeve of het gebied van de
Natuurwetenschappen, Wiskunde en Informatica

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 woensdag 4 maart 2009
om 15:30 uur precies

door

Maarten Jaap de Mol

geboren op 24 mei 1976

te Velp
Promotor:
prof. dr. ir. Rinus Plasmeijer

Copromotor:
dr. Marko van Eekelen

Manuscript commissie:
prof. dr. Herman Geuvers

prof. dr. Rex L. Page University of Oklahoma
(Norman, Oklahoma, USA)

prof. dr. Zoltán Horváth Eötvös Loránd University
(Budapest, Hungary)
Preface

It already seems ages ago that I started the research of my PhD thesis. It has in fact been over ten years, including a break of two years for a research project for the Royal Netherlands Navy, in which working on SPARKLE has become part of my daily life. Although the original plan was to finish in 4.5 years, I have really enjoyed all of my stay at the Radboud University Nijmegen. It has been a pleasant environment to work in, and it is has been a privilege to be able to do research (and implementation) on my own ideas. Writing down the results of research, however, has never come easy to me. Still, I have now finally finished my thesis, and I hope everyone will enjoy reading it.

I would like to thank the following people for their support. I thank Rinus and Marko for their continuous positive guidance, especially at the times that I was not making any significant progress. In particular, I could not have finished this thesis without Marko’s persistent suggestion to concentrate on individual papers, instead of trying to write everything down in one go. I thank Peter and Diederik for teaching me how to use the Object I/O, John for explaining the source code of the CLEAN-compiler, and Bas for helping with the front cover. I thank all of the staff of (the now former) ST for providing a pleasant atmosphere to work in, and Peter and Martijn for all of our (off-topic) lunch conversations. Finally, I thank my family and my friends for their mental support.
# Contents

<table>
<thead>
<tr>
<th>Preface</th>
<th>v</th>
</tr>
</thead>
<tbody>
<tr>
<td>Contents</td>
<td>vii</td>
</tr>
</tbody>
</table>

## 1 Introduction

1.1 Software bugs ............................... 1
1.2 Prevent vs Cure .................................. 2
1.3 Detecting software bugs .......................... 2
1.4 Formal reasoning ............................... 4
1.5 Soundness of formal reasoning ...................... 5
1.6 Characteristics of formal reasoning ................. 5
1.7 Research setting ................................ 6
1.8 Contents of this thesis: SPARKLE ................. 7
   1.8.1 Description of SPARKLE (Chapters 2,3,4; App. A) .... 8
   1.8.2 Reasoning with strictness (Chapters 5,6) ............... 10
   1.8.3 Theoretical background of SPARKLE (Chapters 7,8) .... 11
   1.8.4 Applications of SPARKLE (Chapters 9,10) .............. 12
1.9 Impact of SPARKLE ............................. 13
1.10 Applicability of SPARKLE ....................... 13

## 2 A Proof Tool Dedicated to CLEAN

2.1 Introduction ................................... 15
2.2 Restrictions of the first prototype .................. 16
2.3 Using the prototype to construct proofs ............... 17
   2.3.1 Specification of the program ....................... 17
   2.3.2 Specification of the theorem ....................... 17
   2.3.3 Building a proof ................................ 18
   2.3.4 Automatic proof construction ...................... 19
   2.3.5 Examples of proven theorems and proofs ............. 20
   2.3.6 Upgrading the prototype: further work ............... 20
   2.3.7 Example proof with graphs and co-induction .......... 21
2.4 Conclusions and related work ....................... 22
# Table of Contents

## 3 Theorem Proving for Functional Programmers

3.1 Introduction .................................................. 23
3.2 The specification language of Sparkle .......................... 24
  3.2.1 The Clean program ........................................ 24
  3.2.2 Simplification for Sparkle .................................. 25
  3.2.3 Specification in other theorem provers ......................... 26
  3.2.4 Suitability of Core-Clean for reasoning about Clean .......... 27
3.3 The specification of the property .............................. 28
3.4 Building the proof ........................................... 29
  3.4.1 Reasoning style in Sparkle .................................. 29
  3.4.2 The hint mechanism .......................................... 30
  3.4.3 Proof of the example program ................................ 30
  3.4.4 Remarks concerning the reasoning process ...................... 35
3.5 Conclusions and further work .................................. 36
3.6 Related work ................................................ 36

## 4 Proving properties of lazy functional programs with Sparkle

4.1 Introduction .................................................. 39
4.2 Formal reasoning ............................................ 40
  4.2.1 The abstract process of formal reasoning ...................... 41
  4.2.2 Formal framework ........................................... 41
  4.2.3 Proof language ............................................. 42
  4.2.4 Soundness of formal reasoning ................................ 43
4.3 Design principles of Sparkle ................................... 43
  4.3.1 Intended users: functional programmers ...................... 44
  4.3.2 Design choices ............................................. 44
  4.3.3 Dedicated vs general-purpose formal reasoning ................ 46
  4.3.4 Sparkle’s approximation of dedicated reasoning .............. 47
4.4 Tutorial part I: getting started with Sparkle .................... 48
  4.4.1 Loading a program ........................................... 48
  4.4.2 Undefinedness in Clean and Core-Clean ....................... 50
  4.4.3 Stating a property .......................................... 51
  4.4.4 The meaning of properties .................................... 54
  4.4.5 Reasoning style in Sparkle .................................. 56
  4.4.6 Proving a simple property ................................... 56
4.5 Tutorial part II: specialized Sparkle-features .................... 61
  4.5.1 The influence of sharing on reasoning ........................ 62
  4.5.2 Definedness conditions in properties ......................... 63
  4.5.3 Specialized behavior of extensionality ....................... 66
  4.5.4 Specialized behavior of induction ........................... 66
  4.5.5 Definedness analysis and the ‘definedness’ tactic ............. 68
  4.5.6 Specialized behavior of reduction ........................... 69
  4.5.7 Property specification in Clean .............................. 70
4.6 Related Work ................................................ 72
4.7 Conclusions .................................................. 73
# Table of Contents

## 5 Proof Tool Support for Explicit Strictness

5.1 Introduction ................................................. 75
5.2 The Concept of Explicit Strictness ......................... 76
  5.2.1 When Strictness is not an option but a must ............. 77
  5.2.2 Dramatic Case of the Influence of Explicit Strictness .... 78
  5.2.3 Modeling Explicit Strictness in Formal Semantics ....... 79
5.3 Reasoning in the Context of Strictness ...................... 81
  5.3.1 Strictness and Logical Properties ..................... 81
  5.3.2 Strictness and Formal Reasoning .................... 83
5.4 Support for Explicit Strictness in Sparkle ................. 84
  5.4.1 The ‘Definedness’ Tactic of Sparkle ................... 85
  5.4.2 The ‘Reduce’ Tactic of Sparkle ..................... 87
  5.4.3 Using the ‘eval’ Function for Definedness Conditions .. 89
5.5 Related Work .............................................. 91
5.6 Conclusions / Future Work .................................. 92

## 6 Proving Lazy Folklore with Mixed Lazy/Strict Semantics

6.1 Introduction and motivation .................................. 96
6.2 Mixed lazy/strict graph semantics .......................... 99
  6.2.1 Basic idea of Launchbury’s natural graph semantics ... 99
  6.2.2 Notational conventions .................................. 100
  6.2.3 Mixed lazy/strict expressions ............................... 101
  6.2.4 Definition of mixed lazy/strict graph semantics ......... 101
  6.2.5 Correctness and Computational Adequacy .................. 103
6.3 Relation to lazy semantics .................................... 103
6.4 Example proofs with mixed semantics ......................... 106
6.5 Related work .............................................. 108
6.6 Conclusions .................................................. 108

## 7 A Single-Step Term-Graph Reduction System

7.1 Introduction .............................................. 109
7.2 Desired level of flexibility ................................ 111
  7.2.1 Granularity of reduction steps ............................ 111
  7.2.2 Choice of redex ........................................ 111
7.3 The expression language ...................................... 112
7.4 Reduction System .......................................... 113
  7.4.1 Graphs as self-contained expressions .................... 114
  7.4.2 The reduction rules for applications ...................... 114
  7.4.3 The reduction rules for let lifting ....................... 115
  7.4.4 The reduction rule for unsharing ......................... 117
  7.4.5 Head reduction ......................................... 117
  7.4.6 Locations ............................................ 118
  7.4.7 Inner reduction ........................................ 119
7.5 Correctness of let lifting ................................... 120
7.6 Confluence .................................................. 121
7.7 Related work .............................................. 123
# Table of Contents

## 7.8 Conclusions  

## 8 Semantics of Expression Equality  
8.1 Introduction  
8.2 Used definitions from the full framework  
8.3 Expression equality in CORE: an overview  
8.4 External behavior of CLEAN programs  
8.5 Internal behavior of CLEAN programs  
8.6 Behavior of CORE programs  
  8.6.1 Executable programs in CORE  
  8.6.2 Execution within CORE  
  8.6.3 Observation in CORE  
  8.6.4 Equivalence of output sets in CORE  
  8.6.5 Expression equality in CORE  
8.7 Examples of expression equalities  
8.8 Characteristics of equality: reducibility  
8.9 Conclusions  

## 9 Proof Support for General Type Classes  
9.1 Introduction  
  9.1.1 Type classes  
  9.1.2 Contributions  
  9.1.3 Outline  
9.2 Sparkle  
9.3 Preliminaries  
  9.3.1 Instance definitions  
  9.3.2 Evidence values  
  9.3.3 Class constrained properties  
9.4 Structural induction  
9.5 Induction on instances  
  9.5.1 Proof rule and tactic  
  9.5.2 Results  
9.6 Multiple class constraints  
  9.6.1 Proof rule and tactic  
  9.6.2 Results  
9.7 Implementation  
9.8 Related and future work  
9.9 Conclusion  

## 10 A Common Arrow Based Semantics for GEC and iData  
10.1 Introduction  
10.2 The GEC and iData Toolkits  
  10.2.1 Example: Variable Sum List in GEC and iData  
  10.2.2 Abstractions towards Editor Arrows  
  10.2.3 Example: Variable Sum List in Arrow style  
10.3 Editor Arrows  

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>10.3.1 Denotational Semantics for Editor Arrows</td>
<td>174</td>
</tr>
<tr>
<td>10.3.2 Operational Semantics for Editor Arrows</td>
<td>178</td>
</tr>
<tr>
<td>10.4 Properties of Editor Arrows</td>
<td>180</td>
</tr>
<tr>
<td>10.4.1 Iterate Laws</td>
<td>181</td>
</tr>
<tr>
<td>10.4.2 Eventloop Properties</td>
<td>181</td>
</tr>
<tr>
<td>10.4.3 Editor Laws</td>
<td>181</td>
</tr>
<tr>
<td>10.4.4 Definedness Laws</td>
<td>183</td>
</tr>
<tr>
<td>10.5 Programming with Editor Arrows</td>
<td>184</td>
</tr>
<tr>
<td>10.5.1 Base editor arrows in the <code>EditorArrowCore</code> library</td>
<td>185</td>
</tr>
<tr>
<td>10.5.2 Derived editor arrows in the <code>EditorArrowCore</code> library</td>
<td>190</td>
</tr>
<tr>
<td>10.5.3 Some Small Editor Arrows Programs</td>
<td>192</td>
</tr>
<tr>
<td>10.5.4 Arrow laws for Sparkle</td>
<td>193</td>
</tr>
<tr>
<td>10.6 Related Work</td>
<td>195</td>
</tr>
<tr>
<td>10.7 Conclusions</td>
<td>195</td>
</tr>
</tbody>
</table>

A Tactic Library of Sparkle 197

Summary 209

Samenvatting 211

Curriculum Vitae 215

Bibliography 217
Chapter 1

Introduction

In the following sections, an introductory overview will be presented of the scope and subject of my research. The topics of program correctness, formal reasoning and SPARKLE will be introduced, and the contents of the remainder of this thesis will be summarized.

1.1 Software bugs

Computers and computer software have become an integral part of modern day society. Unfortunately, the same also goes for software bugs, which are errors in computer programs that cause them to behave differently than intended. Unexpected erroneous behavior of software can lead to serious problems, of which there are many examples:

- In 1985-1987, the Therac-25 radiation therapy device malfunctioned, on occasion delivering lethal radiation doses instead of safe ones. At least five patients died, and several others were injured. The cause: programming errors in the underlying operating system.

- In 1996, the European Space Agency launched a prototype Ariane 5 rocket. Forty seconds after launch, the $1 billion costing rocket was destroyed, due to a bug in the on-board guidance program.

- In 1999, the Mars Climate Orbiter was officially assumed lost by NASA. The $125 million orbiter failed due to a communication bug in the system software: one module passed measurements from the metric system to another module that expected measurements in the imperial system.

- In 2006, the Dutch Tax Authorities (‘de Nederlandse Belastingdienst’) mistakenly sent tax assessment forms to 230,000 people who didn’t have to pay taxes at all. This was due to a bug in the software used by the Tax Authorities. All 230,000 were sent a letter of excuse afterwards.
Many, many more examples of malfunctioning software exist. In fact, software bugs are so common that in 2002 a study commissioned by the Department of Commerce’s National Institute of Standards and Technology (NIST) estimated that they cost the US economy $59.5 billion annually, which is about 0.6% of the gross domestic product. From minor annoyances to life-threatening problems, most people are likely to have been affected by software bugs at some point.

1.2 Prevent vs Cure

It is a well-known saying that ‘prevention is better than cure’. This saying can be applied to software bugs as well. In this context, ‘prevention’ translates to quality measures that are applied during the process of software development, and ‘cure’ translates to the correction of errors in the finished product, after the process of development. Because it is easier to correct errors in the earlier stages of development, prevention is indeed more effective than cure, also in the case of software bugs.

Many measures for preventing programming errors have been developed, and in practice they are very effective in reducing the number of bugs in the software that is produced. Unfortunately, prevention cannot be expected to eliminate all software bugs completely, because:

- software is complex: even a medium-sized program may involve millions of dependent decisions to be made, and a mistake in any one of them may lead to unexpected behavior of the program as a whole;

- writing computer programs is a task that is still carried out by humans, and making (small) mistakes is an inherent component of human nature.

We rely on software for carrying out critical tasks in our society. In order to reach the required level of reliability, we therefore need both prevention and cure. This makes it important to perform research both into better software development methods and into the effective detection and correction of software bugs.

1.3 Detecting software bugs

Detecting errors in developed software can be performed in different ways. The following three major methods can currently be distinguished:

- Testing[Bei95, Het88, Mye04].

  In this method, the program itself is executed repeatedly on representative input data. For each input, the actual execution behavior of the program is compared to the expected behavior. If there is a difference, then the program is considered erroneous for the respective input.
Section 1.3: Detecting software bugs

- **Model Checking** [BBF+01, CGP99, Vaa06].
  
  In this method, a simplified model of the program is built. The model focuses on the selected main functionality of the program. It must be small enough, such that the input that it allows can be enumerated completely. Also, it must be expressed in a framework that allows both automatic execution and automatic verification. The correctness of the model is then verified in a brute-force way, by executing it automatically on all of its inputs.

- **Formal Reasoning** [KF80, Kle02].
  
  In this method, the source code of the program is considered to be a formal mathematical object. The expected behavior of the program must be expressed by means of desired logical properties of this object. Then, formal proofs of these properties must be constructed. If this succeeds, it is shown that the program behaves as expected.

These methods are very different, and they all have their respective advantages and disadvantages:

- Testing is the only method that does not require extensive mathematical expertise, and it can be applied realistically to software of any size. It is therefore used predominantly in industrial practice. Note, on the other hand, that testing is mainly aimed at detecting the presence of bugs, but not their absence [Dij79] (EWD249).
  
  An abundance of test tools is available in practice, ranging from commercial tools to open source unit test tools such as JUNIT [BG98] and academic model-based test tools such as TORX [TB03] and GAST [KATP03].

- Model checking is more difficult to apply, but also finds somewhat more errors due to the completeness of the tested input. It is used quite a bit for verifying safety-critical software, see for instance [CSV07, vSU08].
  
  Many model checkers are available in practice, such as for instance UPPAAAL [BDL04], SPIN [Hol97] and MCRL2 [GMR+07]. Also, in 2007, the Turing award [McC71] was granted to Edmund Clarke, Allen Emerson and Joseph Sifakis [CES86, QS82] for “their roles in developing model checking into a highly effective verification technology, widely adopted in the hardware and software industries”.

- Formal reasoning is most difficult to apply, but is also capable of showing the complete absence of errors. It scales really badly [JP04], however, which makes it too expensive to be used for realistic pieces of software. It is only used on occasion for verifying small pieces of safety-critical software.
  
  Although less than for testing and model checking, many proof assistants (tools for formal reasoning) are available in practice, such as for instance LCF [GMW79], PVS [OSRS01], COQ [The06] and ISABELLE [Pau07].
Which method is most suitable depends highly on the situation at hand, and is really an assessment of cost and gain. What all methods have in common, however, is that their application in practice will come at a substantial cost, often of over 50% of the total cost of development.

The research topic of this thesis is formal reasoning. In the following sections, we will explore the process of formal reasoning in more detail.

1.4 Formal reasoning

Formal reasoning is the mathematical process of building proofs. It takes an object to reason about and a desired property of the object as input. The aim is then to produce a proof of the property as output. The process is carried out on the formal level entirely, and formal languages have to be available for all the kinds of data that are used:

- An object language must be available to formalize the objects that will be reasoned about. In principle, any kind of object language is allowed.

- A property language must be available to formalize the properties that can be expressed about the objects. A property is a statement about an object that can either be true or false. The property language must contain those properties that are desired from, or interesting for, elements of the object language. In practice, properties are usually constructed analogously to the theory of logic.

- A proof language must be available to formalize the steps that may be used to build proofs. A proof step is an action that takes a single property to prove as input, and produces a (possibly empty) list of new properties to prove as output. Proof steps must be sound, which means that the validity of the produced properties must imply the validity of the input property. Proof steps are also usually inspired by the theory of logic.

The combination of these three interrelated languages will be called the formal framework. The availability of a formal framework is a prerequisite for formal reasoning. For any object language, either a new specialized formal framework has to built explicitly, or a transformation to an existing framework has to be realized. Note, however, that each framework is usually built upon elements that are well-known from the theory of logic. The concept of formal frameworks is also treated in Section 4.2.2. Examples of formal frameworks can for instance be found in [The06] (for the proof assistant Coq) and in [dvP07a] (for the proof assistant SPARKLE).

The main activity of formal reasoning is the construction of the proof. This process can be considered as the repeated transformation of a list of properties that need to be proved. Initially, the list contains the target property only. At each point, a proof step must be applied to one of the properties in the list. This transforms the list as a whole to a new list, which depending on the output of
the proof step can either be shorter, or at least as long. The proof is complete when the list of properties has become empty.

The proof object that is constructed by means of reasoning is represented by means of a tree, in which the nodes are decorated with properties and the edges correspond to applications of proof steps. The aim of formal reasoning is to build a proof with the target property as its root, and in which all leaves have been ‘closed’ by proof steps that produce the empty list as output.

1.5 Soundness of formal reasoning

Formal reasoning is a process that strives to establish meaning by means of representation. On the one hand, the purpose of reasoning is to show on the semantic level that an object behaves as specified. The proof object that is built for this purpose, however, lives on the representation level entirely, because it consists of the syntactical transformation of formalized properties only.

The connection between meaning and representation can be realized on the level of the underlying formal framework completely, by means of adding the following two components to it:

- A semantics for the property language (and the object language). This semantics describes which properties are true, and which properties are false. Because properties relate to objects, a semantics for the object language is implicitly assumed to be available as well.

  We assume that the semantics can only be described on the mathematical level, and cannot be transformed into an executable decision algorithm. Otherwise, formal reasoning would not be necessary at all!

- Soundness proofs for the proof steps. A proof step is sound if the meaning of the (conjunction of the) produced properties always implies the meaning of the original property. Using the semantics of the property language, this can now be proved formally.

A formal framework that also contains these two components will be called sound itself. If a sound framework is used for formal reasoning, then all proofs that are constructed with it are automatically guaranteed to show the validity of the proved properties on the semantic level. This allows proofs to remain purely syntactical, yet hold a practical meaning as well.

Finally, note that even though proofs themselves are syntactical objects, building them involves the continuous selection of the right reasoning steps, which requires expertise and intuition.

1.6 Characteristics of formal reasoning

Formal reasoning is very different from the other methods of detecting software bugs. Below, we treat several characteristics of formal reasoning that we
find particularly interesting. Note that this list is not intended as a detailed comparison between formal reasoning, model checking and testing.

- Formal reasoning is a positive approach that attempts to verify correctness for all possible circumstances at once. This affects the results obtained by formal reasoning as follows:

  - It strengthens the result of success. If a proof can be built successfully, then it is shown in one go that the program always behaves as specified, regardless of its input.

  - It weakens the result of failure. If a proof cannot be built successfully, then it is still unsure whether a software bug has been detected. It can namely also be the case that the property was specified too generally (i.e. it holds for most cases, but not for certain exceptions), or that the available proof steps are not sufficiently powerful. Additionally, it may also simply be the case that the proof builder lacks the necessary expertise to build the proof in the correct way.

In other words: a high level of reliability can be obtained with formal reasoning, but it is somewhat more difficult to actually find errors with it.

- Formal reasoning does not treat the program as a black box, but operates on the level of its source code. It therefore concerns the actual algorithms of the program, instead of its observed input/output behavior.

  Consequently, formal reasoning requires more expertise: one does not only need to understand the result that is produced, but also the way in which it is computed. The advantage, on the other hand, is that it is implicitly the actual algorithm that is being verified, instead of the correctness of the end result only.

- Formal reasoning is carried out on the formal level completely. In order to apply its results to practice, one has to assume that the real-world behavior of the program corresponds to its formal behavior. Unfortunately, this correspondence is affected by many external factors, such as hardware, operating system, compiler, linker, etc.

  On the other hand, formal reasoning is ideal for the application of theoretic research.

### 1.7 Research setting

We think that formal reasoning is a worthwhile and interesting activity that may contribute to increasing the quality of software. Unfortunately, it is in general also very difficult to carry out. The precise degree of difficulty, however, depends on the circumstances in which formal reasoning is applied. In certain settings, formal reasoning will be a lot easier than in other ones.
In this thesis, we have applied formal reasoning to a setting which we perceive as particularly favourable: namely in the context of the functional programming language Clean \[Pv98\]. From a research point of view, this serves two purposes:

1. We want to make formal reasoning available as an easily accessible tool to Clean programmers, allowing them to make use of it for the purpose of increasing the quality of their programs; and

2. We want to investigate how useful formal reasoning can become if it is applied in a particularly favourable setting.

Note that the functional programming language Haskell \[Pey03\], which is very similar to Clean, would also have been a good choice for our research context. I have chosen Clean as research vehicle because of the expertise in Clean that is available in Nijmegen, and because of my own familiarity with it.

Functional programming languages are mathematical in nature, which not only offers benefits to programming itself\[Hug89\], but also provides particular advantages to formal reasoning:

- The nice mathematical properties of functional programs allow powerful and user-friendly reasoning techniques to be used. In particular, *equational reasoning* is available, because of the referential transparency of functional languages. Equational reasoning allows equal expressions to be replaced with each other at any point in the proof, which is very useful for reasoning.

- Functional programs are formal objects with well-defined rigid semantics. Therefore, reasoning can take place on the level of the program itself, and no additional formalization is necessary.

- Writing functional programs requires a basic understanding of mathematics. Therefore, experienced programmers already have some of the expertise that is required for formal reasoning.

The suitability of functional languages for formal reasoning is also illustrated by \[Bir98\], in which many example properties of Haskell-programs are manually proved with ease.

Note that a Haskell-frontend for Clean is currently being developed at the Radboud University Nijmegen. With this frontend, the results of this thesis will become applicable to Haskell as well.

### 1.8 Contents of this thesis: SPARKLE

In order to apply formal reasoning to the real-world functional programming language Clean, we have developed the specialized proof assistant SPARKLE for it. A proof assistant is a computer tool which aids users in performing formal reasoning. Because of the complexity of formal reasoning, a proof assistant is a
prerequisite for performing it effectively. By means of the addition of Sparkle, we have made it possible to reason about Clean programs in practice.

The main research topic of this thesis is the proof assistant Sparkle. It will be described in detail, its application in (scientific) practice will be examined, and its contribution to theory will be investigated. By means of developing Sparkle and introducing it both to theory and practice, we have researched formal reasoning in the setting of functional programming languages.

The remainder of this thesis is structured as an enumeration of nine separate articles. Seven of these articles have been published in scientific journals and proceedings, one is to be submitted in 2008, and one is a (modified) chapter from an internally published technical report. I have been the main author of five of these articles, and a contributing author to the other four. The (to be) published articles in this thesis have not been modified and they are, with the exception of slightly unified formatting, identical to their published versions.

The contents of this thesis can roughly be divided into four categories:

- Chapters 2, 3 and 4 and Appendix A provide a description of Sparkle;
- Chapters 5 and 6 describe reasoning with strictness;
- Chapters 7 and 8 describe the theoretical background of Sparkle; and
- Chapters 9 and 10 describe applications of Sparkle.

1.8.1 Description of Sparkle (Chapters 2, 3, 4; App. A)

The first proof assistant for Clean was developed in 1999. It was intended as a preliminary prototype to investigate the prospects of formal reasoning, and was named ‘CleanProverSystem’. It was restricted to a simplified functional language, and its simple user interface emphasized automatic reasoning. Within this limited setting, however, it turned out that it was already possible to prove many interesting properties of Clean programs with CleanProverSystem.

Therefore, our research on formal reasoning for Clean was continued, and in 2001 the first version of Sparkle in its current form was produced. The following specific features were realized in Sparkle:

- full support for all Clean programs, both syntactically and semantically, with the exception of I/O operations and machine code;
- seamless integration into the development environment of Clean (by means of the standard editor, which allows Sparkle to be started directly on the current project; but unlike for instance Coq (λ-calculus) and Agda (dependent types) proofs and programs are still stored separately);
- user-friendly reasoning, by means of reasoning steps that are dedicated to Clean, a hint mechanism that allows semi-automatic reasoning, and an extensive graphical user-interface.
I have designed and implemented both Sparkle and CleanProverSystem myself, entirely in the programming language Clean. Sparkle is one of the bigger Clean programs. The current 2008 version consists of approximately 55,000 lines of source code, counting comments as well, and makes use of the Clean 2.0 compiler source code (approximately 40,000 lines of code), and the Object I/O library (approximately 40,000 lines of code).

The following four chapters of this thesis provide a general introduction to both CleanProverSystem and Sparkle:

- **Chapter 2: A Proof Tool Dedicated to Clean: the first prototype.**
  (written by Maarten de Mol and Marko van Eekelen; presented at the 1st International Workshop on Applications of Graph Transformations with Industrial Relevance (AGTIVE 1999), Kerkrade, the Netherlands; published in LNCS proceedings volume 1779, [dv99a])

  This chapter describes the preliminary research on which Sparkle was based, by means of introducing CleanProverSystem. It investigates both the usefulness and the restrictions of the prototype.

- **Chapter 3: Theorem Proving for Functional Programmers.**
  (written by Maarten de Mol, Marko van Eekelen and Rinus Plasmeijer; presented at the 13th International Workshop on the Implementation of Functional Languages (IFL 2001), Stockholm, Sweden; published in LNCS proceedings volume 2312, [dvP02])

  This chapter contains the first scientific publication about Sparkle. It introduces Sparkle by means of its primary purpose, which is to allow functional programmers to make use of formal reasoning in practice. It focuses on the integration of Sparkle into the development environment of Clean, the support for reasoning on the level of the programming language, and the available tools for (semi-)automatic reasoning.

- **Chapter 4: Proving Properties of Lazy Functional Programs with Sparkle.**
  (written by Maarten de Mol, Marko van Eekelen and Rinus Plasmeijer; presented at the 2nd Central-European Functional Programming School (CEFP 2007), Cluj-Napoca, Romania; to be published in LNCS tutorial proceedings, [dvP08a])

  This chapter presents a comprehensive stand-alone description of the use of Sparkle in practice. It introduces the basic features of Sparkle by means of a step-by-step tutorial with exercises, and examines the advanced functionality that is available for sharing, definedness and reduction. It also summarizes all reasoning steps that are made available by Sparkle.

  Note that both this and the previous chapter give a general introduction to Sparkle, but from different perspectives. Chapter 3 focuses on scientific value, and goes further into the purpose and the dedicated nature of Sparkle. Chapter 4 focuses on practical value, and goes further into the reasoning steps and proving capabilities of Sparkle.
Chapter 1: Introduction

- Appendix A: Tactic Library of Sparkle.

This chapter provides a short description of all tactics that are made available by Sparkle. It serves as a preliminary reference guide for the reasoning possibilities of Sparkle.

This chapter has been published as the local appendix of [dvP08a]. In this thesis, it has been moved from Chapter 4 to a global appendix. It also replaces the local appendix of [dvP02], which has been removed from Chapter 3 in this thesis. Furthermore, the tactics that were added to Sparkle after the publication of [dvP08a] have been added to it.

1.8.2 Reasoning with strictness (Chapters 5,6)

An important feature of Clean is the combination of lazy evaluation (‘only compute when necessary’) with optional explicit strictness (‘must compute this now’). Although Haskell [HPW+92] also supports lazy evaluation and explicit strictness, more kinds of strictness annotations are available in Clean, and the feature is used much more often in Clean.

The combination of lazy evaluation and explicit strictness not only influences the underlying semantics, but has a profound effect on reasoning as well. The effect on semantics is well-known, but the effect on reasoning is often underestimated, and little research has been devoted to it. In this thesis, the effect on reasoning will therefore be treated explicitly in the following two chapters:

- Chapter 5: Proof Tool Support for Explicit Strictness.

  (written by Marko van Eekelen and Maarten de Mol; presented at the 17th International Workshop on the Implementation of Functional Languages (IFL 2005), Dublin, Ireland; published in LNCS proceedings volume 4015, [vd06])

  This chapter examines the effect of explicit strictness on the validity of properties and proofs. It shows that the addition (or removal) of explicit strictness may cause properties to become invalid, and that dealing with explicit strictness requires specialized reasoning steps. Furthermore, it introduces the reasoning support that Sparkle offers for dealing with explicit strictness in practice.

  Marko van Eekelen and myself have contributed equally to this chapter.

- Chapter 6: Proving Lazy Folklore with Mixed Lazy/Strict Semantics.

  (written by Marko van Eekelen and Maarten de Mol; published in Reflections on Type Theory, λ-calculus, and the Mind: Essays dedicated to Henk Barendregt on the Occasion of his 60th Birthday, Radboud University Nijmegen, 2007, [vd07])

  This chapter extends the natural semantics of Launchbury[Lau93], which is the standard theoretical description of lazy evaluation, with explicit strictness. The extension is shown to be both correct and computationally adequate.
In this chapter, I have contributed to the motivational introduction, and to the definitions of the extended expression language and the operational semantics.

Note that there is an overlap between the motivational introductions of Chapters 5 and 6. Chapter 5, however, continues on the practical level with the reasoning support that is needed for dealing with explicit strictness, whereas Chapter 6 continues on the theoretical level with a foundation for explicit strictness.

### 1.8.3 Theoretical background of Sparkle (Chapters 7,8)

In order to ensure the correctness of proofs that are constructed with Sparkle, a formal framework [dvP07a] has been constructed for it. The framework is based on the graph-based semantics of Clean. It defines the following:

- formal representations for expressions, programs and properties;
- a reduction mechanism for expressions;
- a proof of confluence of reduction;
- an observational semantics for properties;
- a proof of referential transparency of the semantics;
- formal representations for reasoning steps and proofs;
- correctness proofs for (most of) the reasoning steps.

The formal framework is a (209 page) enumeration of definitions and proofs, all on the formal level. As a whole, it mainly serves as a reference work, and for this reason it has been published as an internal report. From a scientific point of view, the custom reduction mechanism and its properties are of particular interest, and they have been condensed into a publication [dvP08b].

The reduction mechanism publication is included as a chapter in this thesis. Additionally, the semantics of expression equality will be treated in a separate chapter as well, because expression equality plays a central role in the foundation of Sparkle.

- **Chapter 7: A Single-Step Term-Graph Reduction System for Proof Assistants.**
  
  (written by Maarten de Mol, Marko van Eckelen and Rinus Plasmeijer; presented at the 3rd International Workshop on Applications of Graph Transformations with Industrial Relevance (AGTIVE 2007), Kassel, Germany; published in LNCS proceedings volume 5088, [dvP08b])

This chapter presents a reduction mechanism that is specifically suited for formal reasoning. The reduction mechanism is derived from the standard system of Launchbury [Lau93], but makes use of single-step reduction and leaves the choice of redex free. The derived reduction mechanism is proved to be confluent, and is related to the original mechanism of Launchbury.

The reduction mechanism in this chapter is a simplification of the version used in the formal framework. A smaller functional language is used, less reduction rules are required, and the confluence proof is less complicated.
Chapter 8: Semantics of Expression Equality.
(written by Maarten de Mol; modified chapter of the formal framework of Sparkle [dvP07a])

This chapter presents a formal semantics of expression equality in the context of a given program. The equality is defined operationally in terms of observational program behavior. It is sufficiently powerful to deal with both terminating and non-terminating computations.

This chapter also appears in the formal framework, but has been modified to make it self-contained.

1.8.4 Applications of Sparkle (Chapters 9,10)

With my assistance, Sparkle has also been applied in other research projects in Nijmegen. It has either played the role of research vehicle, which can be used to realize new research ideas for reasoning about functional programs, or the role of a finished proof assistant, which can be used to verify important properties of programs (or models) written in Clean.

The following two chapters describe interesting uses of Sparkle in such other research projects I was involved in:

• Chapter 9: Proof Support for General Type Classes.
  (written by Ron van Kesteren, Marko van Eekelen and Maarten de Mol; presented at the 5th International Symposium on Trends in Functional Programming (TFP 2004), München, Germany; published in Intellect proceedings Trends in Functional Programming, volume 5; best student paper award, [vvd04])

This chapter describes a method for reasoning about properties that apply to all available instances of a type class. The property language is extended with class constrained properties, and a reasoning step is defined that performs induction according to the structure of the instance tree. The extension has been implemented in an independent version of Sparkle.

In this chapter, I have contributed to the definition of the induction schemes and the realization in Sparkle.

• Chapter 10: A Common Arrow Based Semantics for GEC and iData Applications.
  (written by Peter Achten, Marko van Eekelen, Maarten de Mol and Rinus Plasmeijer; under submission to JFP, [AvdP08])

This chapter introduces a unified model for the GEC and iData toolkits. The model is based on the arrow framework and defines the standard arrow operators, as well as customized editread and editset operators. Various interesting properties of the model are formulated, including the standard arrow laws, custom laws concerning editors, and definedness properties. All properties have been proved with Sparkle.
In this chapter, I have mainly contributed to the formulation of the definedness laws, the translation of the model to CLEAN and the proofs in SPARKLE.

1.9 Impact of SPARKLE

SPARKLE has also been used by others, both in Nijmegen and in several other places, for various research projects:

- At the Radboud University Nijmegen, Leonard Lensink has extended SPARKLE with general induction schemes in [Lv04].

- In Ireland at Trinity College of University of Dublin, SPARKLE has been used for reasoning about models of I/O-programs by Butterfield and Dowse. In [DBv05], they proved properties of a state-transforming functional I/O-model with SPARKLE. In [DB06], they extended this model to CURIO, and again used SPARKLE to formally verify its properties.

- In Hungary at the Eötvös Loránd University of Budapest, SPARKLE has been used for reasoning with temporal properties by Horváth, Koszik and Tejfel. In [HKT04], they proved temporal properties of an interactive database with SPARKLE, performing specialized temporal proof rules by hand. They incorporated these proof rules in a local independent extension of SPARKLE, called SPARKLE-T [THK06]. Using SPARKLE-T, they have proved temporal properties of CLEAN programs in [THK05].

  Furthermore, they have also used SPARKLE for reasoning about I/O-programs. In [TKH08], they introduced Sio, a model in which simplified CLEAN Object I/O programs can be expressed, and used SPARKLE to prove properties of this model.

- For educational purposes, users have also created their own documentation of SPARKLE, which is available online at:
  - http://www.possibly.me.uk/notes/sparkle.shtml (created by Andy Fugard at University of Edinburgh)

  Note that the SPARKLE-proofs that result from these projects are already of respectable size. The larger proofs are split across several sections, consist of over a hundred subtheorems and thousands of tactic applications, and require considerable time and memory to be loaded into SPARKLE.

1.10 Applicability of SPARKLE

SPARKLE is a fully functional proof assistant that can be used in practice by anyone. Its friendly user interface and integration into the CLEAN programming
environment make the threshold for starting with reasoning very low. Because it is also part of the standard distribution of Clean, Sparkle has made formal reasoning truly available to all who have downloaded the language.

The dedicated features of Sparkle ensure that it can be used easily by Clean-programmers. Firstly, its reasoning steps are tailored towards Clean; in particular, Reduce and Induction are adapted to Clean’s semantics. Secondly, it supports special reasoning steps for dealing with situations that are common in Clean, such as Definedness which directly proves definedness conditions. Thirdly, it makes a hint mechanism available with which trivial proofs can be discarded automatically. These features make building small proofs in Sparkle very easy; scaling issues still exist for the bigger proofs, however.

On the other hand, Sparkle is also lacking in some departments and has its own share of annoyances. Firstly, it does not support all of Clean, and does not allow reasoning on generics, I/O or unique structures. Also, it lacks semantic models for natural numbers (which are currently described by means of axioms) and comprehensions (which are currently translated to basic functions). Secondly, the documentation of Sparkle is lacking. Basically, all available documentation is contained in this thesis. Thirdly, Sparkle lacks a way of replaying proofs that have become invalid in an intermediate stage, but which should still be valid for the most part.

All in all, however, Sparkle is a very useful tool that has contributed both to the applicability of formal reasoning in general and to the programming language Clean in specific.
Abstract. Theorem proving for functional programming languages can be made much easier by the availability of a dedicated theorem prover. A theorem prover is dedicated to a specific programming language when it fully supports the syntax and semantics of the language and offers specialized proving support for it. Using a dedicated theorem prover is easy, because one can reason about a developed program without having to translate it. However, no suited dedicated theorem prover for a functional language exists yet. This paper describes a simple prototype of a dedicated theorem prover for the functional language Clean. A description of the possibilities of the prototype is given and an examination is made of the work that needs to be done to extend the prototype to a fully operational and truly useful programming tool. Also example proofs of some basic properties and of a graph transformation are given.

2.1 Introduction

Functional programming languages like Clean [Pv98] and Haskell [Tho99] are well suited for theorem proving. They are based on the well defined notion of term graph rewriting [SPM93] and are free of side-effects. As can be seen in [Bir98], it is very easy to prove simple properties of functional programs.

Unfortunately, when programs get larger, theorem proving gets increasingly more difficult. Proving properties of real-life applications can take several months and is still only performed by teams of experts. But proving properties of small essential pieces of the program can be very useful as well, especially when it is done in an early phase. Errors in functions can be corrected before they have effect on other parts of the program. Once the correctness of a function has been established, it can be used (and re-used) safely in other parts of the application.
Good support for theorem proving could benefit programmers. Many powerful tools for theorem proving are available, like for instance COQ [The98] and ISABELLE [Pau98], which are claimed to be well suited for functional programming languages. However, proving properties of programs written in CLEAN using COQ or ISABELLE turned out to be very difficult. They do not support the syntax or semantics of CLEAN, making it necessary to model the semantics of CLEAN and to translate the program to this model. The reasoning then takes place on the model of the program, instead of on the program itself. Also, the user interfaces of COQ and ISABELLE are primitive and do not offer much support for the interactive reasoning process. Commands have to be typed explicitly in some kind of syntax and it is difficult to find out what commands are needed to finish a proof.

These problems can be partially overcome by building an interface on top of COQ or ISABELLE. This interface can automatically translate programs to and from the theorem prover and provide a sophisticated user interface. Still, the semantics of CLEAN has to be modeled in COQ or ISABELLE and this is far from trivial.

But for proving simple properties only a small part of COQ or ISABELLE is needed. This makes it feasible to implement a small theorem prover for CLEAN ourselves. This will eliminate the need for translations, since the reasoning will take place on the program itself. Also a dedicated theorem prover can be developed to meet our specific goals, i.e. usable by programmers during the development of programs to prove simple properties fast.

To test the effort needed to implement a small dedicated theorem prover for CLEAN a prototype has been developed. It turned out to be fairly easy to implement a reasonably powerful theorem prover. In this paper a short description of the prototype will be given. First the restrictions of the prototype will be given. It is described how proofs can be built using the prototype. Some examples of proven theorems and proofs are given next. Finally the extension of the prototype to a complete theorem prover is discussed.

### 2.2 Restrictions of the first prototype

Developing a theorem prover which fully supports CLEAN is a lot of work. To allow for the rapid development of a prototype, the input language has been simplified a lot. First of all the graph rewriting which underlies CLEAN is reduced to term rewriting; no cycles are allowed. Secondly the lazy reduction mechanism is reduced to an eager one; no infinite intermediate results are allowed and functions must always terminate. Thirdly partial functions are not allowed; all expressions must have a well defined value. Finally syntactic sugar like comprehensions, dot-dot expressions and local definitions are not supported.

What is left is a very small subset of a functional language. This is not only a subset of CLEAN, but for instance of HASKELL and ML as well. The results of the prototype can therefore be applied to other functional languages than CLEAN as well. Although the subset is very small, many interesting functions
can be expressed in it. In the future work some issues on how the restrictions will be lifted are discussed.

### 2.3 Using the prototype to construct proofs

In order to prove a theorem about a program in the prototype three things have to be done: (1) the program has to be expressed in the prototype (this boils down to removing the sugar from a simple CLEAN program), (2) the theorem has to be expressed in the prototype and (3) the proof for the theorem has to be constructed by supplying proving commands. In the next subsections these phases are described separately.

#### 2.3.1 Specification of the program

Algebraic types are the only definable types in the prototype. An algebraic type is defined by a number of data-constructors which are able to construct elements of the type. The type of the booleans can for instance be defined as:

:: Bool = True | False

It is also possible to define higher-order types and to define types using recursion. In this way the type of the lists can be defined as:

:: List a = Nil | Cons a (List a)

The empty list Nil is usually denoted by [] and the construction of lists by Cons x xs is usually denoted by [x:xs].

Functions are defined using pattern-matching. In the left-hand-side of a pattern only applications of data-constructors on variables are allowed. The right-hand-side of a pattern can be any expression. An expression can either be a variable or an application of a function or data-constructor. Higher-order, partial and recursive applications are allowed. An example of a valid definition is:

Map :: (a -> b) (List a) -> (List b)  
Map f [] = []  
Map f [x:xs] = [f x: Map f xs]

#### 2.3.2 Specification of the theorem

Theorems about programs are basically equalities between expressions stated in a first-order predicate logic. The logical operators that are allowed are ∨, ∧, ¬ and →. Quantifications over types and over expressions of any type are allowed. Examples of stated properties are:

∀a,b.∀f:a→b.Map f [] = []  
∀a.∀x:a.∀xs:List a.Length [x:xs] = Length xs + 1
2.3.3 Building a proof

Proofs are constructed much the same way as in most traditional theorem provers. First the statement to prove is specified as the current goal. This goal is then gradually transformed to simpler goals by the application of reasoning steps. This kind of reasoning is called backwards reasoning.

The reasoning steps are called tactics. Each tactic must be sound with respect to the semantics of the program. A tactic may transform a goal to a logically equivalent or stronger one. The former tactics will be called ‘safe tactics’, the latter ‘risky tactics’. A ‘risky tactic’ can easily lead to a proof state which can’t be extended to a complete proof and must therefore be handled with care. For this purpose the risky tactics return a list of possible outcomes, while the safe tactics produce only one outcome. The following safe tactics are available in the prototype:

1. **Uncurry.** Collects arguments of applications in sequel, for example:
   
   \((\text{Map} +) \, [] \Rightarrow \text{Map} + \, []\). All occurrences are rewritten at once.

2. **SimplifyStep.** Applies a rewrite-rule. Function patterns, lemmas, proven goals and the semantics of the logical operators are all represented by rewrite-rules. At most one rewrite is executed.

3. **UnequalConstructors.** Replaces at most one equality between two different data-constructors by \(\text{False}\).

4. **Split.** Splits a goal \(P \land Q\) in two goals \(P\) and \(Q\).

5. **HypoStep.** Creates rewrite-rules \(P \rightarrow Q \Rightarrow True\) for each suitable hypothesis \(P \rightarrow Q\) in the context and calls **SimplifyStep** with this set of rewrite-rules.

6. **Induction.** Applies standard induction on the outermost quantification. The appropriate induction scheme is dynamically constructed. Induction on all algebraic types is allowed.

7. **Introduction.** Either removes the outermost quantification by adding the typing information to the context, or transforms a goal \(P \rightarrow Q\) to \(Q\) by adding \(P\) as a hypothesis to the context.

The following risky tactics are available:

8. **Generalize.** Substitutes a suitable subexpression by a free variable and then adds a quantification over it. This tactic can not be used on variables. For each suitable subexpression an outcome is generated.

9. **SimplifyEquality.** Rewrites at most one equality between applications of the same function by assuming that the function is injective.

10. **GeneralizeVariable.** Adds a quantification over a free variable in the goal. For each free variable an outcome is generated.
Section 2.3.4: Automatic proof construction

11. **Unintroduce.** Unintroduces a hypothesis in the context by creating an implication in the goal. The hypothesis is not removed from the context. For each hypothesis an outcome is created.

12. **UseEquality.** Creates rewrite-rules $P \Rightarrow Q$ and $Q \Rightarrow P$ for each hypothesis $P = Q$ in the context and calls `SimplifyStep` with this set of rewrite-rules.

All of the tactics are also present in one way or the other in traditional theorem provers. This small set of tactics proved to be powerful enough for the prototype. A more detailed description of the tactics can be found in [dv99b].

### 2.3.4 Automatic proof construction

The tactics in the previous subsection are the basic tactics of the prototype. An advantage of a dedicated theorem prover is that tactics can be composed in the way that is most convenient for the application domain. The prototype provides a composed tactic `Auto` for this purpose, with which automatic proof search specifically for simple theorems about functional programs can be modeled.

Ideally the `Auto` tactic should try all possible combinations of basic tactics. In this way it can be ensured that as many proofs can be found automatically as interactively. Unfortunately this is not possible, since trying all possible combinations of basic tactics simply takes too much time. Therefore the number of tried combinations is reduced using a search heuristic. For this heuristic first a stripped version, called `SafeAuto`, which is used for recursive calls, is defined. This tactic uses the following strategy:

1. Apply the first safe tactic that can be applied on the current goal. Make the outcome the new goal and recursively call `SafeAuto`. Proceed to step 2 when no safe tactic can be applied.
   Note that induction is always tried before introduction.

2. Apply `Generalize` to obtain a list of outcomes. Recursively call `SafeAuto` on each outcome. If calling `SafeAuto` completely solves an outcome, use this result and exit. Otherwise undo the application of `Generalize`.
   This procedure will be abbreviated as ‘multitry `Generalize` with `SafeAuto`’.


A distinction is thus made between the safe tactics and the risky tactics. Applications of safe tactics can never be undone, while a form of backtracking is performed for risky tactics. The `Auto` tactic can now be described as follows:

1. Apply `SafeAuto`.

   *(`GeneralizeVariable2` is applying `GeneralizeVariable` twice, storing all possible outcomes in a single list)*

3. Multitry `Unintroduce2` with `SafeAuto`.

Prohibiting recursive calls of GeneralizeVariable, Unintroduce and UseEquality and eliminating backtracking after safe tactics makes the Auto tactic fast enough. Fortunately, our experiences show that little proving power is lost.

2.3.5 Examples of proven theorems and proofs

The prototype has been tested using examples from the book ‘Introduction to Functional Programming using Haskell’[Bir98]. All 72 tried theorems could be proven. A total of 27 lemmas were introduced to facilitate the proving process. These lemmas were inspired by stuck proving sessions and were easily found. All lemmas could be proven automatically and, using the lemmas, 70 of the 72 theorems could be proven automatically as well. A full list of proven theorems can be found in [de 98], below some examples:

1. $\forall a \forall xs :: \text{List } a. \text{Reverse (Reverse } xs) = xs$
2. $\forall a \forall xs :: \text{List } a. \forall n :: \text{Nat}. (\text{Take } n \ xs) ++ (\text{Drop } n \ xs) = xs$
3. $\forall x :: \text{Nat} \forall y :: \text{Nat} \forall z :: \text{Nat}. x ^ (y + z) = (x ^ y) * (x ^ z)$
4. $\forall x :: \text{Nat}. \text{Log} (2 ^ x) = x$

An example proof of the first theorem is shown in Table 1. An automatic proof attempt fails on proof state 6. Examining this state a lemma was introduced: $\forall a \forall x :: \text{Nat} \forall y :: \text{List } a. \text{Reverse (xs ++ [x])} = [x : \text{Reverse } xs]$. This lemma was proven automatically and using the lemma the proof can be completed automatically.

2.3.6 Upgrading the prototype: further work

The prototype is a very small theorem prover. A lot of work needs to be done to obtain a fully operational dedicated theorem prover for CLEAN:

1. **Support for full syntax.** This can easily be accomplished by re-using the existing parser for CLEAN. By invoking the compiler one can even get a simpler (internal) representation of a program written in CLEAN as well.

2. **Support for full semantics.** The semantics has to be extended with laziness, partial functions and graphs. A large part can be accomplished by implementing lazy graph-rewriting in the theorem prover.

3. **Tactics for infinite structures.** Infinite structures require different proving techniques, like for instance co-induction (see example in next subsection). These techniques are however much more difficult to use than ordinary techniques like structural induction. Therefore it may be necessary in some cases to prove correctness provided no infinite structures occur.
Section 2.3.7: Example proof with graphs and co-induction

1. Introduce a
\[ \forall a. \forall x :: List a. \text{Reverse} (\text{Reverse} \; xs) = xs \]
2. Induction \( xs \)
\[ \text{Reverse} (\text{Reverse} \; []) = [] \quad \text{IB} \]
3. Simplify With "Pattern Match Rule [Reverse []]"
\[ \text{Reverse} \; [] = [] \]
4. Simplify With "Pattern Match Rule [Reverse []]"
\[ [] = [] \]
5. Simplify With "Rule [x = x]"
\[ \text{Reverse} (\text{Reverse} \; xs) = xs \rightarrow \text{Reverse} (\text{Reverse} \; [x:xs]) = [x:xs] \quad \text{IH} \]
6. Simplify With "Pattern Match Rule [Reverse [x:y]]"
\[ \text{Reverse} (\text{Reverse} \; xs) = xs \rightarrow \text{Reverse} ((\text{Reverse} \; xs) ++ [x]) = [x:xs] \]
7. Simplify With "Lemma [Reverse (xs ++ [x]) = [x:Reverse xs]]"
\[ \text{Reverse} (\text{Reverse} \; xs) = xs \rightarrow [x:\text{Reverse} \; (\text{Reverse} \; xs)] = [x:xs] \]
8. Simplify With "Rule [[x:xs] = [y:ys]]"
\[ \text{Reverse} (\text{Reverse} \; xs) = xs \rightarrow x = x \land \text{Reverse} \; (\text{Reverse} \; xs) = xs \]
9. Simplify With "Rule [x = x]"
\[ \text{Reverse} (\text{Reverse} \; xs) = xs \rightarrow \text{True} \land \text{Reverse} \; (\text{Reverse} \; xs) = xs \]
10. Simplify With "Rule [\text{True} \land P]"
\[ \text{Reverse} (\text{Reverse} \; xs) = xs \rightarrow \text{True} \quad \text{Reverse} \; (\text{Reverse} \; xs) = xs \]
11. Simplify With "Rule [P \rightarrow P]"
\[ \text{True} \]

Table 2.1: An example proof of \( \forall a. \forall x :: List a. \text{Reverse} (\text{Reverse} \; xs) = xs \)

4. Support for the standard library. Many functions in the standard library in Clean are inlined: they are implemented in machine code. The semantics of these functions has to be modeled.

Once this has been achieved, a step further can be taken. The dedicated theorem prover can be integrated in Clean by providing links with the existing development tools for Clean. For example, a link between the editor and the theorem prover can be made. Integration in a programming language can greatly enhance the user-friendliness of a theorem prover.

An integrated theorem prover can easily be used to show the correctness of safety critical applications. Also, programs can be annotated with proven logical statements which describe the behavior of components of the application.

2.3.7 Example proof with graphs and co-induction

Suppose one wants to prove \textbf{Iterate} \texttt{id 1 = Ones(1)} using

\[
\text{Ones} = [1: \text{Ones}] \quad \text{id} \; x = x \\
\text{Iterate} \; f \; x = xs \text{ where } xs = [x: \text{Map} \; f \; xs]
\]

Start by expanding the definitions of \textbf{Iterate} and \textbf{Map} once (2):
Chapter 2: A Proof Tool Dedicated to CLEAN

Now remove the \texttt{Cons 1} start-nodes on both graphs and unfold the definition of \texttt{Map} on the left-hand-side (3). Then use the fact that \texttt{Map id xs = xs} and expand the definition of \texttt{Ones} on the right-hand-side again (4):

This equality is the same as (2). Because in going from (2) to (3) a \texttt{Cons 1} was removed (and thus the step was 'productive'), we can now use (2) to prove (4) by a co-inductive argument. This completes the proof. Note that besides co-induction also cycle-unfolding is used in this proof.

2.4 Conclusions and related work

With the prototype it is possible to prove many interesting theorems about CLEAN programs in an easy way. These theorems can contribute to making programs more reliable. Although there is still a long way to go, the early results are very encouraging. The development of a dedicated theorem prover for CLEAN will continue and we hope to report on some results in the near future on \url{http://www.cs.ru.nl/~maartenm/CleanProverSystem/}.

Related work is described in [Min94], in which a description is given of a proof tool which is dedicated to HASKELL. It supports a subset of HASKELL and needs no guidance of users in the proving process. The user can however not manipulate a proof state himself by the use of tactics, and induction is only applied when the corresponding quantifier has been explicitly marked in advance.

Further related work concerns a theorem prover for HASKELL, called the Equational Reasoning Assistant[Win98], which is still under development. This proof tool is also dedicated to HASKELL and supports HASKELL 1.4. Proofs can only be constructed using equational reasoning and case analysis. No other proof methods, like induction or generalization, are supported. ErA is a stand-alone application.
Abstract. SPARKLE is a new theorem prover written in and specialized for the functional programming language CLEAN. It is mainly intended to be used by programmers for proving properties of parts of programs, combining programming and reasoning into one process. It can also be used by logicians interested in proving properties of larger programs.

Two features of SPARKLE are in particular helpful for programmers. Firstly, SPARKLE is integrated in CLEAN and has a semantics based on lazy graph-rewriting. This allows reasoning to take place on the program itself, rather than on a translation that uses different concepts. Secondly, SPARKLE supports automated reasoning. Trivial goals will automatically be discarded and suggestions will be given on more difficult goals.

This paper presents a small example proof built in SPARKLE. It will be shown that building this proof is easy and requires little effort.

3.1 Introduction

It has often been stated that functional programming languages are well suited for formal reasoning. In practice, however, there is little support for reasoning about functional programs. Existing theorem provers, such as PVS [OSRS01], COQ [The98] and ISABELLE [Pau98], do not support the full semantics of functional languages and can only be used if the program is translated first, making them difficult to use.

Still, formal reasoning can be a useful tool for any programming language. To make reasoning about programs written in the functional programming language CLEAN [Pv98] feasible, SPARKLE was developed. Work on SPARKLE started after a successful experiment with a restricted prototype [dv99a]. SPARKLE is a
Semi-automatic theorem prover that can be used to reason about any Clean program. Sparkle supports all functional concepts and has a semantics based on lazy graph-rewriting. It puts emphasis on tactics which are specifically useful for reasoning about functional programs and automatically provides suggestions to guide users in the reasoning process. Sparkle is written in Clean; with approximately 130,000 lines of source code (also counting libraries and comments) it is one of the larger programs written in Clean. It has an extensive user interface which was implemented using the Object I/O library[A W00]. Sparkle is prepared for Clean 2.0 and will be integrated in the new IDE. Currently, Sparkle is a stand-alone application and can be downloaded at http://www.cs.kun.nl/Sparkle.

The ultimate goal of the project is to include formal reasoning in the programming process, enabling programmers to easily state and prove properties of parts of programs. This on-the-fly proving can only be accomplished if reasoning requires little effort and time. This is already achieved by Sparkle for smaller programs, mainly due to the possibility to reason on source code level and the support for automatic proving.

In this paper a global description of Sparkle and its possibilities will be presented. For this purpose a desired property of a small Clean program will be formulated. It will be shown that building a formal proof for this property is very easy in Sparkle. The specialized features of Sparkle that in particular assist programmers in building this proof will be highlighted.

The rest of this paper is structured as follows. First, the specification language of Sparkle will be introduced and the example program will be expressed in it. A comparison with the specification languages of other theorem provers will be made. Then, the logical language of Sparkle is introduced and the property to prove will be defined in it. Then, a detailed description is given of a proof for this property in Sparkle. Finally, the conclusions and related work are presented.

### 3.2 The specification language of Sparkle

Although Sparkle can be used to prove properties about arbitrary Clean programs, the reasoning process itself takes place on a simplified representation of the Clean program. In this section the Clean program to reason about will be presented and its simplification for Sparkle will be described. Then, the specification of functional programs in other theorem provers will be examined and compared to Sparkle.

#### 3.2.1 The Clean Program

The proof that will be constructed in this paper relates the functions `take` and `drop` by means of the function `++`. These functions are defined in the standard environment of Clean. For this proof, however, the definitions of `take` and
Section 3.2.2: Simplification for Sparkle

Drop have been improved to handle negative arguments more consistently. The next distribution of CLEAN will use the improved definitions.

\[
\begin{align*}
take :: \text{Int} ![a] & \rightarrow [a] & \text{drop} :: \text{Int} ![a] & \rightarrow [a] \\
take \ n \ [x:xs] & | \ n < 1 \ = \ [] & \text{drop} \ n \ [x:xs] & | \ n < 1 \ = \ [x:xs] \\
& | \ \text{otherwise} \ = \ [x:\text{take} \ (n-1) \ xs] & \text{otherwise} \ = \ \text{drop} \ (n-1) \ xs \\
take \ n \ [] & = \ [] & \text{drop} \ n \ [] & = \ [] \\
++ :: ![a] \ [a] & \rightarrow [a] & - :: \text{infix} 6 \ !\text{Int} & \rightarrow \text{Int} \\
++ \ [x:xs] \ ys & = \ [x:xs++ys] & - \ x \ y & = \text{code inline} \ \{\text{sub}1\} \\
++ \ [] \ ys & = \ ys & < :: \text{infix} 4 \ !\text{Int} & \rightarrow \text{Bool} \\
& & < \ x \ y & = \text{code inline} \ \{\text{lt}\}
\end{align*}
\]

These definitions are very straightforward, but it is important to take note of the use of strictness. Because there is no exclamation mark in front of the first argument type of take (or drop), the expression take \ \bot \ [] reduces to [] and not to \bot. Adding an exclamation mark would change this behavior. The exclamation mark in front of the second argument of take (or drop), however, is superfluous, because a pattern match is carried out on this argument.

The functions above also make use of several predefined concepts: the integer type \text{Int}, the integer denotations \text{0} and \text{1}, the list type [a], the nil constructor [] and the cons constructor [_:_]. Furthermore, the auxiliary functions - and <, also from the standard environment of CLEAN, are defined in machine code.

3.2.2 Simplification for Sparkle

Reasoning in Sparkle takes place on Core-CLEAN, which is a subset of CLEAN. Core-CLEAN is a simple functional programming language, basically containing only application, sharing and case distinction. Its semantics is based on lazy graph-rewriting and it supports strictness annotations. Reductions leading to an error and non-terminating reductions are represented by the constant \bot.

Sparkle automatically translates each CLEAN program to Core-CLEAN. For this purpose functions from the source code of the new CLEAN compiler are used. These functions transform CLEAN to a variant of Core-CLEAN which is used internally in the compiler. Translating CLEAN to Core-CLEAN is by no means an easy task and would require a huge effort by hand. Using the real compiler saves a lot of work and has an additional advantage as well: it is trivially guaranteed that the translation preserves the semantics of the program.

The program to reason about is a very basic CLEAN program and can be expressed in Core-CLEAN almost immediately. The only concept used that is not supported by Core-CLEAN is pattern matching. The patterns in the functions therefore have to be transformed to case distinctions. The effect of this translation is minimal and is further reduced by Sparkle, which is able to hide top-level case distinctions and present them as patterns.
The translation of predefined concepts is not a problem, because these are also made available by CORE-CLEAN. The semantics of the representation of numbers, however, is different. SPARKLE disregards overflow and rounding errors, because these would complicate the reasoning process too much. Instead, an idealized representation of numbers is assumed, resulting in an Int type without bounds and a Real type without bounds and with infinite precision.

Translating delta rules, which are functions written in machine code, to CORE-CLEAN is problematic, however. SPARKLE is not able to translate an arbitrary delta rule to CORE-CLEAN. Instead, a fixed set of delta rules occurring in the standard environment of CLEAN is recognized. The translation of recognized delta rules is hard-coded in the theorem prover, usually by referring to mathematical definitions working on idealized numbers. This is for example the case for the subtract function from the example program.

3.2.3 Specification in other theorem provers

In order to reason about a program in a theorem prover, it must first be translated to its specification language, which is CORE-CLEAN for SPARKLE. This language is a very important aspect of a theorem prover, because the reasoning process takes place on the translated program in the specification language.

For effective reasoning, a good understanding of the translated program is required. Programmers usually understand the programs they write very well, but this may not be the case for the translated version. If the differences are too big, knowledge of the original program is completely lost and proving will be a lot more difficult. Moreover, a new specification language must be mastered. These obstacles will likely lead to programmers giving up on formal reasoning.

Unfortunately, there is still a big gap between an executable (programming) language which is useful in practice and a formal (specification) language which is useful in theory. Differences between the specification language and the programming language are inevitable. The following differences can be distinguished, in decreasing order of importance:
Section 3.2.4: Suitability of Core-CLEAN for reasoning about CLEAN

1. **Differences in semantics.** These are quite serious, because understanding the translated program may become very difficult. Firstly, the concepts in the specification language may not be known to the programmer. Secondly, the relation between the original program and its translation may be lost, making it difficult to re-use the expertise of the original program.

2. **Differences in notational expressivity.** Sometimes complicated concepts have to be translated to simpler ones, such as translating notational sugar to ordinary function applications. These differences can again make it difficult to relate the translated program to the original program.

3. **Differences in syntax.** These are not so serious and can often be solved easily. However, it can still be very annoying to programmers.

The specification languages of existing theorem provers are very powerful but score badly on the points mentioned above. Most importantly, there are usually many differences in semantics. For instance, Coq supports both reasoning about finite (inductive) and infinite (co-inductive) objects, but these objects can not be combined into one datatype. Strictness annotations are not supported by any existing theorem prover. Writing a translation from CLEAN to for instance the specification language of PVS would require a huge effort and may in fact be as difficult as developing a new theorem prover.

All in all, using an existing theorem prover to reason about CLEAN programs is very problematic for programmers.

### 3.2.4 Suitability of Core-CLEAN for reasoning about CLEAN

Reasoning in Sparkle takes place on Core-CLEAN, which is not a new language but only a subset of CLEAN:

![Diagram](image)

**Figure 3.2:** Reasoning takes place in a subset of the programming language

In contrast to the specification language of other theorem provers, CORE-CLEAN is very similar to CLEAN. There will not be many differences between a CLEAN program and its simplification in CORE-CLEAN:

1. **Semantics.** CORE-CLEAN borrows its semantics from CLEAN [BS98], using a lazy term-graph rewriting system to reduce expressions. All programs written in CORE-CLEAN are valid CLEAN as well and will therefore easily be understood by experienced CLEAN programmers. The only difference in semantics lies in the handling of numbers. This is only a problem for
programs in which overflow or rounding occurs. If one wants to reason about these programs, a different representation of numbers must be chosen.

2. Concepts. In CORE-CLEAN all basic constructs of CLEAN are available. Notational sugar is translated to these basic concepts, including pattern matching (translated to case distinctions), overloading (translated to dictionaries), dot-dot-expressions (translated to functions) and comprehensions (translated to functions). The translated versions are usually recognized and understood easily by programmers, because they are not that different. There is, however, one exception: the translation of comprehensions to functions is not transparent at all. The functions created here are hard to understand and almost impossible to relate to the original program.

3. Syntax. CORE-CLEAN uses the same syntax as CLEAN.

Due to these similarities, CORE-CLEAN is a good specification language for reasoning about CLEAN programs. The translation of comprehensions is, however, still problematic. This could be solved by using a different translation-scheme or by interpreting comprehensions; further investigation is required here.

3.3 The specification of the property

Properties can be specified in SPARKLE using a simple first-order propositional logic which is extended with equalities on expressions. The logical connectives $\neg, \rightarrow, \land, \lor, \leftrightarrow$ and the quantors $\forall, \exists$ are available. Quantification, either existential or universal, is possible over propositions and expressions of an arbitrary type. Predicates and quantification over predicates are not allowed.

A standard semantics for propositional logics is used. The semantics of the equality on expressions is defined using the operational reduction semantics of CLEAN. Two expressions are equal if for all reductions of one expression there exists a reduction of the other expression that produces the same constructors and basic values (and possibly more). This semantics covers both the equality between finite and infinite structures.

SPARKLE offers the following features to make the specification of properties as easy as possible:

- The same syntax may be used as in CLEAN, meaning that infix applications are allowed and no superfluous brackets have to be supplied.
- Top-level universal quantors may be omitted. For each free variable in the proposition, a top-level universal quantor will automatically be created by SPARKLE.
- It is optional to specify the types of the variables in a $\forall$ or $\exists$. If the type is left out, it will be inferred by the theorem prover. The property will always be type-checked.
• Quantification over type-variables is implicit and must not be specified. Properties will always be interpreted as polymorphic as possible.

The property that is going to be proved in this paper relates the functions \texttt{take} and \texttt{drop}. Using the described features it can be specified as follows:

\[
\text{take } n \; \texttt{xs} \; \text{++} \; \text{drop } n \; \texttt{xs} = \texttt{xs}
\]

There is, however, a problem with this property. If \( n = \bot \) and \( \texttt{xs} = \{7\} \), the left-hand-side of the equation will reduce to \( \bot \) while the right-hand-side is \( \{7\} \). These kind of problems with undefined expressions occur frequently and can be very hard to detect beforehand. They will always be revealed in the reasoning process, though. An easy solution is to simply demand that \( n \) is always defined:

\[
n \neq \bot \rightarrow \text{take } n \; \texttt{xs} \; \text{++} \; \text{drop } n \; \texttt{xs} = \texttt{xs}
\]

This property contains two free variables, \( n \) and \( \texttt{xs} \), for which universal quantors will be created automatically by \textsc{Sparkle}. The type of \( n \) will be inferred as \texttt{Int} and the type of \( \texttt{xs} \) will be inferred as \( \{a\} \). A universal quantor for the type variable \( a \) will be omitted. This results in the following property, which will be the starting point of the proof:

\[
\forall n \in \texttt{Int} \forall x : \{a\} | n \neq \bot \rightarrow \text{take } n \; x : \{a\} \; \text{++} \; \text{drop } n \; x : \{a\} = x : \{a\}
\]

### 3.4 Building the proof

In this section the process of building a proof for the given property in \textsc{Sparkle} will be described. Before the proof itself is given, the reasoning style of \textsc{Sparkle} (how proofs are constructed) and its hint mechanism (a mechanism to assist users in building proofs) will be explained.

#### 3.4.1 Reasoning style in \textsc{Sparkle}

Reasoning in \textsc{Sparkle} is similar to reasoning in other theorem provers and consists of the repeated application of tactics on goals until all goals are discarded.

A \textit{goal} is a property that still has to be proven. Each goal is associated with a \textit{goal context}. In a goal context variables are declared and local hypotheses are stored. The \textit{proof state} consists of a list of goals. The active goal being manipulated is called the \textit{current goal}; the others are called \textit{subgoals}. Changing the active goal is always allowed.

A \textit{tactic} is a function from a single goal to a list of goals. Applying a tactic on the current goal will lead to a new proof state, which consists of the created goals and the old subgoals. All tactics must be \textit{sound} with respect to semantics, meaning that the validity of the created goals must logically imply the validity of the original goal.

\textsc{Sparkle} implements a total of 42 tactics. Although all of these tactics can also be found, or expressed, in other theorem provers, their behavior is
specifically geared towards proving properties of lazy functional programs. The *Induction* tactic, for example, can only be applied to admissible propositions (see [Pau87]) and is valid for both finite and infinite structures.

A proof of the example property can be constructed using a subset consisting of eight tactics, which are: (1)*Contradiction* (proof by contradiction); (2)*Definedness* (use absurd hypotheses concerning $\bot$); (3)*Induction* (structural induction); (4)*Introduce* (elimination of $\forall$ and $\rightarrow$); (5)*Reduce* (reduction to root-normal-form); (6)*Reflexive* (prove reflexive equality); (7)*Rewrite* (rewrite according to a hypothesis); (8)*SplitCase* (case distinction). See (global) Appendix A for a more detailed description of these tactics.

### 3.4.2 The hint mechanism

Successfully building a proof in *Sparkle* depends on the selection of the right tactics. For this, knowledge of the available tactics and their effect is needed, as well as expertise in proving. To make the selection of tactics easier, a hint mechanism is available in *Sparkle*.

The hint mechanism is activated each time the current goal changes. It automatically produces a list of applicable tactics. Based on built-in heuristics only the most important applicable tactics are suggested. Each tactic is assigned a score between 1 and 100 that indicates the likelihood of that tactic being helpful in the proof. A score of 100 is reserved for tactics that prove the current goal in one step. The assignment of scores to tactics is hard-coded in *Sparkle*.

The hint mechanism is a valuable tool, especially for those with little expertise in proving. However, it is by no means a failsafe feature. Sometimes the right tactic is not suggested or several wrong tactics get high scores. Programmers can use the mechanism to their advantage but should not completely rely on it. Future work will concentrate on improving the hint mechanism.

On top of making users aware of useful applicable tactics, there are two additional advantages offered by the hint mechanism:

1. Suggested tactics are assigned a hot-key and can be applied instantly. This reduces the typing (or clicking) effort for building proofs considerably.

2. A threshold for automatic application can be set. If the best applicable tactic has a score higher than this threshold, it will be applied automatically. This process continues until no tactic with a high enough score can be found. A low threshold can be used for automatic proving; a medium threshold for semi-automatic proving and a high threshold for manual proving.

### 3.4.3 Proof of the example program

In this subsection a proof of the example property built with *Sparkle* will be presented. The description will focus on the goals that have to be proved. At each goal, a tactic to be applied is chosen. An argument for this choice will be given. The description then continues with the first goal that is created; if
Section 3.4.3: Proof of the example program

several goals are created, they will be proved later. The order in which the goals are proved is the same as in SPARKLE. (to be more precise: all unproved goals are stored in a proof tree, which is traversed from left to right and top-down). A numbering system is used to keep track of the goals.

The initial goal is simply the property to be proven. It has an empty context.

\[ \forall n \in \mathbb{Int} \forall x \in [a] [n \neq \bot \rightarrow \text{take } n \ x \ x \ +\ n \ x \ x \ =\ x\] (1)

Because of the definitions of \texttt{take} and \texttt{drop}, which are tail-recursive in the list argument, structural induction on \( x \) is likely to be useful here. This is accomplished by applying the tactic \texttt{Induction \( x \)}. Three new goals(1.1,1.2,1.3) are created: one for the case that \( x \) is \( \bot \); one for the case that \( x \) is \( [] \) and one for the case that \( x \) is a non-empty list. Note that \( \bot \) is treated as a constructor for all algebraic types; therefore induction creates three new goals instead of two.

\[ \forall n \in \mathbb{Int} [n \neq \bot \rightarrow \text{take } n \bot \ +\ n \bot \ =\ \bot] \] (1.1)

The goal context is used to store introduced variables and hypotheses. It is actually just a prettier representation of a chain of \( \forall \)'s and \( \rightarrow \)'s, which allows the reasoning to focus on the interesting part of the goal. Another induction is not needed in the current goal. The variable \( n \) and the hypothesis \( n \neq \bot \) can therefore safely be moved to the goal context using the tactic \texttt{Introduce \( n \ H1 \)}.

\[ \frac{n \in \mathbb{Int}}{\text{H1: } n \neq \bot} \] (1.1′)

Due to the strictness of \texttt{take} and \texttt{++} and the presence of \( \bot \) arguments, redexes are present in the current goal. The tactic \texttt{Reduce NF All} can be used to reduce all redexes in the current goal to normal form (eager reduction). With other parameters, the tactic \texttt{Reduce} can also be used for stepwise reduction, lazy reduction, reduction of one particular redex and reduction in the goal context.

\[ \frac{n \in \mathbb{Int}}{\text{H1: } n \neq \bot} \] (1.1′′)

This is clearly a trivial goal, because equality is a reflexive relation. Such reflexive equalities are proved immediately with the tactic \texttt{Reflexive}.

\[ \forall n \in \mathbb{Int} [n \neq \bot \rightarrow \text{take } \bot \ +\ n \bot \ =\ \bot] \] (1.2)

This is the second case of the induction, created for the case that \( x = \bot \). Again, introduction in the context should be done first: \texttt{Introduce \( n \ H1 \)}
\[ n \in \text{Int} \]
\[ \text{H1: } n \neq \bot \]
\[ \text{take } n \[] ++ \text{drop } n \[] = [] \]

(1.2')

There are again redexes present in the current goal, due to the pattern matching performed by take and drop. Therefore: \textbf{Reduce NF All}

\[ n \in \text{Int} \]
\[ \text{H1: } n \neq \bot \]
\[ [] = [] \]

(1.2'')

This is another example of a reflexive equality; therefore \textbf{Reflexive}.

\[ \forall x \in a \forall xs \in [a] \]
\[ \forall n \in \text{Int} [n \neq \bot \rightarrow \text{take } n \; xs \; ++ \; \text{drop } n \; xs \; = \; xs] \]
\[ \rightarrow \forall n \in \text{Int} [n \neq \bot \rightarrow \text{take } n \; [x:xs] \; ++ \; \text{drop } n \; [x:xs] \; = \; [x:xs]] \]

(1.3)

This is the third goal created by the induction; the induction step. The current goal looks quite complicated, but introduction can make things a lot clearer. For reasons of clarity, the first hypothesis will be called IH (induction hypothesis) and the variable \( n \) will be introduced as \( m \) (to avoid name conflicts with the \( n \) already present in the induction hypothesis): \textbf{Introduce} \( x \; xs \; \text{IH} \; m \; \text{H1} \).

\[ x \in a, xs \in [a], m \in \text{Int} \]
\[ \text{IH: } \forall n \in \text{Int} [n \neq \bot \rightarrow \text{take } n \; xs \; ++ \; \text{drop } n \; xs \; = \; xs] \]
\[ \text{H1: } m \neq \bot \]

\[ \text{take } m \; [x:xs] ++ \text{drop } m \; [x:xs] = [x:xs] \]

(1.3')

Again, the current goal contains redexes that can be removed by applying the tactic \textbf{Reduce NF All}. Note that a lazy reduction (to root-normal-form) will not suffice here, because ++ is lazy in its second argument and therefore \text{drop} \( m \; [x:xs] \) as a whole will not be reduced at all.

\[ x \in a, xs \in [a], m \in \text{Int} \]
\[ \text{IH: } \forall n \in \text{Int} [n \neq \bot \rightarrow \text{take } n \; xs \; ++ \; \text{drop } n \; xs \; = \; xs] \]
\[ \text{H1: } m \neq \bot \]

\[ \text{case } (m \; < \; 1) \; \text{of} \]
\[ \begin{align*}
\text{True} & \rightarrow [] \\
\text{default} & \rightarrow [x:\text{take} \; (m-1) \; xs]
\end{align*} \]
\[ ++ \]
\[ \begin{align*}
\text{case } (m \; < \; 1) \; \text{of} \\
\text{True} & \rightarrow [x:xs] \\
\text{default} & \rightarrow \text{drop} \; (m-1) \; xs
\end{align*} \]

\[ = \]
\[ [x:xs] \]

(1.3'')

(This proof state is also shown in Fig. 3.3.)

The natural next step is a case distinction on \( m \; < \; 1 \), because that will allow the reduction of both case-expressions in the current goal. A special tactic is
used for this purpose: \textbf{SplitCase 1}. This tactic will examine the first case-expression in the current goal. Three cases are distinguished: (1) $\bot$ (for when $m < 1$ can not be properly evaluated); (2) \textbf{True} (for the first alternative); (3) \textbf{False} (for the default alternative). For each case a new goal (1.3.1, 1.3.2, 1.3.3) is created, in which the appropriate alternatives of the case-expressions are chosen. Also, in each goal hypotheses are introduced to reflect the case chosen.

Figure 3.3: The theorem prover in action

\[
x \in a, xs \in [a], m \in \text{Int} \\
\text{IH: } \forall n \in \text{Nat} [n \neq \bot \rightarrow \text{take } n \text{ xs }++ \text{drop } n \text{ xs } = xs] \\
\text{H1: } m \neq \bot \\
\text{H2: } (m < 1) = \bot \\
\bot ++ \bot = [x:xs]
\]

This is the goal created by \textbf{SplitCase} for the case that $m < 1 = \bot$. This goal can be proved in one step, because hypotheses H1 and H2 are contradictory. This is due to the totality of $<$, which ensures that $x < y$ can only be $\bot$ if either $x = \bot$ or $y = \bot$. Hypothesis H2 states that $m < 1 = \bot$, thus either $m = \bot$ or $1 = \bot$. Of course, $1 = \bot$ is not true, thus from hypothesis H2 it may be concluded that $m = \bot$. This contradicts with hypothesis H1. In \textsc{Sparkle}, a specialized tactic is available to handle these cases: \textbf{Definedness}. This tactic
Chapter 3: Theorem Proving for Functional Programmers

searches for expressions (most notably, variables) that are defined (known to be unequal to ⊥) and expressions that are undefined (known to be equal to ⊥). The analysis makes use of the hypotheses, the ordinary strictness information of functions and the totality of functions such as − and <. If an expression is found which is both defined and undefined, the goal is proved by contradiction.

\[
\begin{align*}
\forall n \in \mathbb{N} \quad & \text{(IH)} \\
\forall n \in \mathbb{N} & \quad \text{take } n \text{ xs ++ drop } n \text{ xs = xs} \\
\text{H1: } m \neq \bot & \quad \text{H2: (m < 1) = True} \\
\end{align*}
\]

This is the goal created by SplitCase for the case that \( m < 1 = \text{True} \). The appropriate case alternatives have been chosen and the resulting goal is clearly a trivial one. It can be proved by a reduction followed by an application of Reflexive. These two can be combined by \text{Reduce NF All; Reflexive}.

\[
\begin{align*}
\forall n \in \mathbb{N} & \quad \text{take } n \text{ xs ++ drop } n \text{ xs = xs} \\
\text{H1: } m \neq \bot & \quad \text{H2: (m < 1) = False} \\
\end{align*}
\]

This is the goal created by SplitCase for the case that \( m < 1 = \text{False} \). Filling in the proper case alternatives has resulted in a goal which contains a redex (++] can be reduced); therefore: \text{Reduce NF All}.

In this goal it is finally possible to use the induction hypothesis, using \((m-1)\) as value for \( n \). This results in the substitution of \text{take } (m-1) \text{ xs ++ drop } (m-1) \text{ xs by xs in the current goal. This is accomplished in SPARKLE by the tactic Rewrite IH. This tactic will create two new goals, one for the goal after substitution (1.3.3.1) and one for the condition } m-1 \neq \bot (1.3.3.2).\]

\[
\begin{align*}
\forall n \in \mathbb{N} & \quad \text{take } n \text{ xs ++ drop } n \text{ xs = xs} \\
\text{H1: } m \neq \bot & \quad \text{H2: (m < 1) = False} \\
\end{align*}
\]

This trivial goal is proved immediately by \text{Reflexive}.
The presented proof was not difficult to build. An examination of the current goal always resulted in a tactic to apply; no overview of the proof as a whole was ever required. This actually turns out to be the case for many small proofs about functional programs.

The hint mechanism is especially useful for building such 'goal-directed' proofs. In fact, \textit{all steps in the presented proof were given as hints by Sparkle}. Building the proof is therefore reduced to selecting hints, which is a lot easier than selecting tactics, simply because there are far less options to choose from. Right now, there are 42 different tactics which can have arguments as well, whereas there are typically less than 15 hints given for a small-sized goal.

Automatic proving is possible in Sparkle by letting it automatically apply the hint with the highest score. \textit{The example property can be proved automatically with the hint mechanism}. Of course, larger and more difficult proofs cannot be built automatically, although often suggestions given by Sparkle can be used successfully. Further improving the hint mechanism will be one of the spearheads in the further development of Sparkle.

A proof of (almost) the same property is also presented in Bird’s Introduction to Functional Programming\cite{Bird}. The proof presented there only takes positive integer arguments into account, but is otherwise quite similar. Note that building such a formal proof with the aid of a theorem prover is much easier than doing it on paper. In \cite{Bird}, a lot of proofs of properties about functional programs are given. A lot of these proofs (about 80\%) have already
been translated to SPARKLE without difficulties. No problems are expected for translating the others.

3.5 Conclusions and further work

Building the proof required little effort and little expertise. The proving action could always be found by examining the current goal and following a few ground rules. The theorem prover is able to follow these same ground rules and suggest the correct tactics to users, reducing the required expertise even more. All in all, a programmer can build this proof in a short time and without many difficulties.

The two features of SPARKLE that contribute the most to this are:

- The possibility to reason about the source program. Starting with proving is trivial: state what you want to prove and run the theorem prover.
- The hint mechanism. Selecting suggested hints is very easy. An application of a hint can easily be undone, making playing with hints possible. This is not only a fast way of learning how to use the system, but also a fast way of actually constructing the proof.

There are, however, lots of things that still need to be done. Although SPARKLE can already be used to build proofs, it is by no means finished. For instance, documentation must still be added to the system. Furthermore, the hint mechanism must be compared to the automatic reasoning abilities of other theorem provers and possibilities to improve the mechanism must be researched.

Also, work needs to be done on the formal framework of the theorem prover. The effect of the tactics must be described formally in this framework and their soundness with respect to the semantics of CLEAN must be proved. Of particular importance is the soundness of Induction for all lazy structures.

3.6 Related work

In many textbooks (for instance [Bir98]) properties about functional programs are proved by hand. Also, several articles (for instance [BS01]) make use of reasoning about functional programs. It seems worthwhile to attempt to formalize these proofs in SPARKLE. In programming practice, however, reasoning about functional programs is scarcely used.

Widely used generic theorem provers are PVS [OSRS01], COQ [The98] and ISABELLE [Pau98]. They are not tailored towards a specific programming language. Reasoning in these provers requires using a syntax and semantics that are different from the ones used in the programming language. For instance, strictness annotations as in CLEAN are not supported by any existing theorem prover. This makes it rather hard for a programmer to use. On the other hand, these well established theorem provers offer features that are not available in SPARKLE. Most notably, the tactic language and the logic are much richer than in SPARKLE.
Somewhat closer related work is described in [Min94], in which a description is given of a proof tool which is dedicated to Haskell [PH99]. It supports a subset of Haskell and needs no guidance of users in the proving process. The user can however not manipulate a proof state by the use of tactics to help the prover in constructing a proof, and induction is only applied when the corresponding quantifier has been explicitly marked in advance.

Further related work concerns a proof tool specialized for Haskell, called ERA [Win98], which stands for Equational Reasoning Assistant. This proof tool is still in development, although a working prototype is available. ERA, however, is intended to be used for equational reasoning, and not for theorem proving in general. Additional proving methods, including induction or any logical tactics, are not supported. ERA is a stand-alone application.

Another theorem prover which is dedicated to a functional programming language is EVT [NFG01], the Erlang Verification Tool. It differs from Sparkle because Erlang is a strict, untyped language which is mainly used for developing distributed applications. EVT has been applied in practice to larger examples.

We do not know of any other theorem prover than Sparkle that is integrated, tailored towards a lazy functional language and semi-automatic.
Chapter 4

Maarten de Mol, Marko van Eekelen, Rinus Plasmeijer:

Proving properties of lazy functional programs with Sparkle

Presented at CEFP'07, published in LNCS tutorial proceedings volume 5161.

Abstract. This tutorial paper aims to provide the necessary expertise for working with the proof assistant Sparkle, which is dedicated to the lazy functional programming language Clean. The purpose of a proof assistant is to use formal reasoning to verify the correctness of a computer program. Formal reasoning is very powerful, but is unfortunately also difficult to carry out.

Due to their mathematical nature, functional programming languages are well suited for formal reasoning. Moreover, Sparkle offers specialized support for reasoning about Clean, and is integrated into its official development environment. These factors make Sparkle a proof assistant that is relatively easy to use.

This paper provides both theoretical background for formal reasoning, and detailed information about using Sparkle in practice. Special attention will be given to specific aspects that arise due to lazy evaluation and due to the existence of strictness annotations. Several assignments are included in the text, which provide hands-on experience with Sparkle.

4.1 Introduction

In 2001, the distribution of the lazy functional programming language Clean [BvvP87, Pv99, Pv01] was extended with the dedicated proof assistant Sparkle [dvP02]. The purpose of a proof assistant is to verify the correctness of a computer program without executing it. This is accomplished by means of the mathematical process of formal reasoning, which makes use of the source code of the program and the semantics of the programming language.

Sparkle is intended as an additional tool for the Clean programmer and aims to make formal reasoning accessible. It is conveniently integrated into the official Development Environment of Clean, allows reasoning on the level of
the programming language itself and offers dedicated support for dealing with
CLEAN programs. Unfortunately, formal reasoning is a complex mathematical
process that requires specialized expertise. Therefore, it is often still difficult to
carry out, even in dedicated proof assistants such as SPARKLE.

In practice, SPARKLE has already been applied for various purposes. It
has been used for proving properties of I/O-programs by Butterfield[BS01] and
Dowse[DBv05]. In [THK06, HKT03], Tejfel, Horváth and Kozsik have proposed
an extension for it for dealing with temporal properties. Support for class-
generic properties has been added to it by van Kesteren[vvd04]. Furthermore,
it has also been used in education at the Radboud University of Nijmegen.

The purpose of this paper is to provide the information that is necessary
for functional programmers to start making use of SPARKLE. A combination of
both theoretical and practical expertise will be provided. No special knowledge
is required to understand the contents of this paper: a basic understanding of
lazy functional languages and elementary logic suffices. Upon completion of this
paper, the reader will be able to use SPARKLE to prove basic properties of small
CLEAN programs with minimal effort. Furthermore, a solid foundation will be
laid for proving properties that are more complex.

This paper is structured as follows. First, the concept of formal reasoning
will be explained independently of SPARKLE in Section 4.2. Then, the important
design principles of SPARKLE will be summarized in Section 4.3, and their effect
on the way that formal reasoning is implemented will be explained. Then, in
Sections 4.4 and 4.5 a tutorial of the use of SPARKLE in practice is presented.
The first part (Section 4.4) presents a step-by-step introduction of all the basic
features of SPARKLE; the second part (Section 4.5) describes several advanced
features that are specific for SPARKLE. We discuss related work in Section 4.6
and draw conclusions in Section 4.7.

The tutorial is written in an explanatory style and contains assignments with
which the provided theory can be put into practice. The assignments require
the standard CLEAN 2.2 distribution to be installed, and the SPARKLE version
merged in it. The worked out answers to the assignments are available online

4.2 Formal reasoning

In the following sections, a general introduction to formal reasoning will be pre-
sented independently of SPARKLE. In Section 4.2.1, formal reasoning will first
be described as an abstract process that transforms input to desired output. In
Section 4.2.2, the underlying formal framework will be identified; this framework
is a prerequisite for carrying out formal reasoning. The most important
component of the framework is the proof language, which will be explored in
more detail in Section 4.2.3. Finally, the soundness of formal reasoning will be
discussed in Section 4.2.4.
4.2.1 The abstract process of formal reasoning

Formal reasoning is a mathematical process that fully takes place on the formal level. The goal of formal reasoning is to verify the correctness of some kind of formal object by means of reasoning about it. The process as a whole can roughly be characterized as follows:

1. Formalize an object \( o \);
2. Formalize a property \( p \) that says something about \( o \);
3. Build a formal proof that shows that \( p \) holds for \( o \).

If formal reasoning succeeds and a formal proof is built, then it is shown with absolute certainty that the formalized object \( o \) behaves as specified by means of property \( p \). This holds for all environments in which \( o \) may occur, because the formal proof is obliged to take all possible circumstances into account. As such, a positive result of formal reasoning is more powerful than for instance a positive result of testing, which is restricted by the test-set that was used.

If formal reasoning does not succeed in building a proof, however, then not much information has been gained. It may either be the case that \( o \) is incorrect, or it may be the case that the desired behavior of \( o \) was incorrectly specified by \( p \), or it may simply be the case that the proof builder did not build the proof in the right way. A negative result of formal reasoning is hard to interpret correctly and is therefore less useful than a negative result of for instance testing.

4.2.2 Formal framework

Formal reasoning makes use of the formal representations of the object to reason about (input), the property to prove (input) and the proof to be built (output). Moreover, to ascertain the soundness of reasoning (see Section 4.2.4), a formal semantics that assigns a meaning to properties must be available as well. The combination of these prerequisites of reasoning will be called a formal framework:

Definition 4.2.2: (formal framework)
A formal framework is a tuple \((O, P, \models_o, \vdash_o)\) such that:

- \( O \) is the set that contains all possible objects to reason about;
  - \( o \in O \) denotes that \( o \) is a valid object to reason about
- \( P \) is the set that contains all possible properties that may be specified;
  - \( p \in P \) denotes that \( p \) is a valid property to prove
- \( \models_o \) is the relation that defines the semantics of properties;
  - \( \models_o p \) denotes that \( p \in P \) holds in the context of \( o \in O \)
- \( \vdash_o \) is the derivation system that defines proofs of properties.
  - \( \vdash_o p \) denotes that a proof of \( p \in P \) exists in the context of \( o \in O \)

The formal framework of SPARKLE is described completely in \([dvP07a]\). In the remainder of this paper, it will be treated implicitly only.
Note that the elements of a framework are connected: it must be possible to refer to components of objects within properties; the semantics of a property can only be determined in the context of a given object; and the derivation of a proof depends on a given object as well.

Using the notations introduced by the formal framework, formal reasoning can now be characterized as follows:

**Definition 4.2.2:** \((formal\ reasoning)\)

Formal reasoning is the process that given a formal framework \((O, P, \models_o, \vdash_o)\), a specific object \(o \in O\) and a specific property \(p \in P\), attempts to determine whether \(\vdash_o p\) holds or not. From the soundness of the formal framework it then follows that \(\models_o p\) holds as well.

In other words, the goal of formal reasoning is to determine \(\models_o\) by means of \(\vdash_o\). This approach only makes sense for frameworks in which \(\vdash_o\) is less complicated than \(\models_o\), which is often the case, because derivation systems are usually simpler than semantic relations.

### 4.2.3 Proof language

The most important component of the formal framework is the proof language, which is usually represented by means of a derivation system. The derivation rules of this system are reasoning steps that form the building blocks of proofs. Building proofs is basically the repeated application of these reasoning steps, and can be characterized as follows:

- **Goal**: prove a property \(p\).

- **Apply**: reasoning step \(R\). This transforms \(p\) to \(p_1, \ldots, p_n\). If \(n = 0\), then the proof is complete (\(R\) proves \(p\)). Otherwise, \(p_1, \ldots, p_n\) become the new goals which all have to be proved recursively by the same reasoning process.

- **Goal**: prove all properties \(p_1, \ldots, p_n\).

In other words, reasoning steps are functions that transform propositions into (possibly more) propositions, and the proof language is the set of functions that one is allowed to apply during reasoning. Furthermore, reasoning itself is ‘goal-busting’: at each point in time a number of propositions (goals) have to be proved, and these propositions can be simplified (busted) by means of the repeated application of predefined reasoning steps.

The result of reasoning is a derivation tree in which the nodes are propositions (and the root node is the initial proposition to prove) and each set of edges leading from a single node corresponds with a reasoning step. Edges in this tree do not necessarily have to lead to another node, because reasoning steps may produce the empty list of propositions. The leaves of the tree are the propositions that still have to be proved.
The derivation tree is of course the formal representation of a proof. It can easily be serialized, provided that the reasoning steps are named. A serialized proof can be transferred to anyone with knowledge of the formal framework that it uses. Furthermore, the receiver can even automatically check the validity of the proof by re-running it. Note that validating proofs is easy, because it only requires the formal framework, but building proofs is difficult, because it requires the continuous selection of the ‘right’ reasoning step.

### 4.2.4 Soundness of formal reasoning

Building formal proofs is an exercise in the repeated simplification of propositions according to predefined reasoning steps. This, however, is a purely syntactic exercise that does not take the actual meaning of propositions into account in any way. In order for the results of reasoning to be meaningful, the underlying formal framework must be sound as well:

**Definition 4.2.4.1:** *(soundness of formal frameworks (1))*

A formal framework \((O, P, \vdash_o, \models_o)\) is sound if for all \(o \in O\) and \(p \in P\) it holds that \(\vdash_o p\) implies \(\models_o p\).

Because \(\vdash_o\) is composed of individual derivation rules, the soundness of a formal framework as a whole can be determined by verifying these rules as follows:

**Definition 4.2.4.2:** *(soundness of a derivation rule)*

A derivation rule \(R \in \vdash_o\) is sound if for all \(p \in P\) it holds that \(\vdash_o (p_1 \land \ldots \land p_n)\) implies \(\models_o p\), assuming that \(R(p) = p_1, \ldots, p_n\).

**Definition 4.2.4.3:** *(soundness of formal frameworks (2))*

A formal framework \((O, P, \vdash_o, \models_o)\) is sound if all its derivation rules \(R \in \vdash_o\) are sound.

Formal reasoning only makes sense if the underlying formal framework is sound. Soundness should therefore preferably be proved explicitly. If the complexity of the derivation system makes this too difficult, then some degree of confidence can still be gained from practice (‘no untrue propositions have ever been proved, so it must be correct’), but this weakens the results of formal reasoning considerably. The soundness of the formal framework of Sparkle has been proved in [dvP07a].

Finally, note that for the usefulness of formal reasoning it is important that the reverse property of completeness (for all properties \(p\), \(\vdash_o p\) implies \(\models_o p\)) holds too. Full completeness is extremely difficult to achieve for complex frameworks. Using proof theory, however, it can usually be approximated quite closely.

### 4.3 Design principles of Sparkle

The main purpose of Sparkle is to allow functional programmers to reason about the Clean programs that they are developing, which improves the quality
of the program as a whole. The reasoning support that SPARKLE offers is in the first place tailored towards this main purpose, although in general SPARKLE is also useable for anyone who would like to reason about functional programs. In particular, a frontend for HASKELL’98 is currently being added to CLEAN, which in the future would allow reasoning about mixed CLEAN/HASKELL-programs.

In the following sections, the effect that the main purpose of SPARKLE has on its design will be explored closely. In Section 4.3.1, first the intended users of SPARKLE will be analyzed in detail. Then, in Section 4.3.2 a list of resulting consequences for the design will be presented. Finally, the important consequence of dedicated reasoning will be explored in detail in Sections 4.3.3 and 4.3.4.

4.3.1 Intended users: functional programmers

The intended users of SPARKLE are functional programmers, or more specifically anyone who has downloaded the CLEAN distribution and is developing programs with it. Of course, there is much diversity in this group, and there is no such thing as ‘the functional programmer’. Still, for the sake of design, we will make the following tentative assumptions about the intended users of SPARKLE:

- they do not necessarily have much experience with formal reasoning, and may not even know about it at all;
- they often have some theoretical background, and usually have at least a basic understanding of elementary logic;
- they usually have good knowledge of functional programming in general and of CLEAN (and its semantics) in specific;
- they are not necessarily aware of the benefits of formal reasoning for the purpose of improving the quality of software;
- they are mainly interested in the programs that they develop.

Other proof assistants may be geared towards different users; for instance, the major independent proof assistants (such as for instance PVS [OSRS01] and COQ [The06]) are mainly intended for logicians who already know about formal reasoning and are interested in it as well.

4.3.2 Design choices

SPARKLE implements a theoretically sound formal framework, and therefore fully supports general formal reasoning on the fundamental level. In its design, however, SPARKLE focuses mainly on functional programmers as its intended users. The most important choices in the design of SPARKLE are:

- The object language should be CLEAN, because this allows programmers to reason on the level of the programming language, which is their area of
Section 4.3.2: Design choices

expertise. Although this has not been realized fully, a good approximation by means of Core-CLEAN has been adopted by Sparkle (see Section 4.3.4).

- For the property language, it suffices to use a standard first-order logic which has been extended with an equality on arbitrary program expressions. In such a logic most common properties can be expressed easily. Moreover, functional programmers are likely to be capable of handling standard first-order logic. The property language will be introduced in the tutorial in Section 4.4.3.

- The semantics of the property language should conform to the semantics of CLEAN. This ensures that properties that are proved with Sparkle hold for the real-world CLEAN program as well. This is achieved by giving \( e_1 = e_2 \) the meaning ‘it is possible to interchange \( e_1 \) with \( e_2 \) in any program without changing its observational behavior’. The full semantics will be introduced on an informal level in the tutorial in Section 4.4.4.

- Formal reasoning should be integrated with programming, such that switching between the two activities becomes easy. This makes formal reasoning more attractive, because it is linked to an activity that is carried out continuously. The integration of Sparkle is realized by allowing it to be started directly from the IDE (Integrated Development Environment) of CLEAN, in which case the current project is loaded automatically in Sparkle.

- The reasoning steps of Sparkle should be specialized for dealing with lazy functional programs in general, and for dealing with CLEAN in specific. In particular, lazy evaluation and explicit strictness have a profound influence on semantics, and therefore on reasoning as well. The specialized features of Sparkle will be described in Section 4.5.

- The first impression of Sparkle should be positive, and should entice programmers to continue with formal reasoning. This is realized by Sparkle’s attractive user interface (see tutorial), and by allowing small proofs to be carried out automatically with the hint mechanism (see Section 4.4.5).

- Sparkle should have up-to-date and extensive documentation. This paper is the first attempt to achieve this goal.

The design choices with the most profound influence on Sparkle are the level of the object language and the specialization of the reasoning steps. The consequences of the level of the object language will be examined further in Sections 4.3.3 and 4.3.4; the specialized features of Sparkle will be described in detail later in Section 4.5.
4.3.3 Dedicated vs general-purpose formal reasoning

If one wants to add support for formal reasoning to a specific programming language, two different approaches can be taken:

1. Build one’s own dedicated proof assistant that directly supports reasoning on the level of the programming language itself; or

2. Build a shell around an existing general-purpose proof assistant, combined with a translation mechanism to and from its object language.

Currently, several good general-purpose proof assistants are available in practice, such as for instance PVS [OSRS01], COQ [The06] and ISABELLE [Pau07]. These proof assistants all have a large user base and make use of well-developed formal frameworks that are extremely expressive and powerful. In the shell approach, such a well-established formal framework is re-used automatically, which is a major advantage.

Unfortunately, general-purpose proof assistants have a major disadvantage as well: none have an object language that fully supports the semantics of CLEAN, which is based on lazy graph-evaluation with explicit strictness. Therefore, the evaluation mechanism of the proof assistant cannot be re-used, and an interpreter for CLEAN has to be built completely within the object language of the general-purpose proof assistant. This has the following important drawback:

> actual reasoning no longer takes place on the level of the CLEAN program, but instead on a meta-representation of it in the object language of the general-purpose proof assistant

From the programmer’s point of view, however, it is crucial that reasoning at least appears to be taking place on the level of the CLEAN program. In the case that a general-purpose proof assistant is used, it is therefore the task of the shell to hide the underlying meta-level completely from the end user. Consequently, applying a reasoning step in a shell actually requires three activities: (1) translate the program and the reasoning step to the meta-level; (2) execute the reasoning step on the meta-level; (3) translate the feedback back to the programming level.

To summarize, the shell approach has the advantage that a well-established formal framework is re-used, but the disadvantage that an interpreter and a two-way translation and communication mechanism have to be realized. We feel that the general-purpose approach poses more practical problems than it offers advantages; therefore, we have chosen to make use of the dedicated approach.

In hindsight, SPARKLE has been the result of only about 18 ‘man-months’ of work, which shows that writing one’s own dedicated proof assistant is certainly doable. We estimate that writing a shell would have taken considerably more effort. On the other hand, the formal framework of SPARKLE does lack some expressiveness, but this has turned out to be only a slight disadvantage for reasoning about functional programs.
4.3.4 **Sparkle’s approximation of dedicated reasoning**

SPARKLE is a dedicated proof assistant and aims to support formal reasoning on the level of the programming language itself. For this purpose it allows reasoning on the level of CLEAN, but with the following restrictions:

- all uniqueness annotations are removed automatically from the program;
- I/O-operations have no semantic model and are meaningless;
- overflow and rounding is disregarded;
- programs are syntactically simplified to an essential subset before reasoning.

Due to the first restriction, it is not possible in SPARKLE to specify properties that make use of uniqueness. Programs with uniqueness, however, can still be loaded: the uniqueness check is first performed as usual, and then the uniqueness annotations are simply removed. Due to the second restriction, it is not possible to use SPARKLE for proving properties of I/O. Due to the third restriction, many laws about numbers (such as for instance $\forall n. n < n+1$) hold in SPARKLE, but are falsified by programs in which overflow/rounding occurs. Adding user-friendly support for uniqueness, I/O, overflow and rounding is still future work.

The fourth restriction differs from the first three. Firstly, it does not restrict the scope of reasoning, because it allows all programs to be simplified without loss of meaning. Secondly, it always has an influence on reasoning, because every program is simplified implicitly. Thirdly, it is almost impossible to avoid, because defining reasoning support (both on the theoretical and on the practical level) for all of the many syntactic constructs of CLEAN is practically undoable.

The simplification of programs is performed automatically by SPARKLE for all programs that are loaded. The target of the simplification is Core-CLEAN, which is the intermediate language of the CLEAN compiler. From the user’s point of view, it seems that SPARKLE operates on the level of CLEAN, but reasoning actually takes place on the level of Core-CLEAN. Still, the level of Core-CLEAN approximates dedicated reasoning very well, because:

- **Core-CLEAN has the same expressive power as CLEAN.**
  
  Without loss of meaning, any CLEAN program can be transformed to an equivalent Core program, on which reasoning with SPARKLE is possible. Furthermore, the transformation itself has already been implemented in the actual CLEAN compiler. Because both SPARKLE and the compiler are written in CLEAN, the existing transformation can be re-used. This not only saves a lot of time, but also ensures soundness of the transformation.

- **Core-CLEAN is a subset of CLEAN.**
  
  Programs written in Core-CLEAN can easily be understood by CLEAN programmers, because they make use of the syntax and semantics of CLEAN. Understanding the program to reason about is vital for the success of formal reasoning.
• Programs in CORE-CLEAN are very similar to their CLEAN originals.

The changes between the CORE-CLEAN program and the CLEAN original are mainly syntactical in nature, and can in many cases even be hidden by SPARKLE. Moreover, the structure of the program is unchanged. As a result, much of the programmer’s expertise of the source program is still valid on the CORE-CLEAN level. Again, this increases the understanding of the program to reason about.

Of the four restrictions, the lack of support for dealing with I/O is the most significant, as I/O is an important component of many programs and one would like to reason about it. On the other hand, the usefulness of properties that make use of uniqueness still has to be established, and rounding and overflow are not an issue for the majority of programs. Furthermore, CORE-CLEAN is a suitable intermediate reasoning level.

The differences between CORE-CLEAN and CLEAN, as well as the feature of SPARKLE to present CORE programs as if they were CLEAN programs, will be explained further in the Tutorial in Section 4.4.1.

4.4 Tutorial part I: getting started with SPARKLE

In the following sections, a step-by-step introduction of the basic functionality of SPARKLE will be presented. The introduction covers the user interface, the specification of programs and properties, the semantics, and the three different supported styles of reasoning. At various places assignments are included, with the purpose of giving the reader the opportunity to gain hands-on experience with the SPARKLE proof assistant.

The tutorial will be continued in Section 4.5, in which the specialized features of SPARKLE will be described. A summary of all available reasoning steps is given in Appendix A.

In order to carry out the assignments in the tutorial, the standard CLEAN 2.2 distribution (available at http://clean.cs.ru.nl) must be installed, and the files from http://www.cs.ru.nl/~marko/research/sparkle/SparkleCEFP2007.zip must be merged in it. This additional package contains both a full version of SPARKLE, and the used example programs undefined and primes (which will be placed in the Examples\CEFP folder of the CLEAN distribution). Note that SPARKLE is available for Windows only. The answers to the assignments are available at http://www.cs.ru.nl/~marko/research/sparkle/cefp2007/.

4.4.1 Loading a program

The first step of formal reasoning with SPARKLE is loading a CLEAN program into its memory. This program provides the context information that is required for stating and proving properties. The fastest way of starting SPARKLE and loading a program is by means of the standard IDE of CLEAN, in which access
to SPARKLE has been integrated:

**Assignment 1:** (loading a program into SPARKLE automatically)
(a) Open the CLEAN project primes.prj in the Examples\CEFP folder.
(b) Examine the code of the main module (primes.icl) and attempt to predict the behavior of the program. Then, compile and run the program.
(c) Find the Theorem Prover Project option and use it to launch SPARKLE.

Alternatively, programs (and individual modules) can also be loaded from within SPARKLE, either by opening entire projects (Ctrl-O), or by opening the standard environment only (Ctrl-E), or by adding single modules (Ctrl-+).

Internally, SPARKLE maintains its own representation of the program. In this representation, a program is simply considered to be a list of (interdependent) modules, and each module is considered to be a list of definitions. SPARKLE does not distinguish between the definition (.dcl) and implementation (.icl) parts of a module and allows access to all components of a program at any time.

\[
\begin{align*}
\text{Program} & := \text{Module}^* \\
\text{Module} & := \text{Definition}^* \\
\text{Definition} & := \text{Algebraic Type} | \text{Record Type} | \text{Function} | \text{Class} | \text{Instance}
\end{align*}
\]

SPARKLE has a powerful graphical user interface that allows the structure of the loaded program to be inspected in detail:

**Assignment 2:** (browsing through the program structure)
(a) Find the window that displays the list of modules that are currently loaded. In this list, find the primes module and open it.
(b) The opened window actually filters all available definitions with the formula ‘functions from the primes module’. Change the filter to find all functions in StdList and StdFunc that begin with the letter ‘s’.

The user interface also allows each individual definition of the loaded program to be displayed in a separate window. Furthermore, these definition windows are interconnected by means of the symbols that are used within it:

**Assignment 3:** (browsing through the program components)
(a) Open the definition of the function isPrime in the primes module.
(b) Follow the internal link to the canBeDividedByAny function.
(c) Follow the internal link to the predefined rem function.

SPARKLE is a dedicated proof assistant that aims to support reasoning on the level of the programming language. Unfortunately, reasoning on the level of CLEAN is not practical, because of the many different syntactical constructs that are allowed. Therefore SPARKLE uses CORE-CLEAN, which is basically the subset of CLEAN in which all syntactic sugar has been removed, as intermediate reasoning language. The only remaining definitions in CORE-CLEAN are algebraic types and global functions, and expressions may only be constructed by means of applications, case distinction and lets.
Even though Core-Clean is a small language only, all Clean programs can be represented in it. When a Clean program is loaded into Sparkle, it is always automatically converted to Core-Clean. As a result, the program in the memory of Sparkle differs from the original Clean version. Some important differences between the Clean program and its Core-Clean equivalent are:

- all local functions have been lifted to the global level;
- all pattern matches have been transformed to case distinctions;
- all sharing has been expressed by means of recursive lets;
- all overloading has been expressed by means of dictionaries;
- all synonym types and macro’s have been expanded fully;
- all list comprehensions and dot-dot-expressions have been transformed to function applications.

Fortunately, the differences between the internally loaded Core-Clean program and the original Clean version only have a slight effect on reasoning, and are therefore hardly noticeable most of the time. Furthermore, the user interface of Sparkle is able to optionally display parts of Core-Clean programs in the syntax of their original Clean versions:

**Assignment 4:** *(effect of the optional display options)*

(a) Open the function definitions `isPrime` and `canBeDividedByAny` from the `primes` module and `span` from the `StdList` module.

(b) Toggle the display options **Pattern Matching** and **Case/Let vs #/!**. The ‘real’ Core-Clean program is displayed when the options are toggled off.

(c) There is one difference between the internal version of `isPrime` and the Clean version that cannot be hidden. What is this difference?

### 4.4.2 Undefinedness in Clean and Core-Clean

As in any other programming language, computations in Core-Clean and in Clean can terminate erroneously. This can happen in a number of situations, for example when dividing by zero, or when a partial function is applied to an argument for which it was not intended. Additionally, Clean even offers two standard functions that always terminate erroneously, namely `abort` and `undef`.

One of the features of lazy languages is that it is possible for a computation to produce a (partial) end result, even when it contains subcomputations that terminate erroneously. This is only possible, however, when the subcomputation is not needed for producing the end result at all.

**Assignment 5:** *(partial undefinedness in practice)*

(a) Open the `undefined` project with the IDE. Run and compile it.
(b) Replace the body of myundefined with another computation that also terminates erroneously.
(c) Cycle through the available Start bodies and examine the run-time results.

A formal model of CLEAN needs to be able to handle expressions that contain undefined subexpressions. For this purpose, CORE-CLEAN defines the additional expression alternative ‘⊥’. This constant expression is treated as a base value of any type, because a computation of any type can terminate erroneously. All different kinds of errors are treated equally; therefore, only one ⊥ suffices and it does not need additional arguments.

Note that ⊥ is a special value with special characteristics. It cannot be used as a pattern, or in a case distinction. In fact, it is not possible at all in CLEAN to produce a defined result based on a successful check of undefinedness.

**Assignment 6: (undefinedness cannot be detected)**

(a) What famous (unsolvable) problem would be solved if it was possible to detect undefinedness within a CLEAN program?

### 4.4.3 Stating a property

A property in SPARKLE is a logical statement, either true or false, that deals with the executonal behavior of a CLEAN program. Properties can be used to state that the program functions correctly with respect to its specification. Expressing the desired behavior of a program by means of properties is very useful.

SPARKLE allows properties to be expressed in an extended first-order logic. The usual logic operators ¬ (not), ∧ (and), ∨ (or), → (implies) and ↔ (iff) are supported, as well as the quantors ∀ (for all) and ∃ (exists), and the constants TRUE and FALSE. Variables and quantors can range over propositions and over expressions of an arbitrary type, but not over predicates or relations of any kind. To state properties of programs, the logic also supports equality on expressions.

```
Prop ::= Var\text{prop} | \text{TRUE} | \text{FALSE} |
       \neg Prop | Prop \land Prop | Prop \lor Prop | Prop \rightarrow Prop | Prop \leftrightarrow Prop |
       \forall Var\text{prop}.Prop | \forall Var\text{expr}.Prop | \exists Var\text{prop}.Prop | \exists Var\text{expr}.Prop |
       Expr = Expr
```

Many concepts of the proposition level are also available on the expression level, which can be a little confusing. Note for instance the subtle differences between:

- **True** and **False**, which are expressions of type $\text{Bool}$, and **TRUE** and **FALSE**, which are propositions;

- **not, &&** and $||$, which are CLEAN functions that operate on values of type $\text{Bool}$, and $\neg$, $\land$ and $\lor$, which are operators that connect propositions;
• ==, which is an overloaded CLEAN function that produces a Bool and must be defined manually for each type, and =, which produces a proposition and is available automatically for each type.

(the CLEAN function == is computable and cannot compare undefined values, while the formal = is not computable and can compare undefined values; this additional expressiveness is really important, because many properties have definedness preconditions that could otherwise not be expressed)

On the other hand, the availability of the expression level also allows boolean expressions to sometimes be used as predicates and relations (see Section 4.5.7).

Assuming the context of the primes project, examples of properties are:

1. \( \forall P \forall Q. (P \land Q) \leftrightarrow (Q \land P) \)
2. \( 17 > 12 = \text{True} \)
3. \( \forall f \forall xs \forall ys. \text{map } f (xs ++ ys) = \text{map } f xs ++ \text{map } f ys \)
4. \( \forall xs. \text{reverse } (\text{reverse } xs) = xs \)
5. \( \forall n \forall xs. (n < \text{length } xs = \text{True}) \to \text{length } (\text{take } n xs) = n \)
6. \( \forall i \forall j. (i > j = \text{True} \land j > 0 = \text{True}) \to \text{primes !!! } i > \text{primes !!! } j = \text{True} \)

Of these properties, the first does not refer to any component of the program; in fact, it is a tautology which is independent of any program. The second property refers to the function >, which is defined for integers in the module StdInt. The third, fourth and fifth properties refer to the functions map, ++, reverse, take and length, which are all defined in the module StdList. The sixth property, finally, is the only property that is really specific for the primes project. It not only depends on the standard functions > and !!!, but also on the primes function of the primes module.

Assignment 7: (validity of the example properties)
(a) Of the six example properties, only five are true, and one is in fact false (it needs an additional precondition). Which one is false?
(Hint: lists may be infinite in CLEAN)
(b) What happens to the sixth property if either \( i \) or \( j \) is undefined?

The only way to enter properties in Sparkle is by means of textual input. The parser allows the natural syntax to be used, with the following conventions:

- \( \neg P \) denotes \( \neg P \);
- \( P \land Q \) denotes \( P \land Q \);
- \( P \lor Q \) denotes \( P \lor Q \);
- \( P \to Q \) denotes \( P \to Q \);
Section 4.4.3: Stating a property

- \( P \leftrightarrow Q \) denotes \( P \leftrightarrow Q \);
- \( \bot \) denotes \( \bot \);
- \( [x] \) denotes \( \forall x \); and
- \( \{x\} \) denotes \( \exists x \).

Type-checking of propositions is performed automatically by Sparkle. During this check, the types of the variables are inferred as well. Alternatively, it is also possible to explicitly specify the type of a variable in a quantor. These explicit types may contain type variables, which are implicitly assumed to be bound by universal quantors. Typed quantors are denoted by:

- \( [x::a] \) denotes \( \forall x :: a \); and
- \( \{x::a\} \) denotes \( \exists x :: a \).

Assignment 8: (specify the example properties (1))
(a) Use New Theorem to manually enter all six example properties.

(Hint: in case of failure, attempt to add brackets)

Assignment 9: (specify properties with overloading)
The manual specification of types is essential when making use of overloading:
(a) Without explicit types, attempt to specify \( \forall x \forall y. x + y = y + x \).
(b) Use explicit types \( (x::\text{Int}, y::\text{Int}) \) to help Sparkle solve the overloading in \( \forall x \forall y. x + y = y + x \).

For the sake of convenience, Sparkle offers two features to make the manual specification of properties easier:

- Each free symbol in the proposition is assumed to be a variable, and a universal quantor is created automatically for it. This feature allows universal quantors to be omitted when specifying properties. It also means, however, that mistyping the name of an identifier, or using an identifier that is not defined by the current program, does not lead to a bind error, but instead results in an incorrect universal quantor.

- When possible, boolean expressions are automatically lifted to propositions by implicitly adding ‘= True’. This feature shortens specifications, but may also lead to confusion between the expression and the proposition level. Note that the ‘= True’ behind a lifted boolean expression is not even displayed by Sparkle if the Boolean Predicates display option is turned on.

Assignment 10: (specify the example properties (2))
(a) Specify the example properties again, using the features described above. Do not quit Sparkle afterwards.
Sparkle organizes theorems and proofs into sections, much in the same way as Clean organizes definitions into modules. Sections are stored in a semi-readable internal format in Sparkle’s \Sections subdirectory. Theorems and (parts of) proofs can be assigned to individual sections, which must then be saved explicitly. The special section main is always available, but it cannot be saved and should only be used for temporary properties. A warning for users: Sparkle does not save sections automatically, and does not prompt you to do so either!

Assignment 11: (save properties into sections)
(a) Create a new section with the name temp.
(b) Open both the main section and the temp section.
(c) Move the example properties from the main section into the temp section.
(d) Save the temp section and quit Sparkle.

Of course, sections can be loaded into Sparkle as well. Because the contents of a section may depend on various other components, the following actions are carried out when a section is loaded:

- First, it is verified if the symbols are available that are required for stating the properties of the section. If this is not the case, then the section is not loaded at all. Otherwise, theorems are created for the properties of the section. The proofs themselves, however, are not loaded yet.
- Then, the sections are loaded recursively that contain the theorems that are used within the proofs of the top-level section.
- Finally, the proofs of the section are loaded and carried out again, step by step. If a step fails, which may be the case if a definition within the program has been altered (but its name and type were unchanged), then the proof can be loaded partially until the error point.

After this process, it can be guaranteed that the internal state of Sparkle is consistent, and that all proofs that were loaded successfully are valid.

Assignment 12: (load sections into memory)
(a) Start Sparkle manually (directly and not from within the IDE).
(b) Attempt to load the predefined section lists.
(c) Use Ctrl-O to open the primes project from within Sparkle.
(d) Load the predefined section lists.
(e) Load the section temp of the previous assignment.

4.4.4 The meaning of properties
The meaning of properties is described by a formal algorithm that determines whether a given property, in the context of a given program, is true or false.
Section 4.4.4: The meaning of properties

This algorithm is expressed at the formal level only, and cannot be executed in practice, neither by a human nor by a computer. If it could be executed, formal reasoning would not have been necessary in the first place.

A meaning must be provided for all alternatives of Sparkle’s first-order logic, which was introduced in Section 4.4.3. This logic contains both standard elements (TRUE, FALSE, ¬, ∧, ∨, →, ↔, ∀ on propositions, ∃ on propositions) and customized ones (=, ∀ on expressions, ∃ on expressions). The meaning of the standard elements is the same as in standard logic, which we assume to be well-known. The meaning of the customized elements is as follows:

- The equality \( e_1 = e_2 \) holds if for all programs \( P \) the observational behavior stays the same if \( e_1 \) is interchanged with \( e_2 \) (or vice versa, \( e_2 \) with \( e_1 \)). The observational behavior of a program is the visible output that is produced when it is executed. Sparkle cannot deal with programs that perform I/O; therefore, only output that is displayed on the console is considered. To be able to determine the equality between observational behaviors, it has to be taken into account that programs may not terminate, and that the output that they produce may be infinite. On the formal level, observational behavior is therefore modeled by time indexed streams, and bisimulation is used to determine equality. On the intuitive level, this is equivalent to assuming that infinite time is available to programs, and that the resulting infinite streams are equal only if all their finite substreams are equal.

  Finally, note that it is not possible to determine if \( e_1 \) and \( e_2 \) are semantically equal based only on the observational behaviors of the programs \( \text{Start} = e_1 \) and \( \text{Start} = e_2 \). This is because \( e_1 \) and \( e_2 \) may be functions that only produce meaningful output when they are supplied with arguments.

- The universal quantification \( \forall x. P \) holds if for all wellformed expressions \( E \) the instantiated proposition \( P[x \mapsto E] \) holds. An expression \( E \) is well-formed if the resulting \( P[x \mapsto E] \) is both closed and welltyped.

  Note that the undefined expression \( \bot \) is always a valid value for \( E \), because it is closed and of any type. Furthermore, if the domain of \( x \) allows for it, infinite expressions are also valid values for \( E \).

- The meaning of the existential quantification \( \exists x. P \) is defined in the same way as the universal quantification.

Assignment 13: (examples of (in)equality)

(a) Are ‘ones’ and ‘\text{let } x = [1:x] \text{ in } x’ equal? If so, argue; if not, give the program that distinguishes between them.

(b) Same question for ‘ones’ and ‘ones ++ ones’.

(c) Same question for ‘ones’ and ‘[2] ++ ones’.

(d) Same question for ‘ones’ and ‘ones ++ [2]’.

(e) Same question for ‘\bot’ and ‘[1:\bot]’.
4.4.5 Reasoning style in Sparkle

As most modern day proof assistants, Sparkle is based on the LCF-approach. This means that reasoning takes place by the repeated simplification of a list of goals by means of the application of tactics. This process of reasoning was first introduced by the LCF\[GMW79\] proof assistant, and has since been named after it.

The theoretical background of this style of reasoning was already introduced in Sections 4.2.3 and 4.2.4. From the user’s point of view, each theorem requires the repeated manipulation of a list of goals (=properties to be proved) by means of the application of tactics (=reasoning steps). The goals can be proved in any order; the goal currently being manipulated is called the active goal and the others are called subgoals. The tactics must be selected from a fixed library, and are guaranteed to be sound. The formal proof tree is maintained internally by Sparkle and can be browsed manually for an overview of the proof, but it is otherwise not needed for reasoning at all.

Assignment 14: (backwards proving)
(a) Why is Sparkle’s reasoning style sometimes also called backwards proving?

A goal corresponds to a property that still to be proved, but on the syntactic level it is broken into components which can be manipulated separately by the reasoning process. The components of a goal are introduced variables, introduced hypotheses and the ‘to prove’. If \(x_1, \ldots, x_n\) are the introduced variables, \(H_1 : P_1, \ldots, H_m : P_m\) are the introduced hypotheses, and \(Q\) is the to prove, then the goal corresponds to the property \(\forall x_1, \ldots, x_n. P_1 \rightarrow \ldots P_m \rightarrow Q\).

Assignment 15: (decompose the property)
The proof states in Fig. 4.1 and Fig. 4.2 are taken from an actual proof.
(a) Which property corresponds to the current goal in Fig. 4.1?
(b) Which property was the starting point of the proof?

4.4.6 Proving a simple property

In this section, we will use Sparkle to prove a simple property which concerns the behavior of the map function from the standard environment of Clean.

Assignment 16: (specification of a property of map)
(a) Open Sparkle from scratch, then load the standard environment (Ctrl-E).
(b) Create a new section with the name map_section.
Section 4.4.6: Proving a simple property

(c) In map section, create a new theorem named \texttt{map\_property}, stating:
\[
\forall f \forall xs \forall ys. \texttt{map} f (xs ++ ys) = \texttt{map} f xs ++ \texttt{map} f ys
\]

(d) Open the proof window (Ctrl-P) that corresponds to the created theorem.

Building a proof is the repeated process of selecting tactics and applying them on the current goal. For this process, \textsc{Sparkle} makes a total of 39 tactics available, which are all described briefly in Appendix A. The user interface of \textsc{Sparkle} allows tactics to be applied by means of three different methods:

- The \textit{hint mechanism}, which is activated by opening the Tactic Suggestion Window during proving. This window holds a dynamically updated list of suggestions for tactics that can be applied to the current goal. \textsc{Sparkle} generates these suggestions automatically based on built-in heuristics. Each suggestion is assigned a score between 1 and 100 that indicates the likelihood of that tactic being helpful in the proof. Based on this score, the suggestions are ordered. A suggested tactic can be applied by either clicking on it, or by means of its associated hot-key (F1 for the first hint, F2 for the second, etc.). It is also possible to configure \textsc{Sparkle} to apply the top hint automatically if it has a score higher than a manually set threshold.

The hint mechanism is mainly for \textit{beginning} \textsc{Sparkle} users. It is fast and easy to use, and requires little expertise of the available tactics (simply trust \textsc{Sparkle}!). The hint mechanism is a valuable tool that can be used...
as a means of learning SPARKLE, and with which many small proofs can be built fully. However, it is not very powerful and by no means failsafe. Sometimes the right tactic is not suggested, or several wrong tactics get high scores.

- The tactic dialogs. Each tactic has its own dialog that can be opened by clicking on its name in the Tactic List Window. This dialog has entries for all the arguments that can be given to the tactic. When possible, the current goal is used to restrict the input to valid values only. When all arguments have been entered, the tactic can be applied from the dialog directly.

  The tactic dialogs are for intermediate users. This method of proving is both powerful, because all tactics can be applied this way, and fairly easy, because one does not need to memorize the name or syntax of a tactic, nor the arguments that it requires.

- The command line interface. This is a textual interface, located at the bottom of the Proof Window, that is for advanced users only. It is powerful, but requires extensive expertise of SPARKLE and its tactics. However, once mastered, it is the fastest way of building proofs, because all tactics can be applied this way and it does not require opening additional dialogs at all.

The property of map that was given above is very easy and can therefore be proved automatically with the hint mechanism:

**Assignment 17:** (proving the map property with the hint mechanism)

(a) Open the Tactic Suggestions Window (Ctrl-H) and set the threshold to 1.

(b) Set the threshold back to 101. Why is this necessary prior to (c)?
Section 4.4.6: Proving a simple property

(c) Enter ◀Restart► at the command-line interface.
(From now on, ◀cmd► will be used to denote textual input to the command-
line. For reasons of parsing, these commands have to end with a closing ‘.’, otherwise SPARKLE will not be able to recognize them.)

(d) Redo the proof by applying suggestions manually with the hot-keys.

The complete proof tree of the example property has now been stored internally by SPARKLE. By means of the Theorem Info Window, this proof tree can be browsed and inspected in detail:

Assignment 18: (browsing through the proof)
(a) Open the Theorem Info Window of the completed proof.
(b) Click ‘browse’ after the first tactic and then browse through the proof using the ‘previous’ and ‘next’ buttons.
(c) Undo the first application of Reflexive only.
(d) Click on the brown star to return to the Proof Window.
(e) Use a different tactic to prove the goal.

The hint mechanism has succeeded in completing the proof automatically, and it did not require any expertise at all. The downside to this, unfortunately, is that no understanding of the tactics has been gained in the process. Therefore, below we will present the entire proof again, and this time we will explain each tactic that was applied too.

The initial goal is simply the property to be proved:

\[
\forall f \forall xs \forall ys. \text{map } f (xs ++ ys) = \text{map } f xs ++ \text{map } f ys
\]  

Because both map and ++ are tail-recursive, structural induction on xs is likely to be useful here. This is accomplished by applying the tactic ◀Induction xs►. Three new goals(1.1,1.2,1.3) are created: one for the case that xs is ⊥; one for the case that xs is Nil; and one for the case that xs is an application of Cons. Note that ⊥ is a base value of any type and is therefore always treated by induction as a constructor case.

\[
\forall f \forall ys. \text{map } f (\bot ++ ys) = \text{map } f \bot ++ \text{map } f ys
\]  

The current proposition starts with two universal quantifications, on which it does not make sense to perform induction (on f it is not possible, and on ys it does not help because ++ is not tail-recursive in its second argument). It is therefore best to apply ◀Introduce f ys►, which removes the quantors and introduces the variables f and xs in the context of the goal. After this action, the main proposition can be accessed more easily.

\[
f :: b \rightarrow a, \ ys :: [b]\\n\text{map } f (\bot ++ ys) = \text{map } f \bot ++ \text{map } f ys
\]  

(1.1')
Due to the strictness of \texttt{map} and \texttt{++} and the presence of \(\bot\) arguments, redexes are present in the current goal. The tactic \texttt{\texttt{\texttt{Reduce NF All.}}} can be used to reduce all redexes in the current goal to normal form. With other parameters, the tactic \texttt{\texttt{Reduce}} can also be used for stepwise reduction, reduction to root normal form, reduction of one particular redex and reduction in the goal context.

\[
\frac{\text{\(f : b \rightarrow a, \ ys : [b]\)}}{\bot = \bot} \tag{1.1''}
\]

This is clearly a trivial goal, because equality is a reflexive relation. Such reflexive equalities are proved immediately with the final tactic \texttt{\texttt{\texttt{Reflexive.}}}.

\[
\forall f \forall ys. \ \text{map} \ f \ (\emptyset \ ++ \ ys) = \text{map} \ f \ \emptyset \ ++ \text{map} \ f \ ys \tag{1.2}
\]

This is the second goal of induction, created for the case that \(xs\) is the empty list. Again, induction makes no sense for \(f\) and \(ys\), and they should therefore be introduced in the goal context by means of \texttt{\texttt{\texttt{Introduce f ys.}}}.

\[
\frac{\text{\(f : b \rightarrow a, \ ys : [b]\)}}{\text{map} \ f \ (\emptyset \ ++ \ ys) = \text{map} \ f \ \emptyset \ ++ \text{map} \ f \ ys} \tag{1.2'}
\]
Section 4.4.6: Proving a simple property

There are again redexes present in the current goal, because both \( \text{map} \) and ++ have patterns that match on the empty list \([\ ]\). Therefore: ◀Reduce NF All.▶.

\[
\begin{array}{c}
  f :: b \rightarrow a, \ \ y :: \ [b] \\
  [\ ] = [\ ] \\
\end{array}
\]  

(1.2'')

This is another example of a reflexive equality; therefore ◀Reflexive.▶.

\[
\begin{array}{c}
\forall x \forall xs.
\text{ (\forall yys. map } f (xs ++ ys) = map f xs ++ map f ys) \\
\rightarrow (\forall yys. map f ([x:xs] ++ ys) = map f [x:xs] ++ map f ys)
\end{array}
\]  

(1.3)

This is the third goal created by induction for the case that \( xs \) is a composed list. The current goal looks quite complicated, but introduction can make things a lot clearer. Here, we will not only introduce variables from universal quantors, but we will also introduce hypotheses from implications. This can be performed in one go with ◀Introduce x xs IH f ys.▶.

\[
\begin{array}{c}
x :: b, \ xs :: [b], \ f :: b \rightarrow a, \ y :: [b] \\
IH : \forall yys. \text{map } f (xs ++ ys) = \text{map } f xs ++ \text{map } f ys \\
\text{map } f ([x:xs] ++ ys) = \text{map } f [x:xs] ++ \text{map } f ys
\end{array}
\]  

(1.3')

Again, the current goal contains redexes, because \( \text{map} \) and ++ have patterns that match on constructed lists of the form \([x:xs]\). Therefore, ◀Reduce NF All.▶.

\[
\begin{array}{c}
x :: b, \ xs :: [b], \ f :: b \rightarrow a, \ y :: [b] \\
IH : \forall yys. \text{map } f (xs ++ ys) = \text{map } f xs ++ \text{map } f ys \\
[f x : \text{map } f (xs ++ ys)] = [f x : \text{map } f xs ++ \text{map } f ys]
\end{array}
\]  

(1.3'')

The current proposition is now of the form \([X:Y] = [X:Z]\). Using the automatic injectivity of all lazy data constructors in CLEAN, we can simplify this to \( X = X \land Y = Z \). Therefore, ◀Injective.▶.

Assignment 19: (injectivity and strictness)

(a) Why does injectivity not hold for strict data constructors?

\[
\begin{array}{c}
x :: b, \ xs :: [b], \ f :: b \rightarrow a, \ y :: [b] \\
IH : \forall yys. \text{map } f (xs ++ ys) = \text{map } f xs ++ \text{map } f ys \\
f x = f x \land \text{map } f (xs ++ ys) = \text{map } f xs ++ \text{map } f ys
\end{array}
\]  

(1.3''')

The current proposition is now of the form \( P \land Q, \) and can obviously be split into subgoals P and Q. Therefore, ◀Split.▶, which creates subgoals 1.3.1 and 1.3.2.

\[
\begin{array}{c}
x :: b, \ xs :: [b], \ f :: b \rightarrow a, \ y :: [b] \\
IH : \forall yys. \text{map } f (xs ++ ys) = \text{map } f xs ++ \text{map } f ys \\
f x = f x
\end{array}
\]  

(1.3.1)
Chapter 4: Proving properties of lazy functional programs with SPARKLE

This is a reflexive equality that can be proved immediately with \texttt{\textbackslash Reflexive.}.

\[
\begin{align*}
\forall x :: b, \forall xs :: [b], \forall f :: b \rightarrow a, \forall ys :: [b], \forall \text{IH}.
\forall f \forall ys. \map f (xs \++ ys) &= \map f xs \++ \map f ys \\
\map f (xs \++ ys) &= \map f xs \++ \map f ys 
\end{align*}
\]

The current proposition is now an instantiation of the induction hypothesis \texttt{IH}.
It can therefore be proved immediately by applying \texttt{IH} with \texttt{\textbackslash Apply IH.}.

Q.E.D.

There are no more subgoals, which means that the proof is complete!

Assignment 20: \textit{(manual proof of the map property)}

(a) Prove the \texttt{map} property again, using the tactic dialogs only.
(b) Prove the \texttt{map} property again, using the command interface only.
   \textit{(Hint: \texttt{\textbackslash Reduce.} abbreviates \texttt{\textbackslash Reduce NF All.}, and \texttt{\textbackslash Intros.} is a variant of introduction that comes up with suitable names on its own)}
(c) The automatic proof consists of the application of 13 tactics. It is possible to prove the property in less steps (our shortest proof consists of 9 steps).
   Try to shorten the proof yourself.

Assignment 21: \textit{(more small proofs)}

Try to prove the following properties, preferably without the hint mechanism:
(a) \(\forall xs \forall ys \forall zs. (xs \++ ys) \++ zs = (xs \++ ys) \++ zs\).
(b) \(\forall n \forall xs. (n = \bot) \rightarrow (xs = []) \rightarrow [\text{hd}\, xs; \text{tl}\, xs] = xs\).
(c) \(\forall n \forall zs. (n = \bot) \rightarrow \text{take}\, n\, xs \++ \text{drop}\, n\, xs = xs\).
(d) \(\forall P \forall Q. (\neg P \leftrightarrow Q) \leftrightarrow (P \leftrightarrow \neg Q)\).

4.5 Tutorial part II: specialized SPARKLE-features

In this section, the tutorial will be continued with advanced information about the dedicated use of SPARKLE in practice, and the features that are specialized for reasoning about CLEAN will be described. The same explanatory style will be used as in part I of the tutorial, and various assignments will again be included.

First, in Section 4.5.1 the importance of sharing in proofs will be explained. Then, the specification of definedness conditions in properties will be described in Section 4.5.2. The specialized behavior of four tactics will be introduced next; for ‘Extensionality’ in Section 4.5.3, for ‘Induction’ in Section 4.5.4, for ‘Definedness’ in Section 4.5.5, and for ‘Reduce’ in Section 4.5.6. Finally, the specification of properties by means of CLEAN functions will be discussed in Section 4.5.7.

4.5.1 The influence of sharing on reasoning

Sharing is important for the efficiency of functional programs. In CLEAN sharing is explicit, because for every construct it is precisely defined what is shared.
and what is not shared[Pv01]. The semantics of Clean are based on graph rewriting [BvG’87a, Pv93, BS99]. This means that during reduction of the Start expression to its result, sharing is maintained as much as possible.

In Sparkle, reduction may be used at many points in proofs as well. This reduction should behave in a semantically equivalent way to reduction in Clean, but it does not have to be exactly the same. Note that reduction in Sparkle is symbolic, because it may encounter free variables that are introduced by logic quantors. In Clean, reduction only operates on closed expressions.

Sharing has no influence on semantics, and reduction in Sparkle is free to either preserve or break it. Currently, the following strategy is realized:

- Within the application of reduction sharing is always preserved;
- But afterwards sharing is always automatically broken.

The idea behind this strategy is twofold. Firstly, efficiency is important in proofs too, therefore sharing is preserved within reduction. Secondly, after full reduction sharing is often not meaningful anymore and only hinders reduction, therefore it is automatically broken.

Assignment 22: (the effect of sharing during reduction in proofs)
(a) Consider in Sparkle the trivial theorem (let $n = 1 + 2 + 3$ in $n + n$) = 12.
   Prove it using $\texttt{Reduce NF All.}$, followed by $\texttt{Reflexive.}$.
(b) Undo the proof with Ctrl-Z and prove the theorem again, this time using reduction with a fixed number of steps ($\texttt{Reduce 4.}$).
(c) Undo the proof with Ctrl-Z and prove the theorem again, this time using repeated single-step reduction ($\texttt{Reduce 1.}$).
(d) Explain why more reduction steps are needed in (c) than in (b).

Unfortunately, Sparkle’s current strategy for handling sharing is not optimal. The main problem is that all meaningful sharing, such as for instance recursion that has been expressed by means of cyclic lets, cannot be dealt with at all. Moreover, the current behavior is not very intuitive, as was already demonstrated in the assignment above.

The way sharing is handled in Sparkle is currently being fixed according to the reduction mechanism described in [dvP08b]. In the next release, Sparkle will always preserve all sharing, and manual reasoning steps will be added that allow users to manipulate, and possibly break, shared expressions at will.

4.5.2 Definedness conditions in properties

Sparkle makes use of a total semantics in which undefinedness is taken into consideration explicitly. This has two consequences for the property language. Firstly, expressions are only equal if they either produce the same defined value, or both produce undefinedness. Secondly, the undefined value $\bot$ is a member of any type, and therefore a valid instantiation of any quantor.
In order to specify properties of CLEAN programs correctly, one therefore has to know precisely how they behave in case some of their input becomes undefined. This behavior is determined by the lazy rewriting semantics of CLEAN, of which a thorough understanding is required for formal reasoning. Below we present a small example to illustrate the propagation of $\bot$-values through expressions. For a full explanation of computation in CLEAN we refer to [Pv01] and [vd06].

Example. Consider the following definition of the well-known function take:

\[
\begin{align*}
| \text{take } n \text{ [] } & = \text{ [] } \\
| \text{take } n \text{ [x:xs] } & = \text{ if } (n>0) \text{ [x: take } (n-1) \text{ xs] [] }
\end{align*}
\]

In CLEAN, patterns are evaluated from top to bottom, and right-hand-sides are only evaluated when their pattern matches. Consequently:

- $\text{take } n \bot = \bot$ for all $n$, because the first pattern always causes $\bot$ to be matched against [], which fails;
- $\text{take } \bot \text{ [] } = \text{ [] }$, because the successful match of the first pattern does not require $\bot$ to be evaluated;
- $\text{take } \bot \text{ [x:xs] } = \bot$ for all $x$ and $xs$, because the second pattern matches, and its right-hand-side requires the computation of $\bot > 0$, which fails.

It is very important that the starting point of formal reasoning is a logically correct property. Therefore, the specification of properties must always involve an analysis of behavior in the undefined case. In some cases, the property turns out to hold automatically for the undefined value, and nothing has to be changed. In other cases, however, the property actually turns out to be false:

Example. Consider the following intuitively true property of drop and take:

\[
\forall n \forall xs. \text{take } n \text{ xs }++ \text{ drop } n \text{ xs } = \text{ xs }.
\]

This property is falsified by the case $n = \bot$, because then the left-hand-side may become undefined, while the right-hand-side remains $xs$:

- Assume $xs = [1]$. Then the left-hand-side reduces to $\bot$, as follows:
  \[
  \text{take } \bot \text{ [1] }++\text{ drop } \bot \text{ [1] } = \bot++\text{ drop } \bot \text{ [1] } = \bot.
  \]
  But the right-hand-side is $[1]$, which is defined.

Assignment 23: (more definedness analysis)

(a) The example property $\forall n \forall xs. \text{take } n \text{ xs }++ \text{ drop } n \text{ xs } = \text{ xs }$ is not falsified in the case that $xs = \bot \land n \neq \bot$. Argue why this is the case.

(Hint: distinguish between $n = 0$ and $n \neq 0$.)

(b) Is the property $\forall f \forall xs \forall ys. \text{map } f \text{ (xs }++\text{ ys)} = (\text{map } f \text{ xs}) ++ (\text{map } f \text{ ys})$ falsified in the undefined case? If so, give example values for $f$, $xs$ and $ys$ that break the property. If not, argue why.

(Hint: see also Section 4.4.6.)

If definedness analysis shows that a property is falsified by a set of variable values $V$, then it can be rectified simply by adding conditions that exclude $V$. These definedness conditions are often simple and of the form ‘$n \neq \bot$’, but they can also be more intricate (see Section 4.5.7).
Section 4.5.2: Definedness conditions in properties

Rectified example: The take-drop property can be corrected by means of:
\[
\forall n \forall xs. n \neq \bot \rightarrow \text{take } n \text{ xs} ++ \text{drop } n \text{ xs} = \text{xs}.
\]

Assignment 24: (proving the rectified take-drop example)
(a) In Sparkle, prove \(\forall n \forall xs. n \neq \bot \rightarrow \text{take } n \text{ xs} ++ \text{drop } n \text{ xs} = \text{xs}\).

Finally, note that Clean supports strictness annotations, with which the strict evaluation of certain expressions can be enforced explicitly. These annotations are often placed without much thought with the purpose of improving efficiency. However, strictness annotations change the definedness behavior of the program, and have an effect on properties and reasoning as well. In the context of formal reasoning, they should therefore only be used with care.

The precise effect of strictness annotations on properties is difficult to predict. Adding a strictness annotation can either: (1) not change a property at all; or (2) falsify a property, requiring additional definedness conditions to be formulated; or (3) allow existing definedness conditions to be removed. The third effect in particular is rather surprising.

Example of (1). Consider the following property:
\[
\forall xs \forall ys \forall zs. (xs ++ ys) ++ zs = xs ++ (ys ++ zs)
\]
This property holds for the standard definition of ++, which is strict in its first argument only. Adding strictness to the second argument does not affect the property, however; it remains valid in the strict case as well.

Example of (2). Consider the following property:
\[
\forall f,g \forall xs. \text{map } (f \circ g) \text{ xs} = \text{map } f (\text{map } g \text{ xs})
\]
This property is valid for lazy lists, but invalid for element-strict lists. Suppose \(xs = [12], g 12 = \bot\) and \(f (g 12) = 7\).
Then \(\text{map } (f \circ g) \text{ xs} = [7]\), both in the lazy and in the strict case. However, \(\text{map } f (\text{map } g \text{ xs}) = [7]\) in the lazy case, but \(\bot\) in the strict case. The property can be adapted to element-strict lists by explicitly enforcing that \(g\) produces a defined result for all elements \(x\) of \(xs\):
\[
\forall f,g, xs. (\forall x \in xs. g x \neq \bot) \rightarrow \text{map } (f \circ g) \text{ xs} = \text{map } f (\text{map } g \text{ xs}).
\]

Example of (3). Consider the following property:
\[
\forall xs. \text{finite } xs \rightarrow \text{reverse } (\text{reverse } xs) = xs
\]
This property is valid both for lazy lists and for spine-strict lists. The condition \(\text{finite } xs\), however, is satisfied automatically for spine-strict lists, because spine-strict lists can never be infinite. In the spine-strict case, the property can therefore safely be reformulated (or, rather, optimized) by removing the \(\text{finite } xs\) condition:
\[
\forall xs. \text{reverse } (\text{reverse } xs) = xs
\]
Note that without the condition, the property is invalid in the lazy case: just choose any infinite list for \(xs\).
4.5.3 Specialized behavior of extensionality

The property of extensionality, which states that two functions are equal iff they produce the same result for all possible arguments, is often considered to be universal. Unfortunately, there is a (rather obscure) example of two functions for which the property of extensionality does not hold unconditionally in the context of lazy evaluation:

\[
\begin{align*}
H &: a \to b \\
F &: (a \to b)
\end{align*}
\]

In the definitions above, \(H\) is a function of arity 1 that only reduces (to itself) when it is given an argument. \(F\) on the other hand is a function of arity 0 that always reduces to itself, regardless of whether it is applied or not. Obviously, \(F x = H x\) now holds for all \(x\), because they both reduce to themselves and are therefore both undefined.

Surprisingly, the property \(F = H\) does not hold, because \(H\) is defined (it is a partial function application, and is thus in head normal form), while the meaning of \(F\) is undefined. It is therefore not safe to replace \(H\) by \(F\) (nor \(F\) by \(H\)); such a replacement could namely change the termination behavior of the program.

Fortunately, the problem can be corrected by weakening the property of extensionality as follows:

**Definition 4.5.3: (revised version of extensionality)**

\[\forall f \forall g. (f = \bot \iff g = \bot) \rightarrow (\forall x. f x = g x) \rightarrow f = g\]

This revised version of extensionality is correct in the context of CLEAN. It cannot be applied to prove \(F = H\), because the condition \(F = \bot \iff H = \bot\) does not hold. SPARKLE defines a reasoning step for extensionality that makes use of the correct behavior.

**Assignment 25: (extensionality)**

(a) Prove using extensionality that \(\text{sum} \circ (\text{map} \, \text{const} \, 1) = \text{length}\) holds.

4.5.4 Specialized behavior of induction

An important reasoning step for dealing with recursive functions over algebraic datatypes is structural induction. Although induction is not always applicable, it is extremely useful in the context of functional programming, because it can be used successfully on many common data structures (such as for instance lists) and on many common kinds of recursive functions (such as for instance those defined by recursion on the results of pattern matching).

In order to deal with lazy evaluation, induction has to be customized in two different ways. Firstly, an extra base step is required for the undefined value \(\bot\). Because \(\bot\) is a member of each type, it must namely be treated as a constructor with no arguments. This behavior of induction is actually quite intuitive; for
instance, if we want to prove $\forall x \in A. P(x)$ with induction on the list structure, we would get the following proof obligations:

- $P(\bot)$;
- $P(\[])$;
- $\forall x \in A \forall xs \in A. P(xs) \rightarrow P([x : xs])$

Note that without the case for undefinedness it is possible to prove properties that are not true. For instance, we could easily prove that every lazy list is finite: the empty list is finite, and the extension of a finite list with a single element is always finite as well. The undefined list, on the other hand, is not finite!

The second customization of induction extends it to infinite structures as well. Because an infinite structure does not end with a base case, the induction principle is in general not applicable to it. In [Pau87], however, Paulson has shown that the results of induction may be applied to infinite structures as long as the induction predicate satisfies the criterion of admissibility. We claim that Paulson’s results may be applied to the context of Clean as well.

The admissibility criterion can be lifted to lazy functional languages easily. The basic idea is that equalities on negative positions (behind a negation) within a proposition must be decidable. An equality on type $\alpha$ is decidable if all possible expressions of type $\alpha$ are finite. This can be approximated statically: if $\alpha$ does not contain any recursion, then all its members are certainly finite. An equality on $\text{Bool}$ is for instance decidable, but an equality on lists is not.

**Definition 4.5.4.1: (finite types)**
A type $\alpha$ is finite if the set $E$ of all possible expressions of type $\alpha$ is finite.

**Definition 4.5.4.2: (decidable equalities)**
An equality between values of type $\alpha$ is decidable if $\alpha$ is finite. We will denote this (informally) with $\text{Decidable} (=)$.

**Definition 4.5.4.3: (admissibility)**
A proposition $P$ is admissible if $\text{Adm}(+1, P)$ holds, by means of:

\[
\begin{align*}
\text{Adm}(\text{sign, True}) &= \text{True} \\
\text{Adm}(\text{sign, False}) &= \text{True} \\
\text{Adm}(\text{sign, $\neg P$}) &= \text{Adm}(\neg \text{sign, } P) \\
\text{Adm}(\text{sign, $P \land Q$}) &= \text{Adm}(\text{sign, } P) \land \text{Adm}(\text{sign, } Q) \\
\text{Adm}(\text{sign, $P \lor Q$}) &= \text{Adm}(\text{sign, } P) \land \text{Adm}(\text{sign, } Q) \\
\text{Adm}(\text{sign, $P \rightarrow Q$}) &= \text{Adm}(\neg \text{sign, } P) \land \text{Adm}(\text{sign, } Q) \\
\text{Adm}(\text{sign, $P \leftrightarrow Q$}) &= \text{Adm}(\text{sign, } P \rightarrow Q) \land \text{Adm}(\text{sign, } Q \rightarrow P) \\
\text{Adm}(\text{sign, $\forall . P$}) &= \text{Adm}(\text{sign, } P) \\
\text{Adm}(\text{sign, $\exists . P$}) &= \text{Adm}(\text{sign, } P) \\
\text{Adm}(\text{sign, $E_1 = E_2$}) &= \text{Decidable}(=) \lor \text{sign} = +1
\end{align*}
\]

**Assignment 26: (induction on lazy lists)**
For each of the theorems below: prove it or show that it is not admissible.
(a) $\forall_{xs} . \text{finite } xs \rightarrow \text{take} (\text{length } xs) \ = \ xs$

(b) $\forall_{xs} . \text{xs} = \text{ones} \rightarrow \text{finite } xs$

(c) $\forall_{xs} \forall_{p} (\forall_{xs} . \text{map } f \ = \ \text{all } (p \ o \ f) \ = \ \text{all } p \ o \ f) \ = \ xs$

(d) $\forall_{xs \in [a]} \forall_{ys \in [a]} . \text{xs} = \text{ys} \rightarrow \text{xs} == \text{ys}$

In order to reason about non-admissible predicates and/or non-inductive types several techniques have been developed. The most renowned of them are on the one hand the take lemma and its improved version the approximation lemma [Bir98], and on the other hand the class of techniques concerning co-induction based on bisimilarity[Gor95]. To treat them in further detail is outside the scope of this paper.

### 4.5.5 Definedness analysis and the ‘definedness’ tactic

A consequence of the specialized behavior described in Sections 4.5.2-4.5.4 is that reasoning in SPARKLE often involves properties of the form $E = \bot$ or $E \neq \bot$. Dealing with definedness is cumbersome, and should therefore be supported as much as possible. For this purpose, SPARKLE derives definedness information automatically, and offers specialized tactics that make use of this information.

Definedness analysis is the process of deriving definedness information. It is carried out automatically by SPARKLE each time a new goal is constructed. The results of definedness analysis are sets $D$ and $U$, which contain expressions that have been determined to be defined and undefined respectively. The sets $D$ and $U$ are stored with each goal and can be used by various tactics.

The process of definedness analysis starts by assigning all occurring basic values to $D$ and $\bot$ to $U$. It then repeatedly extends $D$ and $U$ by examining the hypotheses that have been introduced, and by making use of strictness and totality properties. The following derivation rules are used for this purpose:

- **Definedness by hypothesis equality.**
  - If a hypothesis $E_0 = E_1$ is available, and $E_i \in D$, then add $E_{1-i}$ to $D$.
  - If a hypothesis $E_0 = E_1$ is available, and $E_i \in U$, then add $E_{1-i}$ to $U$.
  - If a hypothesis $E_0 \neq E_1$ is available, and $E_i \in U$, then add $E_{1-i}$ to $D$.

- **Constructor definedness.**
  - Assume that $C$ is a constructor of arity $n$ with strict arguments $S \subseteq \{1 \ldots n\}$.
  - If the application $A = (C \ E_1 \ldots E_n)$ occurs as a subexpression in the goal, and $\{E_i \mid i \in S\} \subseteq D$, then add $A$ to $D$.
  - If the application $A = (C \ E_1 \ldots E_n)$ occurs as a subexpression in the goal, and $\{E_i \mid i \in S\} \cap U \neq \emptyset$, then add $A$ to $U$.

- **Total function definedness.**
  - Assume that $F$ is a function of arity $n$ which is known to be total.
  - If the application $A = (F \ E_1 \ldots E_n)$ occurs as a subexpression in the goal, and $\{E_i \mid 1 \leq i \leq n\} \subseteq D$, then add $A$ to $D$.
  - If the application $A = (F \ E_1 \ldots E_n)$ occurs as a subexpression in the goal, and $\{E_i \mid 1 \leq i \leq n\} \cap U \neq \emptyset$, then add $A$ to $U$. 
• **Normal function definedness.**
   Assume that \( F \) is a function of arity \( n \) with strict arguments \( S \subseteq \{1 \ldots n\} \). If the application \( A = (F \, E_1 \ldots E_n) \) occurs as a subexpression in the goal, and \( \{E_i \mid i \in S\} \cap U \neq \varnothing \), then add \( A \) to \( U \).

Note that the strictness information for the definedness analysis is available explicitly in the source program, whereas the totality information is assumed to be made available externally (in Sparkle, many functions from StdEnv are hard-coded to be total). Furthermore, to maximize the effectiveness of the definedness analysis, the negation of the current goal is treated as a hypothesis as well.

An important tactic that makes use of definedness analysis is ‘Definedness’. It immediately proves any goal that contains contradictory definedness, which is the case if \( D \) and \( U \) overlap. Note that because the negation of the current goal is treated as a hypothesis, it also proves any goal in which the definedness information implies the validity of the to prove. Although the rules of definedness analysis are relatively simple, it is surprisingly powerful. The Sparkle-tactic ‘Definedness’ is therefore extremely useful, and can be applied often in proofs.

**Assignment 27: (using the Definedness-tactic)**

Prove each of the following properties in Sparkle with the Definedness-tactic.
(a) \( \forall f \forall x. \neg (\text{map } f \, x = \bot) \rightarrow \neg (x = \bot) \)
(b) \( \forall n. \text{eval} (n + 12) \rightarrow \neg (n = \bot) \)
   (see Section 4.5.7 for an introduction of the eval function)
(c) \( \forall n. \forall m. (n \div m = 42) \rightarrow \neg (n + m = \bot) \)
(d) \( \forall n. (7 + (12 \times (13 - n)) = \bot) \rightarrow n = \bot \)

More examples of the use of definedness can be found in [vd06].

### 4.5.6 Specialized behavior of reduction

Because of the presence of logic variables that are introduced by quantors on the property level, reduction in Sparkle is **symbolic**. A logic variable may be instantiated with an arbitrary well-typed expression, and its evaluation does not yield anything. Assuming termination, it is therefore no longer possible to reduce every expression to either a weak head normal form or to \( \bot \).

It is important that reduction in Sparkle carries on as far as possible. For this purpose, Sparkle realizes two extensions in its reduction mechanism that allow reduction to continue, even when a logic variable is encountered on a strict position. The first extension involves ignoring unnecessary strictness annotations; the second extension involves using the results of definedness analysis.

The idea of the first extension is that some strictness annotations can safely be removed without changing the semantics of the program. To illustrate this, take a look at the following three Clean functions:

```
id :: !a -> a  K :: !a !b -> a  length :: ![a] -> Int
id x = x  K x y = x  length [x:xs] = 1+length xs
length [] = 0
```
An exclamation mark before the type of an argument indicates strictness. During evaluation, the strict arguments of a function will always be reduced to weak head normal form before the function is expanded, whereas the non-strict arguments will not. A strictness annotation always changes the reduction behavior of the program; however, it does not always change the semantics.

The strictness annotation in the function \( \text{id} \) does not change the semantics, because the evaluation of its body immediately requires the evaluation of its argument anyway. The same goes for the \( \text{length} \) function, because the pattern match enforces evaluation. In the function \( \text{K} \), the first strictness annotation does not change the semantics, but the second one does. In fact, removing the second annotation would cause \( \text{K} \ x \ \bot = \bot \), where in the current situation \( \text{K} \ x \ \bot = \bot \).

The reduction system of \text{SPARKLE} is able to recognize the different kinds of strictness annotations. In case a strict function argument is encountered like in \( \text{id} \) or in \( \text{K} \) (first annotation), it will be reduced first, but the function will always be expanded afterwards. This is different from reduction in \text{CLEAN}, but semantically sound, and much more user friendly for reasoning (not expanding ‘\( \text{id} \ x \)’ would be really inconvenient). The behavior of \text{SPARKLE} on annotations as in \( \text{K} \) (second annotation) is of course not changed, because that would be semantically unsound. The behavior on annotations as in \( \text{length} \) is not changed either, because the pending pattern match requires its argument to be reduced. Expanding the function therefore does not make much sense, because reduction would be stopped by the pattern match anyway.

**Assignment 28:** *reduction in \text{SPARKLE} (1)*
(a) Build a \text{CLEAN} module with the functions above and load it into \text{SPARKLE}.
(b) Prove \( \forall x. \text{id} \ x = x \)
(c) Prove \( \forall x. \text{K} \ x \ 12 = x \)
(d) Attempt to prove \( \forall x. \text{K} \ 12 \ x = 12 \). Why does this property not hold?

The second extension of reduction is very straightforward: simply make use of the results of the definedness analysis. In case \text{SPARKLE} encounters a function argument whose strictness cannot be removed safely, and on which no pattern match is performed, then the function is allowed to be expanded anyway, as long as the argument expression is an element of \( D \). Again, the argument will be reduced as much as possible first. The second extension allows users to influence the reduction mechanism by means of specifying (and later proving) additional definedness properties.

**Assignment 29:** *reduction in \text{SPARKLE} (2)*
(a) Prove \( \forall x \forall y. \neg (y = \bot) \rightarrow \text{K} \ y \ y = x \)

### 4.5.7 Property specification in \text{CLEAN}

The property language of \text{SPARKLE} is a simple first-order proposition logic only, in which predicates and relations cannot be expressed. However, the possibility to define higher-order functions in the programming language and use them as
Section 4.5.7: Property specification in Clean

boolean predicates gives unexpected expressive power. The higher-order of the programming language can be combined with Sparkle’s first order logic.

A good example of a boolean predicate in Clean is the function eval. The purpose of eval is to fully reduce its argument and return True afterwards. Such an ‘eval’ function is usually used to express evaluation strategies in the context of parallelism [Bur87, THLP98]. We use eval for expressing definedness conditions.

In the module StdSparkle of Sparkle’s standard environment, the function eval is defined by means of overloading. The instance on Char is defined by:

```haskell
class eval a :: !a -> Bool
instance eval Char
  where eval :: !Char -> Bool
        eval x = True
```

In a logical property, (eval x = True) can now be used as a manual definedness condition. The meaning of this condition is identical to ¬(x = ⊥), because:

- If x = ⊥, then (eval x) = (eval ⊥) = ⊥ on the semantic level, because eval is strict in its argument. Therefore, eval x = True is not satisfied.
- If x ≠ ⊥, then x must be equal to some defined basic character b. Therefore, (eval x) = (eval b) = True on the semantic level.
- Note that eval is defined in such a way that it is never equal to False.

On characters, eval is not so interesting. However, by means of overloading, it can easily be defined for lists, and all other kinds of data structures as well. The overloading is used to assume the presence of an eval on the element type:

```haskell
instance eval [a] | eval a
  where eval :: ![a] -> Bool | eval a
        eval [x:xs] = eval x && eval xs
        eval [] = True
```

This instance of eval fully evaluates both the spine of the list and all its elements, and only returns True if this succeeds. It can therefore be used to express the intricate definedness condition that a list is finite and contains defined elements only. This condition cannot be expressed on the property level at all.

Assignment 30: (proofs of properties that use eval)
Using the function eval from StdSparkle, prove the following properties:
(a) ∀x∀xs.eval xs → isMember x xs → eval x
(b) ∀xs.eval xs → sum (map (K 1) xs) = length xs
   (using the strict version of function K, see assignment 28)
(c) ∀x∀p∀xs.eval x → eval xs → eval (map p xs) →
    isMember x (filter p xs) = isMember x xs && p x
All instances of eval have to share certain properties. To prove properties of all members of a certain type class, the recently added tool support for general type classes can be used [vvd04]. With this tool, the properties $\forall_{x}. \text{eval } x \rightarrow x \neq \bot$ and $\forall_{x}. \text{eval } x \neq \text{False}$ can be stated and proven in Sparkle.

A useful variation of eval on lists is the function that evaluates the spine of the list only, but leaves the elements alone. This function expresses the condition that a list is finite. It is defined in StdSparkle as follows:

```haskell
finite :: ![a] -> Bool
finite [x:xs] = finite xs
finite [] = True
```

The boolean predicate finite allows several useful properties to be stated and proven in Sparkle:

**Assignment 31: (proofs of properties that use finite)**

Using the function finite from StdSparkle, prove the following properties:

(a) $\forall_{xs}. \text{finite } xs \rightarrow \text{length } xs \geq 0$
(b) $\forall_{xs}. \text{finite } xs \rightarrow \text{finite \ (reverse } xs\) $
(c) $\forall_{xs}. \text{finite } xs \rightarrow \text{reverse \ (reverse } xs\) = xs$

### 4.6 Related Work

Currently, well-known and widely used proof assistants are Pvs [OSRS01], Coq [The06] and Isabelle [Pau07]. They are all generic provers that are not tailored towards a specific programming language. It is very hard for programmers to reason in them, because they require using a different syntax and a different semantics. For instance, strictness annotations as in Clean are not supported by any existing proof assistant. On the other hand, these well established proof assistants offer features that are not available in Sparkle. Most notably, the tactic language and the logic are much richer than in Sparkle.

At Chalmers University of Technology, the proof assistant Agda [ABB+05] has been developed in the context of the COVER [CDHS01] project. Agda is dedicated to the lazy functional language Haskell [HPW+92]. As in Sparkle, the program is translated to a core-version on which the proofs are performed. Being geared towards facilitating the ‘average’ functional programmer, Sparkle offers dedicated tactics and a dedicated semantics based on graph rewriting. AGDA uses standard constructive type theory on $\lambda$-terms, enabling independent proof checking.

Also as part of the COVER project, it is argued in [DHJG06] that “loose reasoning” is “morally correct”, i.e. that the correctness of a theorem under the assumption that every subexpression is strict and terminating implies the correctness of the theorem in the lazy case under certain additional conditions. The conditions that are found in this way, however, may be too restrictive
Another proof assistant dedicated to Haskell is Era [Win98], which stands for Equational Reasoning Assistant. This proof assistant builds on earlier work initiated by Andy Gill [Gil96]. It is intended to be used for equational reasoning, and not for theorem proving in general. Additional proving methods, such as induction or logical steps, are not supported. Era is a stand-alone application. Unfortunately, it seems that work on this project has been discontinued for a while. Recently, Andy Gill took up the project again, producing a version with an Ajax based interface, under the name of Hera [Gil06], short for Haskell Equational Reasoning Assistant.

In [Min94], a description is given of an automated proof tool which is dedicated to Haskell. It supports a subset of Haskell, and needs no guidance of users in the proving process. Induction is only applied when the corresponding quantor has been marked explicitly in advance. The user, however, cannot further influence the proving process at all, and cannot suggest tactics to help the prover in constructing the proof.

Another proof assistant that is dedicated to a functional language is Evt [NFG01], the Erlang Verification Tool. However, Erlang differs from Clean, because it is a strict, untyped language which is mainly used for developing distributed applications. Evt has been applied in practice to larger examples.

The Programatica project of the Pacific Software Research Center in Oregon (www.cse.ogi.edu/PacSoft/projects/programatica) is another project that aims to integrate programming and reasoning. They intend to support a wide range of validation techniques for programs written in different languages. For functional languages they use P-logic, which is based on a modal $\mu$-calculus in which undefinedness can also be expressed. In the Programatica project, properties are mixed with the Haskell source.

Properties about functional programs are proved by hand in many textbooks, for instance in [Bir98]. Also, several articles (for instance [BS01]) make use of reasoning about functional programs. It seems worthwhile to attempt to formalize these proofs in Sparkle. In programming practice, however, reasoning about functional programs is scarcely used.

### 4.7 Conclusions

In this paper, we have presented a thorough description of the dedicated proof assistant Sparkle, which is integrated in the distribution of the lazy functional programming language Clean. We have introduced Sparkle in detail, both on the theoretical and on the practical level. On the theoretical level, we have explained the process of formal reasoning in general, and Sparkle’s dedicated support for it in specific. On the practical level, we have provided an extensive tutorial of the actual use of Sparkle.

The tutorial not only covers the fundamental functionality of Sparkle, but also explains several of its advanced features that are specific for reasoning about
lazy functional programs. Assignments are included at various points in the tutorial; they allow useful hands-on experience with SPARKLE to be obtained in a guided way. After completion of the tutorial, anyone with a basic understanding of functional programming will be able to make effective use of SPARKLE in practice, and will be able to prove small to medium properties with little effort.

Furthermore, we also hope to have sparked an interest in making use of formal reasoning to show important properties of functional programs. With the right tool support, this is already feasible for many smaller examples, and provides an enjoyable challenge for bigger programs too!
Chapter 5

Proof Tool Support for Explicit Strictness
Presented at IFL'05, published in LNCS proceedings volume 4015.

Abstract. In programs written in lazy functional languages such as for example CLEAN and HASKELL, the programmer can choose freely whether particular subexpressions will be evaluated lazily (the default) or strictly (must be specified explicitly). It is widely known that this choice affects resource consumption, termination and semantics in several ways. However, functional programmers tend to be less aware of the consequences for logical program properties and formal reasoning.

This paper aims to give a better understanding of the concept of explicit strictness and its impact on properties and reasoning. It will be shown that explicit strictness may make reasoning more cumbersome, due to the introduction of additional definedness conditions.

Fortunately, these definedness conditions can be handled quite effectively by proof assistants. This paper describes the specific support that is offered by SPARKLE for expressing and proving definedness conditions. This support makes reasoning with explicit strictness almost appear like reasoning without explicit strictness. To our knowledge, SPARKLE is currently the only proof assistant with such strictness specific support.

5.1 Introduction

Lazy functional programming languages, such as for example HASKELL and CLEAN, are excellent for developing readable and reliable software. One of their key features is lazy evaluation, which makes it possible to adopt a natural, almost mathematical, programming style. The downside of lazy evaluation, however, is lack of control; it becomes very difficult to predict when subexpressions will be evaluated, which makes resource management a non-trivial task.

This issue has been addressed by the introduction of explicit strictness, with which a functional programmer can enforce the evaluation of a subexpression by hand. Adding explicit strictness can indeed change the resource consumption of programs significantly, and it is therefore used a lot in practice. Moreover,
explicit strictness can easily be incorporated in the semantics of functional languages, and is therefore theoretically sound as well.

Not all is well, however. In this paper, we will show that the addition (or removal) of strictness to programs may also give rise to many unexpected (and undesirable) effects. Of course, some effects are already widely known, such as for example the possible introduction of non-termination. However, less widely known to programmers, is that explicit strictness may:

- break program properties, forcing them to be reformulated by adding (or removing) definedness conditions;
- break proof rules that are based on reduction, adding a new definedness precondition to them that has to be shown to be satisfied in order for the rule to be applicable.

In other words: changing strictness properties can have serious consequences for formal reasoning. In general, the addition of explicit strictness makes reasoning more cumbersome and forces one to pay attention to technical details that are not so interesting. Fortunately, exactly these kinds of details can be dealt with effectively by means of a proof assistant.

We will demonstrate the facilities that Sparkle, the proof assistant for Clean, offers for dealing with definedness conditions. Sparkle has been introduced in [dvP02], but its specific definedness facilities have not yet been addressed in any publication. The definedness facilities of Sparkle include:

1. a dedicated proof rule for proving definedness conditions;
2. an upgraded reduction system that takes advantage of available definedness information; and
3. a mechanism to conveniently denote definedness conditions.

With these facilities, definedness conditions are often handled in the background and are hidden from the user completely, making reasoning with strictness look like reasoning without strictness.

This paper is structured as follows. First, in Section 5.2 the concept of explicit strictness is introduced, both informally and formally. Also, its effects on program semantics and program transformations are discussed. Then, in Section 5.3 the effects of explicit strictness on program properties and reasoning will be examined. The three kinds of support that Sparkle offers for this purpose will be introduced in Section 5.4. Finally, Sections 5.5 and 5.6 discuss related work and conclusions.

5.2 The Concept of Explicit Strictness

Although it is seldom mentioned in publications, explicit strictness is present in almost every real-world lazy program. Explicit strictness is used for:
Section 5.2.1: When Strictness is not an option but a must

- improving the efficiency of data structures (e.g. strict lists),
- improving the efficiency of evaluation (e.g. functions that are made strict in arguments that always have to be evaluated),
- enforcing the evaluation order in interfacing with the outside world (e.g. an interface to an external C-call is defined to be strict in order to ensure that the arguments are fully evaluated before the external call is issued).

Language features that are used to explicitly enforce strictness include:

- type annotations (in functions: CLEAN and in data structures: CLEAN, HASKELL),
- special data structures (unboxed arrays: CLEAN, HASKELL),
- special primitives (seq: HASKELL),
- special language constructs (let!, #!: CLEAN).

Implementers of real-world applications make it their job to know about explicit strictness, because without it essential parts of their programs would not work properly. The compiler generates code that takes strictness annotations into account by changing the order of evaluation. It is often thought that the only effects of changes in evaluation order can be on the termination properties of the program as a whole and on the program’s resource consumption (with respect to space or time). Therefore, strictness is usually considered an implementation issue only.

However, in the following subsections we will show that explicit strictness is far from an implementation issue only. In Section 5.2.1 it is illustrated that strictness has a fundamental influence on program semantics, because explicit strictness is not just an ‘option’ that may be ignored by the reduction system, but a ‘must’ that changes reduction order. An example of how radical this influence can be, is given in Section 5.2.2. Finally, to deal with that influence, formal semantics are extended with strictness in Section 5.2.3.

5.2.1 When Strictness is not an option but a must

With explicit strictness, performing an evaluation is not anymore just an option. Instead each explicit strictness annotation constitutes an evaluation obligation that has to be fulfilled before proceeding further. We will illustrate the consequences of this changed evaluation with the following example.

Consider the following CLEAN definition of the function $f$, which by means of the !-annotation in the type is made explicitly strict in its first argument. In HASKELL a similar effect can be obtained using an application of seq.

\[
\begin{align*}
f & : \texttt{!Int} \to \texttt{Int} \\
f & x = 5
\end{align*}
\]
Without the strictness annotation, the property $\forall_x[f \ x = 5]$ would hold unconditionally by definition. Now consider the effects of the strictness annotation in the type which makes the function $f$ strict in its argument. Clearly, the proposition $f \ 3 = 5$ still holds. However, $f \ \text{undef} = 5$ does not hold, because $f \ \text{undef}$ does not terminate due to the enforced evaluation of $\text{undef}$. Therefore, $\forall_x[f \ x = 5]$ does not hold unconditionally. The property can be fixed by adding a definedness condition using the special symbol $\bot$, denoting undefined. This results in $\forall_x[x \neq \bot \rightarrow f \ x = 5]$, which does hold for the annotated function $f$.

Another consequence is that the definition of $f$ cannot just be substituted in all its occurrences. Instead it is only allowed to substitute $f$ when it is known that its argument $x$ is not undefined. This has a fundamental impact on the semantics of function application.

The addition of an exclamation mark by a programmer is therefore more than just a harmless annotation. It also has an effect on the logical properties of functions. Changes in logical properties are not only important for the programmer but also for those who work on the compiler. Of course, it is obvious that code has to be generated to accommodate the strictness. Less obvious however, is the consequences adding strictness may have on the correctness of program transformations. There can be far-reaching consequences on various kinds of program transformations. An example of such a far-reaching consequence is given in the next subsection.

5.2.2 Dramatic Case of the Influence of Explicit Strictness

The CLEAN compiler uses term graph rewriting semantics [BvG+87a] to incorporate pattern matching, sharing and cycles. With term graph rewriting semantics, on right-hand sides of definitions those parts that are not connected to the root cannot have any influence on the outcome. These definitions are thrown away in a very early stage of the compilation process. Consequently, possible syntactic and semantic errors in such disconnected definitions may not be spotted by the compiler. This can be annoying but it is consistent with the term graph rewriting semantics. When strictness comes into the picture, however, this early connectedness program transformation of the compiler is no longer semantically valid.

This is illustrated by the following example. Take the following CLEAN programs with definition $K \ x \ y = x$:  

```
Start
  #! y = undef
  = K 42 y
Start
  #! y = undef
  = 42
```

The programs use the $#!$-notation of CLEAN which denotes a strict let. The strict let will be formally defined in Section 5.2.3. It forces $y$ to be evaluated before the result of $\text{Start}$ is computed. In HASKELL the same effect can be achieved using a $\text{seq}$.

For the left program, due to the $#! \ y$ must be evaluated first. So, the result of the program is: "Error: undefined!".
For the right program one would expect the same result. But, the result is different since the compiler removes unconnected nodes before any analysis is done, transforming the right program into \texttt{Start = 42}. So, the result of the right program is 42. This makes the right program a wrong program and the left program the right program. Clearly, this is an unwanted situation.

Due to the combination of connectedness and explicit strictness CLEAN programs are not always referentially transparent anymore. The meaning is not always preserved during reduction and it is not always sound to substitute a definition. Of course, this situation is acknowledged as a bug in the compiler for several years now. The consequences of removing this bug, however, are so drastic for the structure of the compiler that at this point in time this bug still remains to be present.

It may be a relief to the reader that Sparkle’s mixed lazy/strict semantics are not based on connectedness.

5.2.3 Modeling Explicit Strictness in Formal Semantics

The semantics of lazy functional languages have been described elegantly in practice in various ways: both operationally and denotationally, in terms of a term-graph rewrite system, in [Lau93]; or just operationally, in terms of a graph rewrite-system, in [Pv93]. All these semantics are well established, are widely known and accepted in the functional language community, and have been used for various kinds of theoretical purposes.

The basic forms of all these semantics, however, are limited to lazy expressions in which no explicit strictness is allowed to occur. If one wants to include strictness, an upgrade is required, because the introduction of strictness in an expression has an effect on its meaning that cannot be described in terms of existing concepts. In other words, strictness has to be accounted for on the semantic level as well.

As a starting point we will use the operational semantics of Launchbury [Lau93]. We extend this to a mixed lazy/strict semantics, which is able to cope with laziness as well as with strictness. In this paper, we will limit ourselves to the basic definitional components of the mixed semantics. The formal proofs that our extension is correct are therefore not included; however, these proofs can be built analogously to the proofs in [Lau93].

We will choose to extend expressions with the \textit{strict let}, which is the basic primitive for denoting strictness in CLEAN. The strict let is a variation of the normal let, which only allows the actual sharing to take place after the expression to be shared has first successfully been reduced to weak head normal form. Moreover, it only allows a single non-recursive expression to be shared at a time; this keeps the strict let as simple as possible, yet still sufficiently powerful.

In the base set of expressions, we will include basic values \((b \in \text{BasicValue})\), constructors \((c \in \text{Constructor})\) and case distinctions in the same manner as in [Lau93]. Furthermore, we will also include a constant expression \(\bot\) that denotes the undefined computation. This \(\bot\) can simply be regarded as an ab-
breviation for \texttt{let x = x in x}. Adding the strict let to this set of expressions leads to:

\[
\begin{align*}
e \in \text{Exp} &::= \lambda x. e \quad \text{(lambda expressions)} \\
&| x \quad \text{(variables)} \\
&| e x \quad \text{(applications)} \\
&| \text{let } x_1 = e_1 \cdots x_n = e_n \text{ in } e \quad \text{(let expressions)} \\
&| b \quad \text{(basic values)} \\
&| c x_1 \cdots x_n \quad \text{(constructor applications)} \\
&| \text{case } e \text{ of } \{ c_i y_1 \cdots y_m \rightarrow e_i \}_{i=1}^n \quad \text{(case distinctions)} \\
&| \perp \quad \text{(undefined expression)} \\
&| \text{let! } x = e \text{ in } e \quad \text{(strict let expressions)}
\end{align*}
\]

Due to its similarity with the normal let, the strict let is a convenient primitive that can be added to the semantical level with minimal effort. Naturally, all forms of explicit strictness can easily be expressed in terms of the strict let. This also goes for the basic HASKELL primitive, \texttt{seq}:

\[
\text{seq } e_1, e_2 \text{ is equivalent to } \text{let! } x = e_1 \text{ in } e_2.
\]

Launchbury describes both an operational and a denotational semantics, which both have to be updated to cope with the strict let. Here, we treat the extension of the operational semantics only. This semantics is given by means of a multi-step term-graph rewrite system which has to be extended with a rule for the strict let. The new rule is much like the rule for the normal let, but also demands the reduction of the shared expression to weak head normal form as an additional precondition:

\[
\begin{array}{c}
\Gamma, x_1 \mapsto e_1 \cdots x_n \mapsto e_n : e \Downarrow \Delta \upharpoonright z \\
\hline
\Gamma : \text{let } x_1 = e_1 \cdots x_n = e_n \text{ in } e \Downarrow \Delta \upharpoonright z
\end{array}
\]

\text{(for the technical details of this definition: see [Lau93])}

The addition of this single \texttt{StrictLet} rule is sufficient to incorporate the concept of explicit strictness in a formal semantics. Our extension is equivalent to the one that is introduced in [BKT00] for dealing formally with parallelism. In [BKT00] \texttt{seq} is used as the basic primitive to denote explicit strictness. Using the equivalence of \texttt{seq} and \texttt{let!} sketched above, the proofs of soundness and computational adequacy that were given in [BKT00] can be applied to our mixed semantics as well.
5.3 Reasoning in the Context of Strictness

In the previous sections, a general introduction to the concept of explicit strictness has been provided and its, more or less obvious, effects on programs and semantics have been discussed. In this section, the effect of strictness on reasoning will be described. We will show that adding or removing strictness requires program properties to be reformulated. As a consequence, the proofs of the reformulated properties may have to be redone from scratch. In addition, certain proof rules may no longer be applicable and have to be replaced as well.

The effects of strictness on reasoning are not so commonly known, mainly because programming and reasoning are usually separate activities that are not carried out by the same person. With this paper, we strive to show that the effects of strictness on reasoning are quite profound and should not be ignored.

5.3.1 Strictness and Logical Properties

A logical (equational) property about a program is constructed by means of logical operators ($\forall$, $\exists$, $\land$, $\lor$, $\rightarrow$, $\neg$) out of basic equations of the form $E_1[x_1 \ldots x_n] = E_2[x_1 \ldots x_n]$, where $x_1 \ldots x_n$ are the variables that have been introduced by the quantors. The equations in a property can be divided into a number of conditions that precede a single obligatory conclusion. A property with conclusion $E_1 = E_2$ denotes that $E_1$ may safely be replaced by $E_2$ in all contexts, if properly instantiated and if all conditions are satisfied.

Semantically, two expressions may only be replaced by each other if either: (1) they both compute the exact same value; or (2) they both do not compute any value at all. Note that this is a total semantics, and an expression that does not terminate, or terminates erroneously, may not be replaced by an expression that successfully computes a value, nor vice versa.

If explicit strictness is added to or removed from a program, the value that it computes on success is not affected, but the conditions under which it produces this defined value are. Unfortunately, if the definedness conditions of an expression $E_1$ are changed, but the definedness conditions of $E_2$ stay the same, then a previously valid equation $E_1 = E_2$ will become invalid, because the replacement of $E_1$ by $E_2$ is no longer allowed.

In other words: the addition or removal of strictness to programs may cause previously valid logical properties to be broken. From a proving point of view this is a real problem: suppose one has successfully proved a difficult property by means of a sequence of lemmata, then the invalidation of even a single lemma may cause a ripple effect throughout the entire proof! The adaptation to such a ripple effect is both cumbersome and resource-intensive.

Unfortunately, the invalidation of logical properties due to changed strictness annotations is quite common. This invalidation can usually be fixed, either by the addition or, quite surprisingly, by the removal of definedness conditions. This is illustrated briefly by the following two examples:

Example of the addition of a condition:
∀ₕ,ₔₓₛ[map (ₕ o ₔ) 𝑥ₛ = map ₕ (map ₔ 𝑥ₛ)]

**Affected by strictness:**
This property is valid for lazy lists, but invalid for element-strict lists. Note that no assumptions can be made about the possible strictness of ₕ or ₔ. Instead, the property must hold for all possible functions ₕ and ₔ.

**Invalid in the strict case because:**
Suppose 𝑥ₛ = [12], ₁₂ = ⊥ and ₕ (₁₂) = 7.
Then map (ₕ o ₔ) 𝑥ₛ = [7], both in the lazy and in the strict case.
However, map ₕ (map ₔ 𝑥ₛ) = [7] in the lazy case, but ⊥ in the strict case.

**Extra definedness condition for the lazy case:**
The problematic case can be excluded by demanding that for all elements of the list ₔ 𝑥 can be evaluated successfully.

**Reformulated property for the strict case:**
∀ₜ,ᵰ,ᵪ xe_ᵪ[∀ xe xe [g x ≠ ⊥] → map (ₕ o ₔ) xe = map ₕ (map ₔ xe)].

**Example of the removal of a condition:**
∀ xe [finite xe → reverse (reverse xe) = xe]

**Affected by strictness:**
This property is valid both for lazy lists and for spine-strict lists. However, the condition finite xe is satisfied automatically for spine-strict lists. In the spine-strict case, the property can therefore safely be reformulated (or, rather, optimized) by removing the finite xe condition.

**Invalid without finite condition in the lazy case because:**
Suppose xe = [1, 1, 1, . . . ].
Then reverse (reverse xe) = ⊥, both in the lazy and in the strict case.
However, xe = ⊥ in the strict case, while it is unequal to ⊥ in the lazy case.

**Reformulated property for the strict case:**
∀ xe [reverse (reverse xe) = xe]

In Section 5.4.3 it will be shown how mathematical conditions such as finite xe and ∀ xe [g x ≠ ⊥] can be expressed within the SPARKLE framework. In principle, all invalidated properties can be fixed this way. The definedness conditions to be added can be obtained by carefully considering the consequences of components of quantified variables to be undefined. Such an analysis is far from easy, however, and it is easy to forget certain conditions. On paper, this may lead to incorrect proofs; when using a proof assistant, this makes it impossible to prove the property at all.

An automatic analysis to obtain definedness conditions would be helpful. This does not seem too far-fetched. An idea is to extend the GAST-system (see [KATP03]) for this purpose. With GAST, it is possible to automatically generate valid values for the quantified variables and test the property on these values. However, GAST currently is not able to cope with undefinedness.
5.3.2 Strictness and Formal Reasoning

Formal reasoning is the process in which formal proofs are constructed for logical program properties. These proofs are constructed by the repeated application of proof steps. Each proof step can be regarded as a function from a single property to a list of new properties. The conjunction of the produced properties must be logically stronger, and hopefully also easier to prove, than the input property.

In the previous section, it has been shown that the addition or removal of strictness to programs often requires a reformulation of the associated logical properties. This is not the only cumbersome effect of strictness on reasoning, however. A second problem is that strictness changes the behavior of reduction, and consequently also of proof steps that make use of reduction. This in turn may cause existing proofs to become invalid.

A proof step that makes use of reduction is based on the observation that if \( e_1 \) reduces to \( e_2 \), then \( e_1 \) is also semantically equal to \( e_2 \), and therefore \( e_1 \) may safely be replaced with \( e_2 \) within a logical property to be proved. It is clear that this relation is changed by the introduction of strictness. It is not intuitively clear where this change is problematic for the actual proof process.

The hidden reason is the availability of *logical expression variables* within propositions. Such a variable denotes an ‘open position’, to be replaced with a concrete expression later. It is introduced and bound by means of a (existential or universal) quantor. When reduction is forced, due to explicit strictness, to reduce such a variable to weak head normal form, the following problem occurs:

- Suppose that \( e \) is an expression in which the variable \( x \) occurs lazily.
- Suppose that \( e \) reduces to \( e' \).
- Suppose that within \( e \), \( x \) is now marked as explicitly strict.
- Then, the strict version cannot be reduced at all, because the required preparatory reduction of \( x \) to weak head normal fails.

In other words: the introduction of explicit strictness causes a previously valid reduction to become invalid. This in turn causes proof steps that depend on it to become invalid. That in turn causes the proof as a whole to become invalid. This effect is illustrated in the following basic example:

**Property:** \( \forall x [\text{id } x = x] \).

**Proof:** Introduce \( x \). Reduce \( (\text{id } x) \) to \( x \). Use reflexivity. QED.

**Validity:** This proof is only valid if the first argument of \( \text{id} \) is not explicitly marked as strict. If it is, then the strictness annotation forces \( x \) to be reduced to weak head normal form before the application \( (\text{id } x) \) may be expanded. Because \( x \) cannot be brought into weak head normal form, \( (\text{id } x) \) cannot be reduced at all, and the proof sketched above becomes invalid.

This effect actually occurs quite frequently, which is a big nuisance. It causes many previously valid proof steps to become invalid, and therefore requires the
proofs themselves to be revised. Fortunately, this revision is often easily realized. A general solution, which usually suffices, is to distinguish explicitly between \( x = \bot \) and \( x \neq \bot \). In the first case, the whole expression reduces to \( \bot \). In the second case, it is statically known that \( x \) has a weak head normal form, and reduction is therefore allowed to continue in the same way as in the lazy case.

Nevertheless, the introduction of explicit strictness makes reasoning more difficult. To deal with this problem, the proof assistant SPARKLE offers specific support to deal with explicit strictness. The following section is devoted to explaining this support.

### 5.4 Support for Explicit Strictness in SPARKLE

SPARKLE [dvP02] is CLEAN’s dedicated proof-assistant. Apart from its location of origin SPARKLE is used rather intensively in Budapest (Object Abstraction [THK06]) and Dublin (I/O models [DBv04]). SPARKLE works directly on a desugared version of CLEAN, called CORE-CLEAN. SPARKLE allows properties of functions to be expressed using a first-order logic. Predicates are not supported. SPARKLE offers the usual operators and quantors with the restriction that quantification is only allowed over typed expressions and propositions.

**Basic units:** True, False, \( e_1 = e_2 \), \( x \)

**Operators:** \( \neg, \wedge, \vee, \to, \leftrightarrow \)

**Quantors:** \( \forall, \exists \)

SPARKLE is aimed towards making proving possible for the programmer. It contains many features to lower the threshold to start with proving theorems about programs, such as:

- it can be called from within the CLEAN Integrated Development Environment;
- it can load a complete CLEAN project including all the modules of the project;
- the proof environment is highly interactive and allows a wide range of information to be displayed in separate windows at the user’s will;
- the proof tactics are dedicated to the programming language.

SPARKLE’s reduction semantics are based on term graph rewriting. SPARKLE has a total semantics. The constant expression \( \bot \) is used to represent the “undefined” value. Both non-terminating reductions and erroneous reductions are equal to \( \bot \). For example: \( \text{hd} [\ ] \) reduces to \( \bot \) on SPARKLE’s semantic level. Error values propagate stepwise to the outermost level. For example: \( (\text{hd} [\ ] + 7 \) reduces to \( \bot + 7 \) reduces to \( \bot \).
Section 5.4.1: The ‘Definedness’ Tactic of SPARKLE

SPARKLE’s semantics of equality are based on reduction in a manner which is independent of the reduction strategy. The equality copes with infinite reductions and equalities between infinite structures using the concept of an observation of an expression. The observation of an expression is obtained by replacing all its redexes by ⊥. What remains is the fully evaluated part. Two expressions e₁ and e₂ are equal if: (1) for all reducts r₁ of e₁, there exists a reduct r₂ of e₂ such that the observation of r₁ is smaller than the observation of r₂; and (2) also the analogue property holds for all reducts of e₂. The observational ordering is such that an expression r₁ is smaller than r₂ if r₂ can be obtained by substituting subexpressions for ⊥’s in r₁.

Being dedicated to the use of a lazy programming language, SPARKLE generates on the one hand definedness conditions for extensionality (f = g not only requires f x = g x for all x, but also f = ⊥ ↔ g = ⊥), induction (base case for ⊥) and case-distinction (base case for ⊥ as well). On the other hand SPARKLE also offers specific support for reasoning with definedness conditions in the context of explicit strictness. To our knowledge, SPARKLE is currently the only proof assistant that fully supports explicit strictness in the context of a lazy functional programming language. The specific support consists of three components:

1. a specific ‘Definedness’ tactic; and
2. a smart reduction proof step: the ‘Reduce’ tactic;
3. using an ‘eval’ function to denote definedness conditions.

These three kinds of support are explained in detail in the following sections.

5.4.1 The ‘Definedness’ Tactic of SPARKLE

Definedness conditions on variables and expressions occur frequently in proofs. They are introduced by various tactics that take explicit strictness into account, such as ‘Induction’, ‘Case’ and ‘Assume’. These conditions usually appear in parts of the proof that are not in the main line of reasoning. Therefore, one wishes to get rid of them as soon as possible with as little effort as possible.

Unfortunately, proving definedness conditions often involves several small reasoning steps as is illustrated by the following example:

Example of proving a definedness condition:
\[ \forall x,y [\neg(x = \bot) \rightarrow y = (\text{let } z = x \text{ in } \text{Cons } 7 z) \rightarrow \neg(y = \bot)]. \]

Proof without the Definedness tactic:
Introduce x and y.
Assume H1: \(\neg(x = \bot)\) and H2: \(y = (\text{let } z = x \text{ in } \text{Cons } 7 z)\).
Using H1, reduce H2 to H2': \(y = \text{Cons } 7 x\).
Rewrite H2' in the goal, which leaves \(\neg(\text{Cons } 7 x = \bot)\) to be proved.
This follows from the injectivity of \text{Cons}.
QED.
Chapter 5: Proof Tool Support for Explicit Strictness

In Sparkle the ‘Definedness’ tactic is introduced to remove the burden of all such small proofs from the user. This tactic analyzes all subexpressions that occur in the hypotheses that have been introduced, and attempts to determine if they are ‘defined’ (statically known to be unequal to ⊥) or ‘undefined’ (statically known to be equal to ⊥). If the tactic finds any overlap between the defined expressions and the undefined ones, it then proves any goal by contradiction.

The tactic is implemented by the following algorithm, which assumes that it is activated in a goal with hypotheses $H_1 \ldots H_n$ and a statement to prove of the form $\forall x_1 \ldots x_i \left[ P_1 \rightarrow (P_2 \rightarrow \ldots (P_j \rightarrow Q) \ldots) \right]$ (note that $i$ and $j$ can be zero for no top-level quantors or implications, making the form universal):

1. Collect as many known equalities as possible in the set $Eq$ as follows:
   - for all $1 \leq i \leq n$, if $H_i$ states $e_1 = e_2$, then add $(e_1 = e_2)$ to $Eq$;
   - for all $1 \leq j \leq n$, if $P_j$ states $e_1 = e_2$, then add $(e_1 = e_2)$ to $Eq$;
   - if $Q$ states $\neg(e_1 = e_2)$, then add $(e_1 = e_2)$ to $Eq$.

   Note that $\neg Q$ can be used as a hypothesis here, because $Q$ and $(\neg Q \rightarrow False)$ are logically equivalent.

2. Collect as many known inequalities as possible in the set $Eq$ as follows:
   - for all $1 \leq i \leq n$, if $H_i$ states $\neg(e_1 = e_2)$, then add $(e_1 \neq e_2)$ to $Eq$;
   - for all $1 \leq j \leq n$, if $P_j$ states $\neg(e_1 = e_2)$, then add $(e_1 \neq e_2)$ to $Eq$;
   - if $Q$ states $e_1 = e_2$, then add $(e_1 \neq e_2)$ to $Eq$.

3. Determine $X$, the set of all subexpressions that occur in the goal as a whole.

4. Compute $D = \{ e \in X \mid Eq \vdash Defined(e) \}$ and $U = \{ e \in X \mid Eq \vdash Undefined(e) \}$.

5. If $D$ and $U$ overlap, then the tactic proves the goal.

In Tables 5.1 and 5.2, two derivation systems are defined, one for statically computing $Eq \vdash Undefined(e)$ and one for statically computing $Eq \vdash Defined(e)$. The derivation rules are described formally using the representation of expressions given in Section 5.2.3. In practice, Sparkle implements procedural variations of the derivation systems that have been lifted to Core-Clean. Proving the soundness of the derivation systems (meaning that expressions in $D$ have a weak head normal form, while those in $U$ have not) is left as future work.

The special tactic ‘Definedness’ is quite powerful and very useful in practice. It can be used to automatically get rid of almost all kinds of valid definedness conditions that have been stated in order to keep reduction going in strict contexts. The proof of the example can be simplified with it as follows:

Example of proving a definedness condition (2):

$\forall x,y[\neg(x = \bot) \rightarrow y = (let! z = x \text{ in } \text{Cons } z) \rightarrow \neg(y = \bot)]$. 
Section 5.4.2: The ‘Reduce’ Tactic of Sparkle

\[
\begin{array}{c}
\text{Eq} \vdash \text{Undefined}(\bot) \\
(e_1 = e_2) \in \text{Eq} \quad \text{Eq} \vdash \text{Undefined}(e_2) \\
\text{Eq} \vdash \text{Undefined}(e_1) \\
\text{Eq} \vdash \text{Undefined}(\bot) \\
}\end{array}
\]

\[
\begin{array}{c}
\text{Eq} \vdash \text{Undefined}(e_1) \\
(\text{Eq}, (x_1 = e_1), \ldots, (x_n = e_n)) \vdash \text{Undefined}(e) \\
\text{Eq} \vdash \text{Undefined}(\text{let } x_1 = e_1 \ldots x_n = e_n \text{ in } e) \\
\text{Eq} \vdash \text{Defined}(e_1) \\
\text{Eq} \vdash \text{Undefined}(e) \\
\text{Eq} \vdash \text{Undefined}(\text{let! } x = e_1 \text{ in } e) \\
\text{Eq} \vdash \text{Undefined}(\text{case } e \text{ of } \{c_1, y_1 \cdots y_m \rightarrow e_i\}^m_{i=1}) \\
\text{Eq} \vdash \text{Undefined}(\text{let! } x = e_1 \text{ in } e) \\
\end{array}
\]

Table 5.1: Derivation system for statically computing undefinedness

Proof with the Definedness tactic:

Apply Definedness.
Q.E.D.

Explanation:

Eq is computed to be \{\{(x \neq \bot), (y = (\text{let! } z = x \text{ in Cons } 7 \ z)), (y = \bot)\}\}.

Derive(1) \text{Eq} \vdash \text{Undefined}(\bot) (\text{base case})

Derive(2) \text{Eq} \vdash \text{Undefined}(y) (\text{from 1, with equality})

Derive(3) \text{Eq} \vdash \text{Defined}(x) (\text{from 1, with inequality})

Derive(4) \text{Eq}, z = x \vdash \text{Defined}(\text{Cons } 7 \ z) (\text{base case})

Derive(5) \text{Eq} \vdash \text{Defined}(\text{let! } z = x \text{ in Cons } 7 \ z) (\text{from 3+4, with let! rule})

Derive(6) \text{Eq} \vdash \text{Defined}(y) (\text{from 5, with equality})

Contradiction between 2 and 6.

5.4.2 The ‘Reduce’ Tactic of Sparkle

One of the proof steps (or tactics, as they are usually called in the context of mechanized proof assistants) that is made available by Sparkle is ‘Reduce’. This tactic applies reduction within the current logical property to be proved.

Sparkle operates on a basic functional language with a reduction mechanism similar to the one given in Section 5.2.3. The reduction tactic of Sparkle does not necessarily have to correspond completely to the formal reduction relation of this language; instead, it suffices that it is sound, meaning that it may only transform \(e_1\) to \(e_2\) if \(e_1 = e_2\) formally holds. Of course, the tactic does have to be based on reduction, because it must look like normal reduction to the end-user.

This degree of freedom is used by Sparkle to offer specific support for the reduction of explicitly strict subexpressions that contain logical expression
variables. The aim of this support is to hide the cumbersome effects of strictness to the user, allowing the same proof style and the same proof rules to be used both for the lazy and for the strict case.

The support offered by SPARKLE manifests itself in the following customized behavior when reduction encounters explicit strictness of the form \( \text{let}! \ x = e_1 \) in \( e \):

- First, reduction is recursively applied to \( e_1 \) as usual.
- If this results in either \( \bot \) or a weak head normal form, then reduction continues as usual.
- Suppose that, due to logical expression variables, the recursive reduction cannot be completed and instead results in some expression \( e'_1 \) that is neither \( \bot \) nor a weak head normal form.
- Then, and this is new, apply the same definedness analysis that was described in Section 5.4.1. If this analysis determines that \( e_1 \) is defined \((e_1 \in D)\), then reduction is allowed to continue by expanding the strict let.

This expansion is semantically sound, because the definedness analysis shows that \( \neg (e_1 = \bot) \), which implies that \( e_1 \) has a weak head normal form, even though it is not known at this point what it actually looks like.

- If this fails, then add \( x = \bot \) as hypothesis and perform another definedness analysis. If this analysis shows that \( e \) is undefined \((e \in U)\), then reduction is allowed to continue by expanding the strict let.

This expansion is semantically sound, because the definedness analysis shows that \( x = \bot \to e = \bot \), which means that the explicit strictness annotation has no effect on semantics and may safely be ignored.
If either of the two ‘escape clauses’ succeed, then it seems to the user as if reduction has the same effect in the strict case as in the lazy case. In other words: by silently checking for additional conditions, SPARKLE can sometimes hide the cumbersome effects of explicit strictness on reduction altogether.

To illustrate the additional power of the reduction mechanism, consider the following two basic examples:

**Example of continuation of reduction:**
Suppose that datatype `Tree a` is defined as follows:
```
:: Tree a = Leaf | Edge !a !(Tree a) !(Tree a)
```
Suppose that the function `treeDepth` has the following signature:
```
treeDepth :: !(Tree a) -> Int
```
Suppose that the logical expression variable `x` (of type `Tree Int`) and the hypothesis `¬ (x = ⊥)` have both been introduced earlier in the proof. Then, SPARKLE allows the application `treeDepth (Edge 7 Leaf x)` to be expanded, because by means of recursive analysis SPARKLE is able to determine that `Edge 7 Leaf x` is unequal to ⊥.

**Note that:** this example uses strict constructors and strict functions, which can be considered as notational sugar for the strict let.

**Example of increased stability of proofs:**
Suppose that the identify function is defined as follows:
```
id :: !a -> a
id x = x
```
SPARKLE determines statically that if the argument of the function is undefined, then the result of the function will be undefined as well. Therefore, SPARKLE allows applications of `id` to be expanded, regardless of its argument.

The proof of Section 5.3.2, which was shown to be invalid with a standard reduce tactic, in fact becomes valid when the powerful strictness specific ‘Reduce’ tactic of SPARKLE is used.

**Note that:** this example uses strict functions as well.

### 5.4.3 Using the ‘eval’ Function for Definedness Conditions

In many cases, it may seem impossible to express definedness conditions just using the first-order logic of SPARKLE. For instance, spine evaluation of data-structures is very hard to express. However, the possibility to define functions in the higher-order programming language and the possibility to use these functions as predicates gives unexpected expressive power. The higher-order of the programming language can be combined with the SPARKLE’s first order logic.

On the programming level we define a function `eval`. The purpose of this function is to fully reduce its argument and return `True` afterwards. Such an ‘eval’ function is usually used to express evaluation strategies in the context of
Chapter 5: Proof Tool Support for Explicit Strictness

parallelism [Bur87, THLP98]. We use eval for expressing definedness conditions.

In the standard program library of Sparkle (StdSparkle), the function eval is defined by means of overloading. The instance on characters is defined by:

```haskell
class eval a :: !a -> Bool

instance eval Char
  where  eval :: !Char -> Bool
         eval x = True
```

Now, in a logical property, (eval x) can be used as termination condition. As is usual in proof assistants, this is equivalent to (eval x = True). The meaning of this condition is as follows:

- If (it is known that) x can be successfully reduced to an arbitrary character, then eval x will produce True and the condition will be satisfied, since True = True is True.

- If (it is known that) x cannot successfully be reduced to a character, then eval x does not terminate and is equal to ⊥ on the semantic level. Therefore, the condition is not satisfied, because ⊥ = True is False.

- Note that eval is defined in such a way that eval x never reduces to False. So, all cases are covered in the previous reasoning.

The same principle can be used for lists, making use of overloading to assume the presence of ‘eval’ on the element type. This leads to the following definition:

```haskell
instance eval [a] | eval a
  where  eval :: ![a] -> Bool | eval a
         eval [x:xs] = eval x && eval xs
         eval [] = True
```

This instance of eval fully evaluates both the list itself and all its elements. It can therefore be used to express the condition that a list must be fully evaluated. Below we give a few examples of the use of eval in properties of functions:

- \(\forall_n[\text{eval } n \rightarrow n < n = \text{False}]\)

- \(\forall_{n, xs}[\text{eval } n \rightarrow \text{take } n \text{ xs } ++ \text{drop } n \text{ xs } = \text{xs}]\)

- \(\forall_{p, xs}[\text{eval } (\text{map } p \text{ xs}) \rightarrow \text{takeWhile } p \text{ xs } ++ \text{dropWhile } p \text{ xs } = \text{xs}]\)

- \(\forall_{x, p, xs}[\text{eval } x \rightarrow \text{eval } xs \rightarrow \text{eval } (\text{map } p \text{ xs}) \rightarrow
  \text{isMember } x \text{ (filter } p \text{ xs}) = \text{isMember } x \text{ xs } \&\& p \text{ x}]\)

The conditions in the examples of Section 5.3.1 can be expressed using ‘eval’. The property of the first example is then expressed as follows (using isMember instead of the mathematical \(\in\)): 
∀f,g,xs[∀x[isMember x xs → eval(g x)] → map (f o g) xs = map f (map g xs)]

To express the definedness condition of the second example of Section 5.3.1 we need another variant of ‘eval’ that does not evaluate its argument fully but that evaluates only the ‘spine’ of the argument. This is given below.

Expressing Spine Evaluation and List Finiteness. Spine evaluation can be expressed easily by means of an ‘eval’ variant. However, if already an instance for full evaluation is given, then a new function must be defined since the type class system allows only one instance per type.

```
 evalSpine :: ![a] -> Bool
 evalSpine [x:xs] = evalSpine xs
 evalSpine [] = True
```

This same function `evalSpine` also expresses finiteness of lists, as when the spine of a list is fully evaluated, the list is evidently finite.

Some valid properties that are defined using `evalSpine`:

- ∀xs[eval (length xs) → evalSpine xs]
- ∀xs[evalSpine xs → evalSpine (reverse xs)]

The second example of Section 5.3.1 can now be reformulated to:

∀xs[evalSpine xs → reverse (reverse xs) = xs]

Properties of ‘eval’. All instances of the class ‘eval’ have to share certain properties. To prove properties of all members of a certain type class, the recently added tool support for general type classes can be used [vvd04]. With this tool, the following properties of ‘eval’ can be stated and proven in SPARKLE.

- ∀x[eval x → x ≠ ⊥]
- ∀x[eval x ≠ False]

## 5.5 Related Work

In [DJ04] Danielsson and Jansson perform a case study in program verification using partial and undefined values. They assume proof rules to be valid for the programming language. They do not use a formal semantics. We expect that our formal semantic approach can be used as a basis to prove their proof rules.

With the purpose of deriving a lazy abstract machine Sestoft [Ses97] has revised Launchbury’s semantics. Launchbury’s semantics require global inspection (which is unwanted for an abstract machine) for preserving the Distinct Names property. When an abstract machine is to be derived from the semantics used in this paper, analogue revisions will be required. As is further pointed out by Sestoft [Ses97] the rules given by Launchbury are not fully lazy. Full laziness can
be achieved by introducing new let-bindings for every maximal free expression [GS01].

In [BKT00], an equivalent extension of Launchbury’s semantics can be found. In this paper, a formal semantics for Glasgow Parallel Haskell is constructed on top of the standard Launchbury’s semantics. Interestingly, not only parallelism is added, but enforced strictness in terms of a seq-construct as well. Furthermore, it is formally shown that this extension is sound. However, no properties are proven that are specific for the seq, such as the relation between ‘lazy’ and ‘strict’ terms. It is possible to translate seq’s to let!s (and vice versa) and shown properties can be compared directly.

Andrew Pitts [Pit98] discusses non-termination issues of logical relations and operational equivalence in the context of the presence of existential types in a strict language. He provides some theory that might also be used to address the problems that arise in a mixed lazy/strict context. That would require a combination of his work and the work of Patricia Johann and Janis Voigtländer [JV04] who use a denotational approach to present some “free” theorems in the presence of HASKELL’s seq.

At Chalmers University of Technology for the language HASKELL a proof assistant Agda [ABB+05] has been developed in the context of the COVER project. As with SPARKLE the language is translated to a core-version on which the proofs are performed. Being geared towards facilitating the ‘average’ functional programmer SPARKLE uses dedicated tactics and proof rules based on standard proof theory. Agda uses constructive type theory on λ-terms enabling independent proof checking. However, in contrast to SPARKLE, Agda has no facilities to prove properties that are related to changed strictness properties.

Another project that aims to integrate programming, properties and validation is PROGRAMATICA (www.cse.ogi.edu/PacSoft/projects/programatica) of the Pacific Software Research Center in Oregon. A wide range of validation techniques for programs written in different languages is intended to be supported. For functional languages they use a logic (P-logic) based on a modal μ-calculus (in which also undefinedness can be expressed). In the PROGRAMATICA project properties are mixed with the HASKELL source. So, reasoning is bound to take place on the more complex syntactical source level instead of on a simpler core-language.

5.6 Conclusions / Future Work

The impact of changes in strictness properties on logical program properties is shown to be quite significant. It is illustrated how program properties can be adapted to reflect these changes. Furthermore, it is explained what the influence of explicit changes in strictness is on the semantics and on the reasoning steps.

We have shown that the special combination of several techniques, that have been made available in the proof assistant SPARKLE to deal with definedness aspects, is well suited to assist the programmer in constructing the required proofs. We do not know of any other proof assistant with such a combined set
of techniques to help dealing with these kinds of proofs.

Future work could be to study the relation of our approach to an approach which only aims to prove partial correctness.
Chapter 6

Marko van Eekelen, Maarten de Mol:

Proving Lazy Folklore
with Mixed Lazy/Strict Semantics

Published in Reflections on Type Theory, λ-calculus, and the Mind:
Essays dedicated to Henk Barendregt on the Occasion of his 60th Birthday.

Abstract. Explicit enforcement of strictness is used by functional programmers for many different purposes. Few functional programmers, however, are aware that explicitly enforcing strictness has serious consequences for (formal) reasoning about their programs. Some vague “folklore” knowledge has emerged concerning the correspondence between lazy and strict evaluation but this is based on experience rather than on rigid proof.

This paper employs a model for formal reasoning with enforced strictness based on John Launchbury’s lazy graph semantics. In this model Launchbury’s semantics are extended with an explicit strict let construct. Examples are given of the use of these semantics in formal proofs. We formally prove some “folklore” properties that are often used in informal reasoning by programmers.

This paper is written at the occasion of the celebration of the 60th anniversary of Henk Barendregt. Henk was the supervisor for the Ph.D. Thesis of Marko van Eekelen. This thesis was just one of the many results of the Dutch Parallel Reduction Machine project in which Henk played a central role.

Quite some time ago, he brought the authors of this paper together knowing that they had common interests in formal proofs for functional programs. This lead to a Master Thesis, the SPARKLE dedicated proof assistant for the language CLEAN, a pile of papers and a Ph.D. manuscript in preparation. Henk taught us how to perform research on a fundamental level without losing sight of the applications of your work.

We are very grateful to him for enlightening us.
6.1 Introduction and motivation

Strictness is a mathematical property of a function. A function \( f \) is strict in its argument if its result is undefined when its argument is undefined, in other words: if \( f \bot = \bot \), where \( \bot \) is the symbol representing the undefined value.

Strictness analysis is used to derive strictness properties for given function definitions in programs written in a functional programming language. If the results of such an analysis are indicated in the program via strictness annotations then of course these annotations do not change the semantics at all. Therefore, it is often recommended to use strictness annotations only when strictness holds mathematically. These annotations are then meant to be used by the compiler for optimisation purposes only.

For the cases of explicit strictness that have the intention to change the semantics, this recommendation is not sensible at all. Although it is seldom mentioned in papers and presentations, such explicit strictness that changes the semantics, is present in almost every lazy programming language (and in almost every program) that is used in real-world examples. In such programs, strictness is used:

- for improving the efficiency of data structures (e.g. strict lists),
- for improving the efficiency of evaluation (e.g. functions that are made strict in some arguments due to strictness analysis or due to the programmers annotations),
- for enforcing the evaluation order in interfacing with the outside world (e.g. an interface to an external call \(^1\) is defined to be strict in order to ensure that the arguments are fully evaluated before the external call is issued).

Language features that are used to denote this strictness include:

- type annotations (in functions: \texttt{CLEAN} and in data structures: \texttt{CLEAN}, \texttt{HASKELL}),
- special data structures (unboxed arrays: \texttt{CLEAN}, \texttt{HASKELL}),
- special primitives (seq: \texttt{HASKELL}),
- special language constructs (let!, #!: \texttt{CLEAN}),
- special tools (strictness analyzers: \texttt{CLEAN}, \texttt{HASKELL}).

Implementers of real-world applications make it their job to know about strictness aspects, because without strictness annotations essential parts of their programs would not work properly. Hence, it is not an option but it is an obligation

\(^1\) (An external call is a call to a function which is defined in a different (possibly imperative) programming language, e.g. \texttt{C}).
Section 6.1: Introduction and motivation

for the compiler to generate code that takes these annotations into account. For reasoning about these annotated programs, however, one tends to forget strictness altogether. Usually, strictness is not taken into account in a formal graph semantics for a programming language. Disregarding strictness can lead to unexpected non-termination when programs are changed by hand or automatically transformed. So, strictness indicated via annotations must form a essential part of the semantics. This may have surprising consequences.

Example of semantic changes due to strictness annotations:
Consider for instance the following CLEAN definition of the function \( f \), which by means of the \(!\)-annotation in the type is made explicitly strict in its first argument. In HASKELL a similar effect can be obtained using an application of \texttt{seq}.

\[
\begin{align*}
\text{f :: !Int} & \rightarrow \text{Int} \\
\text{f x} & = 5 \\
\end{align*}
\]

Without the strictness annotation, the property \( \forall x [f x = 5] \) would hold unconditionally by definition. Now consider the effects of the strictness annotation in the type which makes the function \( f \) strict in its argument. Clearly, the proposition \( f 3 = 5 \) still holds. However, \( f \text{ undefined} = 5 \) does not hold, because \( f \text{ undefined} \) does not terminate due to the enforced evaluation of \texttt{undefined}. Therefore, \( \forall x [f x = 5] \) does not hold unconditionally. The property can be fixed by adding a definedness condition using the special symbol \( \bot \), denoting undefined. This results in \( \forall x [x \neq \bot \rightarrow f x = 5] \), which \textit{does} hold for the annotated function \( f \).

The example above illustrates that the definition of \( f \) cannot \textit{unconditionally} be substituted in all its occurrences. It is only allowed to substitute \( f \) when it is \textbf{known} that its argument \( x \) is \textbf{not undefined}. This has a fundamental impact on the semantics of function application.

The addition of an exclamation mark by a programmer clearly has an effect on the logical properties of functions. The change of a logical property due to addition or removal of strictness can cause problems for program changes made by a programmer. If a programmer is unaware of the logical consequences, this can lead to errors not only at development time but also in the later stage of maintaining the program. A programmer will reason formally or informally about the program and make changes that are consistent with the perceived logical properties.

Changes in logical properties are not only important for the programmer but also for those who work on the compiler. Of course, it is obvious that code has to be generated to accommodate the strictness. Less obvious however, is the consequences adding strictness may have on the correctness of program transformations. There can be far-reaching consequences on various kinds of program transformations.

In other words: the addition or removal of strictness to programs may cause previously valid logical properties to be broken. From a proving point of view
this is a real problem: suppose one has successfully proved a difficult property by means of a sequence of lemmata, then the invalidation of even a single lemma may cause a ripple effect throughout the entire proof! The adaptation to such a ripple effect is both cumbersome and resource-intensive.

Unfortunately, the invalidation of logical properties due to changed strictness annotations is quite common. This invalidation can usually be fixed by the addition of a condition for the strict case (see the example below).

**Example of the addition of a condition:**
\[ \forall f, g \forall x \left[ \text{map} \left( f \circ g \right) x = \text{map} \ f \left( \text{map} \ g \ x \right) \right] \]

**Affected by strictness:**
This property is valid for lazy lists, but invalid for element-strict lists.
Note that no assumptions can be made about the possible strictness of \( f \) or \( g \). Instead, the property must hold for all possible functions \( f \) and \( g \).

**Invalid in the strict case because:**
Suppose \( x = \left[12\right] \), \( 12 = \bot \) and \( f \left( 12 \right) = 7 \).
Then \( \text{map} \left( f \circ g \right) x = \left[7\right] \), both in the lazy and in the strict case.
However, \( \text{map} \ f \left( \text{map} \ g \ x \right) = \left[7\right] \) in the lazy case, but \( \bot \) in the strict case.

**Extra definedness condition for the lazy case:**
The problematic case can be excluded by demanding that for all elements of the list \( g \ x \) can be evaluated successfully.

**Reformulated property for the strict case:**
\[ \forall f, g, x \left[ \forall x \in x \left( x \neq \bot \right) \rightarrow \text{map} \left( f \circ g \right) x = \text{map} \ f \left( \text{map} \ g \ x \right) \right] \]

However, quite surprisingly, it may also be that the invalidation of logical properties due to changed strictness annotations requires the removal of definedness conditions. Below an example is given where the strict case requires the removal of a condition which was required for the lazy case.

**Example of the removal of a condition:**
\[ \forall x \left[ \text{finite} \ x \rightarrow \text{reverse} \left( \text{reverse} \ x \right) = x \right] \]

**Affected by strictness:**
This property is valid both for lazy lists and for spine-strict lists. However, the condition \( \text{finite} \ x \) is satisfied automatically for spine-strict lists. In the spine-strict case, the property can therefore safely be reformulated (or, rather, optimized) by removing the \( \text{finite} \ x \) condition.

**Invalid without finite condition in the lazy case because:**
Suppose \( x = \left[1,1,1,\ldots\right] \).
Then \( \text{reverse} \left( \text{reverse} \ x \right) = \bot \), both in the lazy and in the strict case.
However, \( x = \bot \) in the strict case, while it is unequal to \( \bot \) in the lazy case.
Section 6.2: Mixed lazy/strict graph semantics

Reformulated property for the strict case:
\[ \forall xs[\text{reverse} (\text{reverse} \; xs) = xs] \]

For reasoning with strictness, there is only little theory available so far. In this paper we develop an appropriate mixed denotational and operational semantics for formal reasoning about programs in a mixed lazy/strict context.

6.2 Mixed lazy/strict graph semantics

Since we consider graphs as an essential part of the semantics of a lazy language [BvG87a, van88], we have chosen to extend Launchbury’s graph semantics [Lau93]. Cycles (using recursion), black hole detection, garbage collection and cost of computation can be analyzed formally using these semantics. Launchbury has proven that his operational graph rules are correct and computationally adequate with respect to the corresponding denotational semantics. Informally, correctness means that an expression which operationally reduces to a value will denotationally be equal to that value. Computational adequacy informally means that if the meaning of an expression is defined denotationally it is also defined operationally and vice-versa. Below, we introduce the required preliminaries.

6.2.1 Basic idea of Launchbury’s natural graph semantics

Basically, sharing is represented as let-expressions. In contrast to creating a node for every application, nodes are created only for parts to be shared.

\[
\text{let } x = 3 \times 7 \\
\text{in } x + x
\]

represents the graph on the right:

Graph reduction is formalized by a system of derivation rules. Graph nodes are represented by variable definitions in an environment. A typical graph reduction proof is given below. A linear notation is used. Below the correspondence is illustrated by showing the linear notation on the left and its equivalent graphical notation on the right.

\[
\begin{align*}
\Gamma : e \\
\text{subderivation}_1 \\
\vdots \\
\text{subderivation}_n \\
\Delta : z
\end{align*}
\]

Let

\[
\begin{align*}
\text{subderivation}_1 & \quad \cdots \quad \text{subderivation}_n \\
\Gamma : e \Downarrow \Delta : z
\end{align*}
\]

Each reduction step corresponds to applying a derivation rule (assuming extra rules for numbers and arithmetic; the standard rules are given in Sect. 6.2.4). Below we give the derivation corresponding to the sharing example above. We
leave out normalization and renaming of variables where this cannot cause confusion.

\[
\begin{align*}
\{ \} & : \text{let } x = 3 \ast 7 \text{ in } x + x \\
\{ x \mapsto 3 \ast 7 \} & : x + x \\
\{ x \mapsto 3 \ast 7 \} & : x \\
\{ \} & : 3 \ast 7 \\
\{ \} & : 21 \\
\_\text{Var} & : N_{\text{num}}, N_{\text{num}}, * \\
\{ x \mapsto 21 \} & : 21 \\
\_\text{Var} & : x \mapsto 21 \\
\{ x \mapsto 21 \} & : 21 \\
\{ x \mapsto 21 \} & : 21 \\
+ & : 42 \\
\{ x \mapsto 21 \} & : 42 \\
\_\text{Let} & \\
\end{align*}
\]

6.2.2 Notational conventions.

We will use the following notational conventions:

- \( x, y, v, x_1 \) and \( x_n \) are variables,
- \( e, e', e_1, e_n, f, g \) and \( h \) are expressions,
- \( z \) and \( z' \) are values (i.e. expressions of the form \( \lambda x. e \) and constants, when the language is extended with constants),
- the notation \( \hat{z} \) stands for a renaming (\( \alpha \)-conversion) of a value \( z \) such that all lambda bound and let-bound variables in \( z \) are replaced by fresh ones,
- \( \Gamma, \Delta \) and \( \Theta \) are taken to be heap variables (a heap is assumed to be a set of variable bindings, i.e. pairs of distinct variables and expressions),
- a binding of a variable \( x \) to an expression \( e \) is written as \( x \mapsto e \),
- \( \rho, \rho', \rho_0 \) are environments (an environment is a function from variables to values),
- the judgment \( \Gamma : e \Downarrow \Delta : z \) means that in the context of the heap \( \Gamma \) a term \( e \) reduces to the value \( z \) with the resulting set of bindings \( \Delta \),
- and finally \( \sigma \) and \( \tau \) are taken to be derivation trees for such judgments.
6.2.3 Mixed lazy/strict expressions

We extend the expressions of Launchbury’s system with a non-recursive strict variant of let-expressions.

From a semantic point of view a standard recursive let-expression combined with a strict non-recursive let-expression gives full expressiveness. Due to the possibility of recursion in the standard let, there is no need for adding recursion to the strict let. (Consider for example let \( x = e \ x \ in \ let! \ y = x \ in \ e' \).)

So, we have chosen not to allow recursion in the strict let, although allowing a recursive strict let would not give any semantic problems (as shown in [vd04]). This corresponds to the semantics of the strictness constructs of Haskell [Bir98, Hud00, HPW+92] and Clean [BvvP87, Pv99, Pv01] that do not allow recursion for their strictness constructs.

In strict let-expressions only one variable can be defined in contrast to multiple ones for standard lazy let-expressions. This is natural since the order of evaluation is important. With multiple variables an extra mechanism for specifying their order of evaluation would have to be introduced. With single variable let-expressions an ordering is imposed easily by nesting of let-expressions.

With the extension of these strict let-expressions the class of expressions to consider is given by the following grammar:

\[
x \in Var \quad e \in Exp \quad ::= \quad \lambda x. e \quad |
\quad e \ x \quad |
\quad x \quad |
\quad let \ x_1 = e_1 \ \cdots \ x_n = e_n \ in \ e \quad |
\quad let! \ x_1 = e_1 \ in \ e
\]

As in Launchbury’s semantics we assume that the program under consideration is first translated to a form of lambda terms in which all arguments are variables (expressing sharing explicitly). This is achieved by a normalization procedure which first performs a renaming (\( \alpha \)-conversion) using completely fresh variables ensuring that all bound variables are distinct and then introduces a non-strict let-definition for each argument of each application. The semantics are defined on normalized terms only.

6.2.4 Definition of mixed lazy/strict graph semantics

We extend the basic rules of Launchbury’s natural (operational) semantics (the Lambda, Application, Variable and Let-rules) with a recursive StrictLet rule. This operational StrictLet rule is quite similar to the rule for a normal let, but it adds a condition to enforce the shared evaluation of the expression.

The added let! derivation rule has two requirements. One for the evaluation of \( e_1 \) (expressing that it is required to evaluate it on forehand) and one for the standard lazy evaluation of \( e \). Sharing in the evaluation is achieved by extending the environment \( \Theta \) resulting from the evaluation of \( e_1 \) with \( x_1 \mapsto z_1 \). This environment is then taken as the environment for the evaluation of \( e \).
A striking difference between a standard `let` and a strict `let` is that the environment is extended before the evaluation for a standard `let` and after the evaluation for a strict `let`. This will by itself never give different results since a strict `let` is non-recursive. A strict `let` will behave the same as a standard `let` when \( e_1 \) has a weak head normal form. Otherwise, no derivation will be possible for the strict `let`.

If we would replace `let`'s by standard `let`'s in any expression, the weak head normal form of that expression would not change. However, if we would replace in an expression non-recursive `let`'s by `let`'s, then the weak head normal form of that expression would either stay the same or it would become undefined. This is one of the “folklore” properties that is proven in Sect. 6.3.

**Definition 6.2.41:** (Operational Mixed Lazy/Strict Graph Semantics)

\[
\begin{align*}
\Gamma : \Lambda x.e & \downarrow \Gamma : \Lambda x.e \\
\Gamma : e \downarrow \Delta : \Lambda y.e' & \quad \Delta : e'[x/y] \downarrow \Theta : z \\
\Gamma : e \downarrow \Theta : z \\
\Gamma, x \mapsto e : x \downarrow (\Delta, x \mapsto z) : z \\
\Gamma : let x_1 = e_1 \cdots x_n = e_n in e \downarrow \Delta : z \\
\Gamma : e_1 \downarrow \Theta : z_1 & \quad (\Theta, x_1 \mapsto z_1) : e \downarrow \Delta : z \\
\Gamma : let! x_1 = e_1 in e \downarrow \Delta : z
\end{align*}
\]

Corresponding to the operational semantics given above, we define below the denotational meaning function including the `let!` construct. As in [Lau93] we have a lifted function space ordered in the standard way with least element \( \bot \) following Abramsky and Ong [Abr90, AO93].

We use \( \text{Fn} \) and \( \downarrow \text{Fn} \) as lifting and projection functions. An environment \( \rho \) is a function from variables to values where the domain of values is some domain, containing at least a lifted version of its own function space. We use the following well-defined ordering on environments expressing that larger environments bind more variables but have the same values on the same variables: \( \rho \leq \rho' \) is defined as \( \forall x. [\rho(x) \neq \bot \Rightarrow \rho(x) = \rho'(x)] \). The initial environment, indicated by \( \rho_0 \), is the function that maps all variables to \( \bot \). We use a special semantic function which is continuous on environments \( \{ \} \). It resolves the possible recursion and is defined as: \( \{ x_1 \mapsto e_1, \cdots, x_n \mapsto e_n \}_\rho = \mu \rho'. \rho \cup (x_1 \mapsto \{ e_1 \}_\rho' \cdots x_n \mapsto \{ e_n \}_\rho') \) where \( \mu \) stands for the least fixed point operator and \( \cup \) denotes the least upper bound of two environments. It is important to note that for this definition to make sense the environment must be consistent with the heap (i.e. if they bind the same variable then there must exist an upper bound on the values to which each binds each such variable).
The denotational meaning function extends [Lau93] with meaning for let!-expressions that is given by a case distinction: If the meaning of the expression to be shared is ⊥, then the meaning of the let!-expression as a whole becomes ⊥. For the other case, the definition is similar to the meaning of a let-expression.

**Definition 6.2.4-2: (Denotational Mixed Lazy/Strict Graph Semantics)**
\[
\begin{align*}
\llbracket \lambda x. e \rrbracket_\rho &= F_n (\lambda v. \llbracket e \rrbracket_{\rho \cup (x \mapsto v)}) \\
\llbracket e \mathbf{x} \rrbracket_\rho &= (\llbracket e \rrbracket_\rho) \downarrow F_n (\llbracket \mathbf{x} \rrbracket_\rho) \\
\llbracket \mathbf{x} \rrbracket_\rho &= \rho(x) \\
\llbracket \text{let } x_1 = e_1 \ldots x_n = e_n \text{ in } e \rrbracket_\rho &= \llbracket e \rrbracket_{\rho \cup \{x_1 \mapsto e_1, \ldots, x_n \mapsto e_n\}} \\
\llbracket \text{let! } x_1 = e_1 \text{ in } e \rrbracket_\rho &= \llbracket e \rrbracket_{\rho \cup (x_1 \mapsto [e_1]_\rho)} \quad \text{if } \llbracket e_1 \rrbracket_\rho = \bot
\end{align*}
\]

**6.2.5 Correctness and Computational Adequacy**

Using the definitions above, correctness theorems as in [Lau93] have been established (proofs can be found in [vd04]). The first theorem deals with proper use of names.

**Theorem 6.2.5-1: (Distinct Names)**
If \( \Gamma : e \Downarrow \Delta : z \) and \( \Gamma : e \) is distinctly named (i.e. every binding occurring in \( \Gamma \) and in \( e \) binds a distinct variable which is also distinct from any free variables of \( \Gamma : e \)), then every heap/term pair occurring in the proof of the reduction is also distinctly named.

Theorem 6.2.5-2 essentially states that reductions preserve meaning on terms and that they possibly only change the meaning of heaps by adding new bindings.

**Theorem 6.2.5-2: (Correctness)**
\[
\Gamma : e \Downarrow \Delta : z \Rightarrow \forall \rho. \llbracket \Gamma \rrbracket_\rho \leq \llbracket \Delta \rrbracket_\rho \land \llbracket e \rrbracket_{\Gamma_\rho} = \llbracket z \rrbracket_{\Delta_\rho}
\]

The Computational Adequacy theorem below states that a term with a heap has a valid reduction if and only if they have a non-bottom denotational meaning starting with the initial environment \( \rho_0 \).

**Theorem 6.2.5-3: (Computational Adequacy)**
\[
\llbracket e \rrbracket_{\Gamma_\rho_0} \neq \bot \Leftrightarrow (\exists \Delta, z \text{. } \Gamma : e \Downarrow \Delta : z)
\]

**6.3 Relation to lazy semantics**

Consider the following “folklore” knowledge statements of programmers:

A expressions that are bottom lazily, will also be bottom when we make something strict;
Chapter 6: Proving Lazy Folklore with Mixed Lazy/Strict Semantics

B when strictness is added to an expression that is non-bottom lazily, either the result stays the same or it becomes bottom;

C expressions that are non-bottom using strictness will (after !-removal) also be non-bottom lazily with the same result.

We will turn this “folklore” ABC of using strictness into formal statements. The phrase “is bottom lazily” is taken to mean that when lazy semantics is used the meaning of the expression is ⊥. The phrase “result” indicates of course a partial result: this can be formalized with our operational meaning.

Theorem 6.3:5 will constitute the formal equivalents of these “folklore” statements. In order to formulate that theorem we first need formally define a few operations. For completeness we give below the full definition of the trivial operation of !-removal.

**Definition 6.3:1: (Removal of strictness within expressions)**

The function \( e^{-1} \) is defined on expressions such that \( e^{-1} \) is the expression \( e \) in which every let!-expression is replaced by the corresponding let-expression:

\[
\begin{align*}
(x)^{-1} &= x \\
(\lambda x.e)^{-1} &= \lambda x.(e^{-1}) \\
(e \ x)^{-1} &= (e^{-1})(x^{-1}) \\
(let \ x_1 = e_1 \cdots x_n = e_n \ in \ e)^{-1} &= let \ x_1 = e_1^{-1} \cdots x_n = e_n^{-1} \ in \ e^{-1} \\
(let! \ x_1 = e_1 \ in \ e)^{-1} &= let \ x_1 = e_1^{-1} \ in \ e^{-1}
\end{align*}
\]

**Definition 6.3:2: (Removal of strictness within environments)**

The function \( \Gamma^{-1} \) is defined on environments such that \( \Gamma^{-1} \) is the environment \( \Gamma \) in which in every binding every expression \( e \) is replaced by the corresponding expression \( e^{-1} \):

\[
\begin{align*}
(\Gamma, x \mapsto e)^{-1} &= (\Gamma^{-1}, x \mapsto e^{-1}) \\
\{ \}^{-1} &= \{ \}
\end{align*}
\]

We followed here [Lau93] indicating the empty environment by \( \{ \} \) instead of by \( \emptyset \).

The analogue of !-removal is of course !-addition. We model addition of !’s to an expression \( e \) by creating a set of all those expressions that will be the same as \( e \) after !-removal. In this way we cover all possible ways of adding a !.

**Definition 6.3:3: (Addition of strictness to expressions and environments)**

The function AddStrict is defined on expressions and environments such that AddStrict(\( e \)), respectively AddStrict(\( \Gamma \)), is the set of all expressions, respectively environments, that can be obtained by replacing any number of lets in \( e \), respectively \( \Gamma \), with lets.

\[
\begin{align*}
\text{AddStrict}(e) &= \{ e' \mid (e')^{-1} = e \}; \quad \text{AddStrict}(\Gamma) = \{ \Gamma' \mid (\Gamma')^{-1} = \Gamma \}
\end{align*}
\]
Section 6.3: Relation to lazy semantics

The definition above induces the need of an extension of the semantics of expressions to a semantics of sets of expressions.

Theorem 6.3.4: (Semantics of sets of expressions)
In order to formally reason about the semantics of expressions after the addition of strictness, it must be possible to apply the meaning predicate \[ e \in E \] to sets of expressions and environments, instead of to single expressions and environments. This is realized as follows:

\[ [E]_{[\Gamma]}_{\rho_0} = \{ [e]_{[\Gamma]}_{\rho_0} \mid e \in E, \Gamma \in \Gamma_s \} \]

We are now almost ready to formalize the “folklore” ABC. We will use the standard lazy denotational and operational meanings of [Lau93] and indicate them by \[ \llbracket \] and \[ \rrbracket \]. It goes without saying that \[ \llbracket \] and \[ \rrbracket \] are equivalent to \[ \llbracket \] and \[ \rrbracket \] for expressions and environments that do not contain any strict let expressions.

Theorem 6.3.5: (Formal Folklore ABC)
\( A : \ [e]_{[\Gamma]}_{\rho_0} = \bot \Rightarrow [AddStrict(e)]_{[AddStrict(\Gamma)]_{\rho_0}} = \{ \bot \} \)
\( B : \ [e]_{[\Gamma]}_{\rho_0} = z \Rightarrow [AddStrict(e)]_{[AddStrict(\Gamma)]_{\rho_0}} \subseteq \{ \bot \} \cup AddStrict(z) \)
\( C : \ [e]_{[\Gamma]}_{\rho_0} = z \Rightarrow [e^{-1}]_{[\Gamma^{-1}]}_{\rho_0} = z^{-1} \)

Proof:
The proofs proceed by straightforwardly combining computational adequacy (for lazy and for mixed semantics) and the three additional Theorems 6.3.6, 6.3.7 and 6.3.8 below that capture the essential properties of \( ! \)-removal.

Consider e.g. \( C \): applying computational adequacy on \( [e]_{[\Gamma]}_{\rho_0} = z \) yields \( \Gamma : e \downarrow \Delta : z \), applying Theorem 6.3.7 gives \( \exists \Theta, \Gamma^{-1} : e^{-1} \downarrow^{|\text{lazy}} \Theta : z^{-1} \) and computational adequacy gives the required \( [e^{-1}]_{[\Gamma^{-1}]}_{\rho_0} = z^{-1} \).

Theorem 6.3.6: (Meaning of \( ! \)-removal)
\( [e]_{[\Gamma]}_{\rho_0} \neq \bot \Rightarrow [e^{-1}]_{[\Gamma^{-1}]}_{\rho_0} = [e^{-1}]_{[\Gamma^{-1}]}_{\rho_0} \neq \bot \).

Proof:
Since by definition both for lazy and mixed semantics \( [e]_{[\Gamma]}_{\rho_0} = [e]_{[\Gamma]}_{\rho_0} \), a difference between lazy and mixed meaning can only occur when the mixed semantics is \( \bot \) due to a let!-rule. So, if \( [e]_{[\Gamma]}_{\rho_0} \neq \bot \) then \( [e]_{[\Gamma]}_{\rho_0} = [e^{-1}]_{[\Gamma^{-1}]}_{\rho_0} \) does not hold.

Theorem 6.3.7: (Compare Reduction Strict to Lazy)
\( \Gamma : e \downarrow \Delta : z \Rightarrow \exists \Theta, \Gamma^{-1} : e^{-1} \downarrow^{|\text{lazy}} \Theta : z^{-1} \wedge [z^{-1}]_{[\Theta]}_{\rho_0} = [z]_{[\Delta]}_{\rho_0} \)
Proof:
Assume we have $\Gamma : e \Downarrow \Delta : z$ with derivation tree $\sigma$. Compare the operational rules for `let`! and `let`. The condition on the right of the `let`! rule has (up to `!`-removal) the very same expressions as the `let`! rule but a different environment. This environment captures the 'extra' non-lazy reductions that are induced by the `let`-rule. Clearly, there is an environment $\Theta$ such that $\Gamma \!:\! e \Downarrow \Theta : z$ and by lazy correctness and computational adequacy

$$\llbracket e \rrbracket _{\Gamma \!:\! \Theta} \rho_0 = \llbracket z \rrbracket _{\Delta \!:\! \Theta} \rho_0 \neq \bot$$

By mixed correctness and Theorem 6.3 it follows that

$$\llbracket e \rrbracket _\Gamma \rho_0 = \llbracket z \rrbracket _\Delta \rho_0 = \llbracket e \rrbracket _{\Gamma \!:\! \Theta} \rho_0 = \llbracket z \rrbracket _\Theta \rho_0 \neq \bot$$

Theorem 6.3:

(Compare Reduction lazy to strict)

$$\Gamma^{-1} : e^{-1} \Downarrow^{\text{lazy}} \Theta : z^{-1} \Rightarrow \llbracket e \rrbracket _\Gamma \rho_0 = \bot$$

$$\exists \Theta. \Gamma : e \Downarrow \Theta : z \land \llbracket z \rrbracket _\Theta \rho_0 = \llbracket z \rrbracket _\Delta \rho_0$$

Proof:
Assume that $\llbracket e \rrbracket _\Gamma \rho_0 \neq \bot$, then by mixed computational adequacy $\exists \Theta. \Gamma : e \Downarrow \Theta : z$, and by mixed correctness, Theorem 6.3 and lazy correctness

$$\llbracket z \rrbracket _\Theta \rho_0 = \llbracket z \rrbracket _\Delta \rho_0 = \llbracket e \rrbracket _{\Gamma \!:\! \Theta} \rho_0 = \llbracket z \rrbracket _\Theta \rho_0$$

6.4 Example proofs with mixed semantics

With a small example we will show how proofs can be made using mixed semantics; the proof shows formally that with mixed semantics it is possible to distinguish operationally between terms that were indistinguishable lazily. The lazy semantics as defined by Launchbury [Lau93] makes it possible to yield $\lambda x.\Omega$ ($\Omega$ is defined below) and $\Omega$ as different results. However, in such lazy semantics it is not possible to define a function $f$ that produces a different observational result depending on which one is given as an argument [AO93]. We say that two terms “produce a different observational result” if at least one term produces a basic value and the other one either produces a different basic value or $\bot$. This means that in lazy natural semantics $\lambda x.\Omega$ and $\Omega$ belong to a single equivalence class of which the members cannot be distinguished observationally by the programmer.

With mixed semantics a definition for such a distinguishing function $f$ is given below. The result of $f$ on $\lambda x.\Omega$ will be 42 and the result of $f$ on $\Omega$ will be $\bot$. Note that it is not possible to return anything else than $\bot$ in the $\Omega$ case.

Theorem 6.4.1:
($\lambda x.\Omega$ and $\Omega$ can be distinguished)
Let $\Omega$ denote $(\lambda x.xx)(\lambda x.xx)$, and $f$ denote $\lambda x.(\text{let}! y = x \text{ in } 42)$.
Then (A) $\not\Delta z. \{ : f \Omega \Downarrow \Delta : z$
and (B) $\exists \Delta. \{ : f (\lambda x.\Omega) \Downarrow \Delta : 42$ hold.
Section 6.4: Example proofs with mixed semantics

Proof(A):
We have to prove that it is impossible to construct a finite derivation according to the operational semantics. Applying Theorem 6.2.5:3, the computational adequacy theorem, it is sufficient to show that the denotational meaning of $f \Omega$ is undefined. The proof is as follows using the denotational semantics:

$$
\llbracket f \Omega \rrbracket_{\rho_0} = \llbracket (\lambda x. let! y = x in 42)(\Omega) \rrbracket_{\rho_0} = (\llbracket (\lambda x. let! y = x in 42) \rrbracket_{\rho_0}) \downarrow_{Fn} (\llbracket \Omega \rrbracket_{\rho_0}) = (\llbracket (\lambda v. let! y = x in 42) \rrbracket_{\rho_0 \cup (x \rightarrow \Omega_0)}) \downarrow_{Fn} (\llbracket \Omega \rrbracket_{\rho_0}) = \llbracket let! y = x in 42 \rrbracket_{\rho_0 \cup (x \rightarrow \Omega_0)} = \bot \text{ since } \llbracket x \rrbracket_{\rho_0 \cup (x \rightarrow \Omega_0)} = (\rho_0 \cup (x \mapsto \llbracket \Omega \rrbracket_{\rho_0}))(x) = \llbracket \Omega \rrbracket_{\rho_0} = \bot \text{ since for } \Omega \text{ no derivation can be made.}
$$

Proof(B):
This is proven by a derivation in the operational semantics written down as in Sect 6.2.1. To work with numerals we assume the availability of a standard reduction rule (\textit{Num}) that states that each numeral reduces to itself.

\[
\begin{cases}
\{ \} : f (\lambda x. \Omega) \\
\{ \} : (\lambda x. let! y = x in 42) (\lambda x. \Omega) \\
\{ \} : (\lambda x. let! y = x in 42) \\
\{ \} : (\lambda x. let! y = x in 42) \\
\{ \} : (let! y = x in 42) [\lambda x. \Omega/x] \\
\{ \} : let! y = \lambda x. \Omega in 42 \\
\{ \} : \lambda x. \Omega \\
\{ \} : \lambda x. \Omega \\
\{ y \mapsto \lambda x. \Omega \} : 42 \\
\{ y \mapsto \lambda x. \Omega \} : 42 \\
\{ y \mapsto \lambda x. \Omega \} : 42 \\
\{ y \mapsto \lambda x. \Omega \} : 42 \\
\{ y \mapsto \lambda x. \Omega \} : 42 \\
\end{cases}
\]
6.5 Related work

In [DJ04] a case study is done in program verification using partial and undefined values. They assume proof rules to be valid for the programming language. The connection with our approach could be that our formal semantic approach can be used as a basis to prove their proof rules.

With the purpose of deriving a lazy abstract machine Sestoft [Ses97] has revised Launchbury’s semantics. Launchbury’s semantics require global inspection (which is unwanted for an abstract machine) for preserving the Distinct Names property. When an abstract machine is to be derived from our mixed semantics, analogue revisions will be required. As is further pointed out by Sestoft [Ses97] the rules given by Launchbury are not fully lazy. Full laziness can be achieved by introducing new let-bindings for every maximal free expression [GS01].

Another extension of Launchbury’s semantics is given by Baker-Finch, King and Trinder in [BKT00]. They construct a formal semantics for Glasgow Parallel Haskell on top of the standard Launchbury’s semantics. Their semantics that are developed for dealing with parallelism, are equivalent to our semantics that are developed independently for dealing with strictness. Equivalence can be shown easily by translating \texttt{seq} into let!-expressions. They do not prove properties expressing relations between ‘lazy’ and ‘strict’ terms.

As part of the COVER project [CDHS01], it is argued in [DHJG06] that “loose reasoning” is “morally correct”, i.e. that if, under the assumption that every subexpression is strict and terminating, you can prove your theorem than the theorem will also hold in the lazy case under certain conditions. However, the conditions that are found in this way, may be too restrictive for the lazy case. The Nijmegen proof assistant SPARKLE [dvP02] has several facilities for defining and proving the proper definedness conditions [vd06].

6.6 Conclusions

We have extended Launchbury’s lazy graph semantics with a construct for explicit strictness. We have explored what happens when strictness is added or removed within such mixed lazy/strict graph semantics. Correspondences and differences between lazy and mixed semantics have been established by studying the effects of removal and addition of strictness. Our results formalize the common “folklore” knowledge about the use of explicit strictness in a lazy context.

Mixed lazy/strict graph semantics differs significantly from lazy graph semantics. It is possible to write expressions that with mixed semantics distinguish between particular terms that have different lazy semantics while these terms can not be distinguished by an expression within that lazy semantics. We have proven this formally.

Acknowledgement

We would like to thank the anonymous referee(s) of (an earlier version of) this paper for the(ir) valuable comments.
Chapter 7

Maarten de Mol, Marko van Eckelen and Rinus Plasmeijer:

A Single-Step Term-Graph Reduction System for Proof Assistants

Presented at AGTIVE'07, published in LNCS proceedings volume 5088.

Abstract. In this paper, we will define a custom term-graph reduction system for a simplified lazy functional language. Our custom system is geared towards flexibility, which is accomplished by leaving the choice of redex free and by making use of single-step reduction. It is therefore more suited for formal reasoning than the well-established standard reduction systems, which usually fix a single redex and realize multi-step reduction only. We will show that our custom system is correct with respect to the standard systems, by proving that it is confluent and allows standard lazy functional evaluation as a possible reduction path.

Our reduction system is used in the foundation of Sparkle. Sparkle is the dedicated proof assistant for Clean, a lazy functional programming language based on term-graph rewriting. An important reasoning step in Sparkle is the replacement of an expression with one of its reducts. The flexibility of our underlying reduction mechanism ensures that as many reduction options as possible are available for this reasoning step, which improves the ease of reasoning.

Because our reduction system is based on a simplified lazy functional language, our results can be applied to any other functional language based on term-graph rewriting as well.

7.1 Introduction

Clean [Pv98] and Haskell [HPW+92] are lazy functional programming languages that have a semantics based on term-graph rewriting. Due to their mathematical nature, functional programming languages are well suited for formal methods. Industry is beginning to acknowledge the importance of formal methods for verifying safety-critical components of both hardware and software.
(for instance, see [BHLv06]). Consequently, functional languages are being used increasingly often in industrial practice (for instance, see [Mor04]).

The distribution of Clean was extended with the dedicated proof assistant Sparkle [dvP08a, dvP02] in 2001. A proof assistant is a tool that supports formal reasoning about programs. Since its introduction, Sparkle has been used in practice for various purposes. It has been used for proving properties of I/O-programs by Dowse [DBv05] and Butterfield [BS01]. An extension for dealing with temporal properties has been proposed for it by Tejfel, Horváth and Koszik [THK06, HKT03]. It has been used in education at the Radboud University of Nijmegen. Furthermore, support for class-generic properties has been added to it by van Kesteren [vvd04].

A very important reasoning step in the library of Sparkle is ‘Reduce’, which makes use of the operational semantics of Clean to replace an expression with one of its reducts. The usefulness of ‘Reduce’ depends on the reduction options that are made available by the underlying formal reduction system, which must therefore be sufficiently flexible. Of course, it also has to support lazy evaluation, graphs and sharing. Normally, the natural choice would be the well-established system of Launchbury [Lau93]. This system, however, is geared towards evaluation: it uses multi-step reduction and fixes a single redex. Therefore, both partial and inner reductions are not elements of its reduction relation and are not provided as reduction options, which is undesirable for formal reasoning.

In this paper, we will define a custom and flexible reduction system for a lazy functional language. Our system is based on Launchbury’s, but uses single-step reduction and leaves the choice of redex free. The formalized reduction relation therefore contains partial and inner reducts as well, which makes our system suited for formal reasoning. We will show that our system is confluent and that the standard lazy functional reduction path is allowed by it. This ensures that our system behaves correctly with respect to Launchbury’s system.

An extended version of our reduction system is used in the full mathematical foundation of Sparkle, which is described in [dvP07a]. There are two main differences between this paper and the extended version. Firstly, this paper uses a simplified generic expression language, which makes our reduction system applicable to other functional languages as well. Secondly, this paper improves on the handling of sharing, by explicitly enforcing it for function arguments beforehand and by not making use of external environments. This makes unsharing in our system much easier, and allows for local confluence as well.

This paper is structured as follows. In Section 7.2, we examine the desired level of flexibility. We introduce our expression language in Section 7.3, and describe our reduction system in Section 7.4. We show how to express standard reduction paths in our system in Section 7.5, and we prove confluence of our system in Section 7.6. Finally, we discuss related work in Section 7.7 and draw conclusions in Section 7.8.
7.2 Desired level of flexibility

Replacing expressions with reducts is a very natural and intuitive reasoning step. The flexibility of the underlying reduction system determines the number of reduction options that are available for this step. In principle, having more reduction options increases the power of reasoning. This reasoning power is only useful, however, if the options can intuitively be recognized as reducts.

In the introduction, two factors were mentioned that influence flexibility: the granularity of the reduction relation (single-step vs multi-step), and the freedom of choice of redex (fixed redex vs free redex). In the following sections, we will examine the precise effect of these factors on formal reasoning more closely.

7.2.1 Granularity of reduction steps

On the intuitive level, reduction is mainly considered to be defined by means of the reduction steps, and only secondary by means of the overarching reduction relation. On the reasoning level, the reduction options that are offered to the proof builder should therefore include the results of partial reductions as well. To formalize this, a single-step reduction system is needed, in which the reduction relation is defined in terms of single applications of individual reduction steps.

**Example:** (proof that requires intermediate reducts)

Assume that the following property has been proved:

\[ \forall b \left[ \text{not} \left( \text{not} \ b \right) = b \right]. \]

Using this property, assume that we now want to prove the following:

\[ \text{not} \left( \text{id} \ (\text{not} \ X) \right) = X \] (where \( X \) is some complex computation)

On the intuitive level, this is a trivial proof: simply replace ‘\( \text{id} \ (\text{not} \ X) \)’ with ‘\( \text{not} \ X \)’, and then apply the assumed property. QED.

This intuitive proof, however, relies on single-step inner reduction. If no inner reduction is available, then ‘\( \text{id} \ (\text{not} \ X) \)’ cannot be selected as redex; if no single-step reduction is available, then the reduction of ‘\( \text{id} \ (\text{not} \ X) \)’ cannot be stopped after the first step and ‘\( X \)’ will be evaluated unnecessarily.

7.2.2 Choice of redex

Because lazy functional languages are referentially transparent, it is always safe to apply reduction to an inner redex. Formally, however, referential transparency has to be proved too. This proof can be constructed in two different ways:

1. Start with a reduction system that allows leftmost-outermost reduction only. Define semantic equality on top, and prove that it is referentially transparent.

2. Start with a reduction system that allows arbitrary redexes to be reduced. Prove that this system is confluent, define a semantic equality on top of it, and let referential transparency follow from the already shown confluence.
Because semantic equality needs to cope with infinite reductions (bisimulation), the second approach is much easier to carry out. Therefore, in this paper we will allow the redex to be chosen freely, and we will explicitly prove confluence.

7.3 The expression language

Our expression language models the core of an arbitrary lazy functional language. The basic components of our language are variables, functions, applications and let expressions. Without loss of generality, we assume that each function symbol has a fixed arity, and we abstract from constructors and cases, which can be added without difficulties. We represent function definitions in a constant external environment, and do not use lambda expressions. We consider sharing to be a basic component of any lazy functional language.

Notations: (variables, function symbols and lists)

Let $V$ denote the set of variable names, $F$ the set of function symbols, and $\text{Arity}: F \rightarrow n$ the function that obtains the arity of a function symbol.

Let $\text{Vars}$ and $\text{Bound}$ denote the functions that obtain the free and bound variables of an expression respectively. Let ‘[‘ and ‘]’ denote lists, $\#xs$ the length of a list $xs$, and $xs!i$ the $i$-th element of $xs$, if it exists. Let $\text{Unq}(xs)$ denote that all elements in $xs$ occur only once.

Notation: (construction of sets)

In this paper, sets will be denoted by means of $\{O(x_i) \mid x_i \in X_i \mid P(x_i)\}$, in which $O(x_i)$ describes the syntactical shape of the set elements, $x_i \in X_i$ describes the domains of the variable placeholders, and $P(x_i)$ describes the condition that all elements of the set must adhere to.

Definition 7.3.1: (set of expressions)

The set $E$ of expressions is defined recursively by:

$E = \{\text{var } x \mid x \in V\} \\
\cup \{\text{fun } f \text{ on } xs \mid f \in F, xs \in \langle V \rangle \mid \text{Arity}(f) \geq \#xs\} \\
\cup \{\text{app } e \text{ to } x \mid e \in E, x \in V\} \\
\cup \{\text{let } xs = es \text{ in } e \mid xs \in \langle V \rangle, es \in \langle E \rangle, e \in E \mid \#xs = \#es \land \text{Unq}(xs)\}$

Example: (term-graph expression with cycles)

Our representation of expressions allows cycles to be represented by means of recursive lets. For instance, assuming the availability of a function symbol $F$ (arity 2) and a variable $x$, and assuming that the leftmost occurrence of $F$ is the root, the following graph and expression are equivalent:

```plaintext
let ⟨a, b, c⟩ = (fun F on ⟨var c, var b⟩, fun F on ⟨var c, var a⟩, var x)
in (var a)
```
**Assumption 7.3.2:** *(programs)*  
Assume the function $\text{Body} : \langle V \rangle \times \langle V \rangle \times F \rightarrow E$, which models the program context and binds function symbols to fresh copies of their function bodies. Assume that $\text{Body}(xs, ys, f)$ denotes the body of $f$ in which the arguments have been replaced by $xs$ and the bound variables have been replaced by $ys$.

**Example:** *(use of the program function)*  
Assume that the function $f$ is defined as follows:  
$$ f \ x = \text{let } \ y = x+x \ \text{in } y+y $$  
Formalized by means of the $\text{Body}$-function, this becomes:  
$$ \text{Body}(E, z, f) = (\text{let } z = E+E \ \text{in } z+z) $$  
The $\text{Body}$-function therefore expands a function on given arguments, using the argument variables to create a fresh instantiation of the function body.

Note that there are two different alternatives for application in our language. The ‘fun’-alternative is used for lifting function symbols to the expression level, and for gradually collecting function arguments. The ‘app’-alternative is used for applications of expressions that still have to be reduced to function symbols.

Note further that the arguments of both kinds of applications must always be variables. Because of this convention (which we borrow from [Lau93]), expressions need to be converted before they can be represented in our language. Each application that occurs in the expression has to be transformed as follows:

$$ \text{Transform}(\text{fun } f \text{ on } es) = \text{let } xs = es \ \text{in } (\text{fun } f \text{ on } xs) $$  
$$ \text{Transform}(\text{app } e_1 \text{ to } e_2) = \text{let } \langle x \rangle = \langle e_2 \rangle \ \text{in } (\text{app } e_1 \text{ to } x) $$  

This transformation has to be carried out recursively, and the variables that are created must be fresh. We do not lose expressiveness, because each expression can be transformed this way. The advantage of this convention is that function arguments can be duplicated without loss of sharing. This makes our function expansion rule much easier, as it is no longer necessary to create fresh variables (for sharing function arguments) within the rule itself.

Note that the transformation can never be reversed, because the result would be an expression that cannot be represented in our system. This is not a problem, because reduction never requires the transformation to be reversed.

### 7.4 Reduction System

In the following sections, we will introduce our reduction system step-by-step. First, we introduce our approach to handling sharing in Section 7.4.1. Then, we describe the individual rules of our system in Sections 7.4.2(applications), 7.4.3(lets) and 7.4.4(unsharing). By combining individual rules, head reduction is formalized in Section 7.4.5. Finally, locations are introduced in Section 7.4.6, and they are used to upgrade head reduction to inner reduction in Section 7.4.7.
7.4.1 Graphs as self-contained expressions

Sharing is handled in our reduction system in a way that is not standard. We do not use an external environment for storing graph nodes, and we do not have a reduction rule that removes let bindings from an expression and transfers them to an external environment. Instead, we store graph nodes within the expression by means of lets and use a let-lifting mechanism.

The goal of our method is get rid of external environments completely, which normally have to be dragged along continuously. By maintaining graph nodes internally, expressions become self-contained; they can be reduced and given a meaning without pairing them to an external object. This makes handling expressions more transparent, and makes subsequent definitions and proofs easier.

The disadvantage of our method is that additional functionality is needed for maintaining let definitions internally. Two tasks have to be performed:

- If reduction requires a subexpression at a specific location to be in a certain form, then it must be possible to remove a leading let from that location.
  
  Example: ‘app (let ⟨x⟩=⟨e⟩ in (fun f on ⟨x⟩)) to y’. (arity of f is 2)
  
  Reduction should first join the outer app and the inner fun, adding y to the argument list ⟨x⟩. Then, reduction should expand f.
  
  Unfortunately, the let expression in the middle prevents the contraction rule from matching immediately. Normally, this would not be a problem, because reduction would be able to move the inner let to an external environment. In our case, the inner let cannot be removed, and another solution is needed.

- If reduction requires a variable to be unshared, then an explicit link has to be created to the corresponding let binding.
  
  Example: ‘let ⟨x⟩=⟨e⟩ in (app (var x) to y)’. (assume that e is in nf)
  
  Reduction should now replace the inner ‘var x’ with e. This requires the inner reduction of ‘var x’ to know about the external binding of x to e.
  
  Normally, reduction of the expression as a whole would introduce x = e into the external environment, by means of which the information would be made available. Because we do not use external environments, we have to find another way of passing down this information.

Fortunately, solutions to the issues above can be realized easily, see Sections 7.4.3 and 7.4.4 respectively. Overall, our reduction system remains very simple.

7.4.2 The reduction rules for applications

In our system, applications are contracted from initial sequences of app-nodes into fun-nodes. When sufficient arguments have been collected, the function is expanded. This process can be realized by the following two reduction rules:

- The collect-rule accumulates function arguments into a central fun-node by removing them from surrounding app-nodes. This process is repeated
until the fun-node is filled and contains as many arguments as its arity describes.

- The expand-rule replaces a filled fun-node with (a fresh copy) of the body of the function (obtained with Body, see Assumption 7.3.2). Additional context information is required in the form of a list of fresh variables, which are used as instantiation for the bound variables of the body.

In this paper, we will formalize reduction by means of deterministic functions, because this makes proving confluence much easier. If additional information is required to accomplish deterministic behavior, then it is assumed to be available by means of input arguments. In the later stages of the formalization of reduction, it will be described how this information is obtained.

The reduction rules collect and expand are formalized as follows:

Definition 7.4.2.1: (the realization of the collect-rule)

The function Collect : \( \mathcal{E} \rightarrow \mathcal{E} \) is defined by:

\[
\text{Collect}(e) = \begin{cases} 
\text{fun } f \text{ on } \langle xs : x \rangle & \text{if } e = (\text{app } (\text{fun } f \text{ on } xs) \text{ to } x) \\
\text{e} & \text{otherwise}
\end{cases}
\]

Definition 7.4.2.2: (the realization of the expand-rule)

The function Expand : \( (\mathcal{V}) \times \mathcal{E} \rightarrow \mathcal{E} \) is defined by:

\[
\text{Expand}(ys, e) = \begin{cases} 
\text{Body}(xs, ys, f) & \text{if } e = (\text{fun } f \text{ on } xs) \\
\text{e} & \text{otherwise}
\end{cases}
\]

Note that, as a consequence of allowing only variables at argument positions, the reduction rules for function application do not have to take sharing into account in any way. Instead, sharing is preserved automatically.

7.4.3 The reduction rules for let lifting

For the administration of sharing, our reduction system maintains lets within expressions, instead of moving them into an external environment. This means that lets may get in the way of reduction: when a subexpression has to be brought into a certain form, it is possible that a let is created on its outer level. For reduction to continue, it must be possible to remove this hindering let.

Our basic idea is to move lets upwards until they are no longer in the way. This approach works, because: (1) lets at the outermost level can never be in the way; and (2) upward moves can be achieved easily at all relevant locations. We will call the upward move of a let a let lift; our alternative for external environments is therefore the process of let lifting.

In our system, there are two places where a let must be lifted upwards:

- On the left-hand-side of an application.
Chapter 7: A Single-Step Term-Graph Reduction System

The expression on the left-hand-side of an \textbf{app}-node must be reduced to a \textbf{fun}-node in order for reduction to continue by means of an application of the \textit{collect}-rule. If a let expression appears at the outermost level of the left-hand-side of an application, it therefore has to be moved out of the way.

- \textit{On the right-hand-side of a let binding.}

An important step in the functional reduction strategy is the unsharing of a stored let binding. This is only allowed if the binding is in a certain form; in particular, it may not be a let expression. If a let expression appears at the outermost level of the right-hand-side of a let binding, it therefore has to be moved out of the way.

The two reduction rules that perform let lifting are \texttt{lift app} and \texttt{lift let}. They are formalized by means of the functions \texttt{LiftApp} and \texttt{LiftLet}. The function \texttt{LiftApp} does not require additional context information, but \texttt{LiftLet} requires the index of the let binding to be lifted for reasons of disambiguation.

**Definition 7.4.3:1** (the realization of the \texttt{lift-app}-rule)

The function \texttt{LiftApp} : \(E \rightarrow E\) is defined by:

\[
\text{LiftApp}(e) = \begin{cases} 
\text{let } xs = es \text{ in } (\text{app } e'' \text{ to } x) & \text{if } e = (\text{app } e' \text{ to } x) \\
\text{otherwise} & \text{otherwise}
\end{cases} \\
\wedge e' = (\text{let } xs = es \text{ in } e'')
\]

**Definition 7.4.3:2** (the realization of the \texttt{lift-let}-rule)

The function \texttt{LiftLet} : \(n \times E \rightarrow E\) is defined by:

\[
\text{LiftLet}(i, e) = \begin{cases} 
\text{let } \langle xs_1 : y : x : x_2 \rangle = \langle as_1 : bs : b : as_2 \rangle \text{ in } a & \text{if } e = (\text{let } \langle xs_1 : x_i : x_2 \rangle = \langle as_1 : a_i : as_2 \rangle \text{ in } a) \\
\#xs_1 = \#as_1 = i - 1 & \text{and } a_i = (\text{let } y = bs \text{ in } b) \\
\text{otherwise} & \text{otherwise}
\end{cases}
\]

Note that \texttt{LiftLet} joins two let expressions into a single new one. The argument \(i\) determines which inner let should be lifted. It is required, because multiple inner bindings may be a let itself. The bindings of the inner let are inserted in the outer let just before the original binding. This ensures that the order in which inner lets are lifted does not matter; the result will always be the same.

**Example:** (example of the \texttt{lift-app}-rule)

In Section 7.4.1, the following example of a hindering let was given:

\['\text{app } (\text{let } \langle x \rangle = \langle e \rangle \text{ in } (\text{fun } f \text{ on } \langle x \rangle)) \text{ to } y'.\] (arity of \(f\) is 2)

By applying \texttt{LiftApp}, this expression can now be transformed to:

\['\text{let } \langle x \rangle = \langle e \rangle \text{ in } (\text{app } (\text{fun } f \text{ on } \langle x \rangle)) \text{ to } y'.\]

Reduction can now continue on the inner let by means of a \textit{collect}.

**Example:** (example of the \texttt{lift-let}-rule)

In the following expression, both the inner lets can be lifted:
The reduction rule for unsharing

The last remaining task for which a reduction rule has to be defined is the task of unsharing. This is the process of replacing variables with the expressions that they are associated with by means of a let binding. We will model one single unshare at a time. Note that cyclic let definitions are allowed; therefore, the process of repeated unsharing does not always terminate. A single unshare, however, always terminates.

Because efficiency is important even when building proofs, we do not allow duplication of unfinished computations. Therefore, an expression may only be unshared if it can statically be determined that it does not contain any redexes. In our language, this is only the case for partial applications. Chains of variables $(x = y, y = \ldots)$ cannot be unshared immediately. Instead, the final binding has to be reduced to a partial application first, after which the chain can be collapsed.

The rule for unsharing is called unshare, and its function is Unshare. The function can only be applied to a variable, and takes the binding as additional input. It is assumed that the binding occurs in the context of the redex.

**Definition 7.4.4: (the realization of the unshare-rule)**

The function $Unshare : V \times E \times E \to E$ is defined by:

$$Unshare(x, u, e) = \begin{cases} u & \text{if } e = (\text{var } x) \land u = (\text{fun } f \text{ on } xs) \\ & \land \text{Arity}(f) < \#xs \\ e & \text{otherwise} \end{cases}$$

Note that this unshare can replace a variable $x$ with any expression $u$ that it is given as additional argument. On this level, there is no verification that $x = u$ actually appears in the context of the redex. This verification is performed later, on the level of inner reduction (see Section 7.4.7).

**7.4.5 Head reduction**

Head reduction is the combination of the five reduction functions defined in the previous sections. It operates on a rule selector and an expression. Based on the rule selector, one of the five reduction functions is selected, which is then applied to the expression. A rule selector is an artificial identifier that denotes one of the five reduction rules. For simplicity, we incorporate the additional input arguments of the individual rules into the rule selectors defined below:
Definition 7.4.5:1: (set of rule selectors)

The set \( \mathcal{R} \) of rule selectors is defined by:
\[
\mathcal{R} = \{ \text{collect, lift app} \} \\
\cup \{ \text{expand } xs \mid xs \in \mathcal{V} \} \\
\cup \{ \text{lift bind } i \mid i \in n \} \\
\cup \{ \text{unshare } x \text{ to } u \mid x \in \mathcal{V}, u \in \mathcal{E} \}
\]

The head reduction function is simply a case distinction on the rule selector:

Definition 7.4.5:2: (head reduction)

The function \( \text{HeadReduce} : \mathcal{R} \times \mathcal{E} \to \mathcal{E} \) is defined by:
\[
\text{HeadReduce}(\text{collect}, e) = \text{Collect}(e) \\
\text{HeadReduce}(\text{expand } xs, e) = \text{Expand}(xs, e) \\
\text{HeadReduce}(\text{lift app}, e) = \text{LiftApp}(e) \\
\text{HeadReduce}(\text{lift bind } i, e) = \text{LiftLet}(i, e) \\
\text{HeadReduce}(\text{unshare } x \text{ to } u, e) = \text{Unshare}(x, u, e)
\]

A summary of the total system of reduction rules is given in Table 7.1.

<table>
<thead>
<tr>
<th>name</th>
<th>rule</th>
<th>conditions</th>
</tr>
</thead>
<tbody>
<tr>
<td>collect</td>
<td>( \text{app (fun } f \text{ on } xs \text{) to } x ) ( \text{fun } f \text{ on } (xs:x) )</td>
<td>( \text{Arity}(f) &gt; #xs )</td>
</tr>
<tr>
<td>expand ( ys )</td>
<td>( \text{fun } f \text{ on } xs ) ( \text{Body}(xs, ys, f) )</td>
<td>( \text{Arity}(f) = #xs )</td>
</tr>
<tr>
<td>lift app</td>
<td>( \text{app (let } xs=es \text{ in } e \text{) to } x ) ( \text{let } xs=es \text{ in } (\text{app } e \text{ to } x) )</td>
<td>–</td>
</tr>
<tr>
<td>lift bind ( i )</td>
<td>( \text{let } (x_1 \ldots x_n) = (e_1 \ldots e_n) \text{ in } e ) ( \text{let } (x_1 \ldots x_{i-1} : ys : x_{i+1} \ldots x_n) = (e_1 \ldots e_{i-1} : as : a : a_{i+1} \ldots a_n) \text{ in } e )</td>
<td>( 1 \leq i \leq n, e_i = (\text{let } ys=as \text{ in } a) )</td>
</tr>
<tr>
<td>unshare ( x \text{ to } u )</td>
<td>( \text{var } x ) ( \text{u} )</td>
<td>( u = (\text{fun } f \text{ on } xs), \text{Arity}(f) &lt; #xs )</td>
</tr>
</tbody>
</table>

Table 7.1: The reduction system as a whole

7.4.6 Locations

All the reduction functions that have been defined so far can only be applied to the head of an expression. In order to lift these function to inner reduction, we will use the concept of locations. A location is an artificial identifier that points to a specific subexpression within a compound expression. The basic operations on locations are \( \text{Get} \) and \( \text{Set} \). For a full formalization of locations we refer to the technical report [dvP07b]. Here, we introduce locations informally only:

Notation 7.4.6.1: (locations and operations on locations)

Let \( \mathcal{L} \) denote the set of available locations, \( \text{Get} : \mathcal{L} \times \mathcal{E} \leftarrow \mathcal{E} \) the function
that gets the subexpression from an indicated location, and \( \text{Set} : \mathcal{L} \times \mathcal{E} \times \mathcal{E} \rightarrow \mathcal{E} \) the function that sets the subexpression at an indicated location.

Note that both \( \text{Get} \) and \( \text{Set} \) are partial functions; they fail when the location is not valid within the indicated expression.

### 7.4.7 Inner reduction

The final step in defining our custom reduction system is the upgrade of head reduction to inner reduction, which allows reduction to take place on an arbitrary redex. Inner reduction is represented by a function that operates on a location, a rule selector and an expression. It selects the redex at the indicated location, and applies head reduction to it using the given rule selector as argument.

Inner reduction performs partial verification of the incoming rule selector as well. It checks two conditions, namely: (1) whether the variables of an \texttt{expand} are indeed fresh with respect to the expression that is reduced; and (2) whether the binding of an \texttt{unshare} is indeed available in the context of the redex. These conditions are checked using a combination of the redex location and the expression as a whole. The other reduction functions operate on the redex alone, and can therefore not perform these verifications themselves.

The verification of the freshness of an \texttt{expand}-rule is formalized by means of the relation \( \text{Fresh} \). It simply extracts the variables from the rule and checks whether there is an overlap with the bound variables of the expression.

**Definition 7.4.7.1: (verification of an \texttt{expand}-rule)**

The relation \( \text{Fresh} \subseteq \mathcal{R} \times \mathcal{E} \) is defined by:

\[
\text{Fresh}(r,e) \iff \forall x \in \mathcal{V} \exists s \in \langle \mathcal{V} \rangle [r = (\text{expand} \ s) \Rightarrow \neg \exists x \in \mathcal{V} [x \in s \land x \in \text{Bound}(e)]]
\]

The verification of an \texttt{unshare}-rule is formalized in two steps. First, an auxiliary function \( \text{Defs} \) is defined which collects all let bindings within an expression. Then, the relation \( \text{Occurs} \) extracts the binding from an \texttt{unshare}-rule and checks whether it is an element of \( \text{Defs} \). Because reduction is only allowed on wellformed expressions (i.e. they must be closed and they must have unique variables), being an element of \( \text{Defs} \) automatically ensures the validity of a let binding.

**Definition 7.4.7.2: (let bindings within an expression)**

The function \( \text{Defs} : \mathcal{E} \rightarrow \mathcal{V} \times \mathcal{E} \) is defined recursively by:

\[
\begin{align*}
\text{Defs}(\text{var } x) &= \emptyset \\
\text{Defs}(\text{fun } f \text{ on } xs) &= \emptyset \\
\text{Defs}(\text{app } e \text{ to } x) &= \text{Defs}(e) \\
\text{Defs}(\text{let}(x_1 \ldots x_n) = (e_1 \ldots e_n) \text{ in } e) &= \bigcup_{i=1}^{n} [(x_i, e_i) \cup \text{Defs}(e_i)] \cup \text{Defs}(e)
\end{align*}
\]

**Definition 7.4.7.3: (verification of an \texttt{unshare}-rule)**

The relation \( \text{Occurs} \subseteq \mathcal{R} \times \mathcal{E} \) is defined by:

\[
\text{Occurs}(r,e) \iff \forall x \in \mathcal{V} \forall u \in \mathcal{E} [r = (\text{unshare} \ x \text{ to } u) \Rightarrow (x,u) \in \text{Defs}(e)]
\]
The verification of a rule selector can now be formalized by means of the relation \( \text{Valid} \), which is simply a conjunction of \( \text{Fresh} \) and \( \text{Occurs} \):

**Definition 7.4.7-4: (verification of a rule selector)**

The relation \( \text{Valid} \subseteq R \times E \) is defined by:

\[
\text{Valid}(r, e) \Leftrightarrow \text{Fresh}(r, e) \land \text{Occurs}(r, e)
\]

Inner reduction is formalized by means of the total function \( \text{InnerReduce} \). This function acts as the identity if the input arguments are not wellformed, or the reduction rule cannot be applied successfully. The input is wellformed if: (1) the location is valid; (2) the rule selector is valid; (3) the expression is closed; and (4) the bound variables within the expression are unique. The explicit conditions (3) and (4) restrict reduction to wellformed expressions only.

**Definition 7.4.7-5: (inner reduction)**

The function \( \text{InnerReduce} : L \times R \times E \to E \) is defined by:

\[
\text{InnerReduce}(l, r, e) = \begin{cases} 
\text{Set}(l, \text{HeadReduce}(r, e'), e) & \text{if } \text{Get}(l, e) = e' \land \text{Valid}(r, e) \\
& \land \text{Vars}(e) = \emptyset \land \text{Unq(Bound}(e)) \\
e & \text{otherwise}
\end{cases}
\]

Note that the result of reduction is always a wellformed expression itself. This property can be verified easily; therefore, its proof is omitted here.

### 7.5 Correctness of let lifting

Our system is non-standard only in the handling of sharing. Other than that, it can be regarded as a simplification of a single-step version of [Lau93]. It is easy to see, however, that our approach with let lifting is equivalent to the standard approach which makes use of external environments:

- Suppose that \( R \) is our reduction system, and that \( R' \) is obtained out of \( R \) by replacing the let-lifting mechanism with a usual external environment mechanism. That is, \( R' \) is obtained out of \( R \) by:
  - leaving out the rules \( \text{lift app} \) and \( \text{lift let} \);
  - introducing external environments \( \Gamma \subseteq V \times E \);
  - changing the signature of reduction from \( E \to E \) to \( \Gamma \times E \to \Gamma \times E \);
  - adding a rule \( \text{introduce let} \) that removes a let expression and moves the let bindings in the external environment; and
  - altering the rule \( \text{unshare} \) to use the external environment.

- Then, all reduction paths of \( R' \) can be transformed to \( R \) by:
  - leaving out external environments and all applications of \( \text{introduce let} \);
  - inserting as many \( \text{lift app} \)'s before each application of \( \text{collect} \) as there are inner lets in the application node;
  - inserting as many \( \text{lift let} \)'s before each application of \( \text{unshare} \) as there are inner lets in the binding to be unshared; and
Section 7.6: Confluence

Confluence is a well-known property of rewrite systems. It is important for our system, because it ensures that all possible reductions preserve the meaning of an expression, and can therefore safely be applied in the context of reasoning.

In our reduction system, confluence only holds modulo $\alpha$-conversion, because no explicit $\alpha$-conversion rule is available. Therefore, if two expands are carried out on the same redex, or two expands are carried out on different redexes but there is an overlap in the variables that they introduce, then the reduction results cannot be brought together. This precondition of confluence is formalized by the relation $\text{Joinable}$. Furthermore, $\text{Joinable}$ also excludes the irrelevant and trivial case that the two reductions are identical.

Definition 7.6.1: (precondition of confluence)

The relation $\text{Joinable} \subseteq \mathcal{L} \times \mathcal{R} \times \mathcal{L} \times \mathcal{R}$ is defined by:

\[
\text{Joinable}(l_1, r_1, l_2, r_2) \iff \neg(l_1 = l_2 \land r_1 = r_2) \land \forall x, y \in V \left[ (r_1 = \text{expand } x \land r_2 = \text{expand } y) \Rightarrow (l_1 \neq l_2 \land \neg \exists x \in V \left[ x \in x \land x \in y \right]) \right]
\]

Below we present the proofs of confluence, which are built incrementally. First, we prove confluence for two single head steps, then for one head step and one inner step, and then finally for two inner steps. Without loss of generality, we present simplified proofs and abstract from wellformedness altogether.

Lemma 7.6.2: (confluence - head/head version)

\[
\forall e \in \mathcal{E} \forall r_1, r_2 \in \mathcal{R} \left[ \text{Joinable}(\langle \rangle, r_1, \langle \rangle, r_2) \Rightarrow \exists r_1, r_2 \in \mathcal{R} \left[ \text{HeadReduce}(r_1, \text{HeadReduce}(r_1, e)) = \text{HeadReduce}(r_1, \text{HeadReduce}(r_2, e)) \right] \right]
\]

Proof:

Assume $e \in \mathcal{E}$, $r_1, r_2 \in \mathcal{R}$ and $\text{Joinable}(\langle \rangle, r_1, \langle \rangle, r_2)$.

As can be seen in Table 7.1, on each kind of expression there is only one kind of reduction rule available. Therefore, $r_1$ and $r_2$ must be of the same kind.

Due to assumption $\text{Joinable}(\langle \rangle, r_1, \langle \rangle, r_2)$, $r_1$ and $r_2$ cannot be the same and cannot be expand's. Therefore, $r_1$ and $r_2$ can only be different applications of lift bind:
[5]e = (let xs = bs in e₁).
[6]1 ≤ i < j (if i > j then simply swap them),
[7]xs = (x₁:x₂:x₃) (with #x₁ = i-1 and #x₂ = j-1).
[8]bs = (b₁:b₂:b₃) (with #b₁ = i-1 and #b₂ = j-1).
[9]bᵢ = (let ys = gs in y) and [10]bⱼ = (let zs = hs in h).

The basic idea is that the lift lifts can simply be swapped. However, the index of the binding in r₁ has to be increased, because the lift performed by r₁ has pushed additional bindings upwards. This is not necessary in the reverse case, because the lift of j takes place behind the lift of i.


Now, using HR as abbreviation for HeadReduce, the following holds:

HR(r₃, HR(r₁, e))
= HR(r₃, HR(lift bind i, let xs = bs in e₁))
= HR(lift bind j + #ys, let (x₁:ys:x₂:x₃) = (b₁:gs:b₂:b₃) in e₁)
= (let (x₁:ys:x₂:x₃) = (b₁:gs:b₂:b₃) in e₁).

Again using HR as abbreviation for HeadReduce, the following also holds:

HR(r₄, HR(r₂, e))
= HR(r₄, HR(lift bind j, let xs = bs in e₁))
= HR(lift bind i, let (x₁:ys:x₂:x₃) = (b₁:gs:b₂:b₃) in e₁)
= (let (x₁:ys:x₂:x₃) = (b₁:gs:b₂:b₃) in e₁).

Therefore, HR(r₃, HR(r₁, e)) = HR(r₄, HR(r₂, e)). QED.

Lemma 7.6.3: (confluence - head/inner version)
∀e ∈ E, r₁, r₂ ∈ R, l ∈ L : Joinable(⟨⟩, r₁, l, r₂) ⇒
∃r₃, r₄ ∈ R, l' ∈ L : InnerReduce(l', r₃, HeadReduce(r₁, e)) =
HeadReduce(r₄, InnerReduce(l, r₂, e))

Proof:
Assume e ∈ E, r₁, r₂ ∈ R, l ∈ L and Joinable(⟨⟩, r₁, l, r₂).
If l = ⟨⟩, then the previous Lemma can simply be applied.
If l occurs within a free expression variable of the left-hand-side pattern of r₁ (i.e. no overlap with r₁), then the following arguments hold:
• Rule r₂ on a modified l₂ is applicable on HeadReduce(r₁, e).
  All expression variables that are used in the left-hand-side of a reduction rule occur unchanged in the right-hand-side. In other words: r₁ moves the redex of r₂ around, but does not change it.
• Rule r₁ is applicable at the head of e₂.
  The reduction r₂ only changes the contents of an expression variable in the left-hand-side pattern of r₁. If r₁ matches on e, it therefore also syntactically matches (at the head) on e₂. Furthermore, note that it is not possible that the conditions of r₁ are falsified by r₂, or vice versa.
• The reductions r₁ and r₂ can be swapped, without changing the result.
  This follows from the two arguments above.
This only leaves a partial overlap between \( r_1 \) and \( r_2 \) to be considered. An inspection of Table 7.1 reveals that there are two such cases: either \( r_1 \) is a ‘lift app’ and \( r_2 \) is a ‘lift bind’; or \( r_1 \) is a ‘lift bind’ and \( r_2 \) is an inner ‘lift bind’.

In both cases, \( r_1 \) and \( r_2 \) can be swapped, similarly to Lemma 7.6:2. The full proof is omitted here, but it can be found in [dvP07b]. QED.

**Theorem 7.6: (confluence)**

\[
\forall e \in \mathcal{E} \forall r_1, r_2 \in \mathcal{R} \forall l_1, l_2 \in \mathcal{L} \left[ \text{Joinable}(l_1, r_1, l_2, r_2) \Rightarrow \exists r_3, r_4 \in \mathcal{R} \exists l'_1, l'_2 \in \mathcal{L} \left[ \text{InnerReduce}(l'_1, r_3, \text{InnerReduce}(l_1, r_1, e)) = \text{InnerReduce}(l'_2, r_4, \text{InnerReduce}(l_2, r_2, e)) \right] \right]
\]

**Proof:**

Assume \( e \in \mathcal{E} \), \( r_1, r_2 \in \mathcal{R} \), \( l_1, l_2 \in \mathcal{L} \) and \( \text{Joinable}(l_1, r_1, l_2, r_2) \).

Assume that \( l_1 \) is at least as close to the root of \( e \) as \( l_2 \). If otherwise, then simply swap \( l_1 \) and \( l_2 \). We distinguish two cases:

- **Case 1:** \( l_2 \) is a sublocation of \( l_1 \). Now, \( r_1 \) is a head reduction of \( \text{Get}(l_1, e) \), and \( r_2 \) is an inner reduction of \( \text{Get}(l_1, e) \). By applying Lemma 7.6:3, \( r_1 \) and \( r_2 \) can be brought together in the context of \( \text{Get}(l_1, e) \). Because a reduction of a subexpression is always also a reduction of the expression as a whole, \( r_1 \) and \( r_2 \) can be brought together in the context of \( e \) as well.

- **Case 2:** \( l_2 \) is not a sublocation of \( l_1 \). In this case, \( r_1 \) and \( r_2 \) are completely disjoint. Their redex transformations therefore do not interfere with each other at all, and can be swapped leading to the same single result. QED.

### 7.7 Related work

Our reduction system is based on reduction as proposed by Launchbury in [Lau93], which has since 1993 been used as the de facto standard for evaluating lazy functional programs. Several systems have been derived from Launchbury’s, but none that we know of leaves the choice of redex free. Derived systems of interest are [BS99], which defines an operational semantics specifically for CLEAN, and [HBTK98], which defines a single-step reduction system for parallel evaluation. Both systems fix a single redex, however, and are therefore less suited for formal reasoning.

In [AMO+95] the authors describe a single-step reduction system based on a call-by-need extension of the lambda calculus, which fully supports lazy evaluation and sharing. It is both single-step and leaves the choice of redex free. The disadvantage of this system, however, is the syntactical distance between the lambda calculus and (the core of) a lazy functional programming language. This distance is most apparent in the representation of functions and applications. Due to this distance, the system of [AMO+95] is not suited for dedicated formal reasoning on the level of the program, which is one of the trademark features of SPARKLE.
Chapter 7: A Single-Step Term-Graph Reduction System

Related more generally is the $\rho_g$-Calculus\[BBCK07\], which integrates term-rewriting with lambda-calculus, expressing sharing and cycles. It uses both unification and matching constraints, leading to a term-graph representation in an equational style. This calculus is more general than classical term graph rewriting\[SPM93, BvG+87b\], which can be simulated in it. We feel that our work can serve as a first basis for creating a reduction system for a proof assistant based on the $\rho_g$-calculus.

Another future issue concerns the addition of tactical support for equivalency of cyclic graphs. This may be based upon the work of \[Gra07\], which establishes the bisimilarity of different proof systems for equational cyclic graph specifications.

7.8 Conclusions

We have defined a term-graph reduction system for a simplified lazy functional language. Our system uses single-step reduction and leaves the choice of redex free. This offers a degree of flexibility that is not available in the commonly used reduction systems for functional languages. Due to this degree of flexibility, our system is much better suited for the foundation of formal reasoning. Our reduction system is used in the foundation of Sparkle, Clean’s proof assistant.

Our system maintains sharing within expressions and does not use external environments. This offers the advantage of orthogonality: expressions can be given a meaning as they are, whereas in the common reduction systems they have to be combined with an environment first. The internal maintenance of sharing does not make the reduction system more complicated; it suffices to add two additional rules for let-lifting. All in all, our system consists of five reduction rules only, and is very simple.

All common reduction paths can be expressed in our system. Furthermore, we have proved that our system is confluent. This implies that our system is equivalent to the standard systems: there is at least one reduction path that corresponds to normal reduction, and all other paths can be converged to it.
Chapter 8

Abstract. The formal framework of SPARKLE describes the definitions that are needed for formal reasoning (expressions, programs, reduction, propositions, semantics, proofs, reasoning steps), and proves several important properties of these definitions (confluence of reduction, referential transparency of semantics and soundness of reasoning steps). The full framework serves as a reference work, and has been published as a 209 page technical report. Its main scientific contributions are the reduction mechanism and its confluence, which have been condensed into a published scientific article (included as Chapter 7 of this thesis). Also of interest is the definition of the semantics of expression equality, which is not new from a scientific point of view, but is very important for understanding the foundation of SPARKLE.

This chapter presents the framework definition of expression equality. It is a modified version of chapter 7 of the framework, in which sections 7.1, 7.12, 7.14 and 7.15 have been removed, because they are not relevant for the use of expression equality in this thesis. Furthermore, the chapter has been updated slightly, and explanations of used notations have been added to make the chapter self-contained.

The formal framework builds upon a complete version of CORE, which is also the foundation of this chapter. The definitions that are presented in the other chapters of this thesis, and in particular the expression language and reduction mechanism that are defined in Chapter 7, are in essence simplifications of this complete version. Consequently, this chapter can be related to the rest of this thesis easily.

8.1 Introduction

This chapter presents a formal semantics of expression equality. The goal is to determine, by means of a logic formula, whether two arbitrary CORE expressions
are equal in the context of a fixed CORE program. The definition of equality is non-trivial, because it has to take lazy evaluation, non-termination and infinite expressions into consideration, which cause it to undecidable in general.

The basic idea is that two expressions $e_1$ and $e_2$ should only be considered equal if $e_1$ can safely be replaced by $e_2$ in an arbitrary program without changing its observational behavior. This behavior is determined completely by the output that the program produces when it is executed, which can be formalized elegantly. This strictly operational view in terms of execution and observation is able to cope with all required aspects of equality, including non-termination, without having to rely on heavy mathematics. Moreover, the desired property of referential transparency is implied by it automatically.

This chapter is structured as follows. In Section 8.2, the used definitions from the earlier chapters of the full framework will be explained informally. In Section 8.3, a general overview is presented of our approach to defining semantical expression equality, and it is briefly compared to other approaches. In Section 8.4, an informal observational model of the behavior of a CLEAN program is developed from the user’s point of view, who perceives the program as a black-box. In Section 8.5, this model is refined by allowing the contents of the program to become visible. In Section 8.6, the refined model is formalized in CORE step-by-step, equivalence of program behaviors is formalized, and equality of CORE expressions is expressed in terms of it. In Section 8.7, some examples of equal and unequal expressions are presented. In Section 8.8, the property of ‘reducibility’ is proved for our semantic equality. In Section 8.9, finally, conclusions are drawn.

### 8.2 Used definitions from the full framework

This chapter makes use of definitions that are formally introduced in the earlier chapters of the full framework[dlvP07a]. To allow this chapter to be read on its own, these definitions will be explained by means of informal assumptions, as follows:

- Assume that $\langle A \rangle$ denotes the set of lists of elements of $A$, and that elements of $\langle A \rangle$ are denoted by $\langle a_1, a_2, \ldots, a_n \rangle$. Assume that lists can be matched upon by means of the usual constructors nil and cons, and that $|xs|$ denotes the length of the list $xs$.

- Assume that $\mathcal{V}_e$ denotes the set of expression variables, $\mathcal{B}_v$ the set of basic values, $\mathcal{C}$ the set of constructor symbols, $\mathcal{F}$ the set of function symbols, and $\mathcal{S} = \mathcal{C} \cup \mathcal{F}$ the set of all symbols.

- Assume that $\mathcal{E}$ denotes the set of expressions, which can either be:
  - a variable expression ‘exprvar $x$’ ($x \in \mathcal{V}_e$);
  - a basic expression ‘basic $b$’ ($b \in \mathcal{B}_v$);
  - a symbol expression ‘symbol $s \sigma$ es’ ($s \in \mathcal{S}$, es $\in \langle \mathcal{E} \rangle$; $\sigma$ is a list of type instances for the the free variables of $s$);
Section 8.3: Expression equality in CORE: an overview

- an application;
- a case distinction;
- a lazy let ‘let binds in e’ (e ∈ E; binds is a list of variable bindings);
- a strict let;
- an error value ⊥.

Applications, case distinctions, strict lets and error values will not be used explicitly in this chapter; therefore, no representation is provided for them.

An expression is in root normal form if its root cannot be reduced any more, which is the case for variable expressions, basic expressions, partial applications, total constructor applications and ⊥.

- Assume that free variables are collected by the function Vars.
  Assume that \( E_{\text{closed}} = \{ e \in E \mid \text{Vars}(e) = \emptyset \} \) denotes the set of closed expressions.
  Assume that \( E_{\text{opened}}(x) = \{ e \in E \mid \text{Vars}(e) \subseteq \{ x \} \} \) denotes the set of expressions that contain \( x \) as free variable only.

- Assume that the arity of a symbol is retrieved with the function Arity.  
  Assume that \( \text{Partial}(S) = \{ \text{symbol } s \ as \ es \in E \mid \text{Arity}(s) > |es|, s \in S \} \) denotes the set of partial applications of symbols \( s \in S \).
  Assume that \( \text{Total}(S) = \{ \text{symbol } s \ as \ es \in E \mid \text{Arity}(s) = |es|, s \in S \} \) denotes the set of total applications of symbols \( s \in S \).

- Assume that \( \Psi \) denotes the set of all programs. A program consists of symbol definitions and a start expression.

- Assume that a reduction mechanism is available for expressions, by means of the relation \( e_1 \xrightarrow{\psi} e_2 \) (‘\( e_1 \) reduces to \( e_2 \) in one step in the context of \( \psi \)’) and the function \( \text{Reducts}_\psi(e) \) (which produces the set of all \( e' \) that can be obtained from \( e \) by applying one or more reduction steps).

  Assume furthermore that this reduction is confluent modulo \( \alpha \)-conversion, which means that for all (\( e \xrightarrow{\psi} e_1 \)) and (\( e \xrightarrow{\psi} e_2 \)) there exist \( e'_1 \) and \( e'_2 \) such that (\( e_1 \xrightarrow{\psi} e'_1 \)), (\( e_2 \xrightarrow{\psi} e'_2 \)) and \( e'_1 =_{\alpha} e'_2 \). Note that both the formal framework and Chapter 7 of this thesis define a reduction mechanism that satisfies confluence modulo \( \alpha \)-conversion.

The full formal definitions can be found in [dvP07a].

8.3 Expression equality in CORE: an overview

As was already stated in the introduction, two expressions will be considered semantically equal if and only if they can be replaced in an arbitrary program without changing its observational reduction behavior. In this (yet informal) definition, two different kinds of equalities can be identified: (1) equality between programs, which is a practical concept that can be determined by means of observation of reduction behavior; and (2) equality between expressions, which
is a theoretical concept that cannot be determined by means of observation, but is instead translated to a quantified statement over a set of program equalities.

This definition of theoretical expression equality is thus given in terms of practical program embeddings. This approach to defining expression equality has the following characteristics:

- **It is straightforward to formalize.**
  Using program embeddings, formalizing expression equality boils down to formalizing program equality in terms of their observational reduction behaviors. Because this observation is a practical concept, understanding it on the intuitive level is not difficult. It turns out that this intuition can be transformed to a formal definition quite elegantly. Although the resulting formal definitions themselves are still somewhat complicated, they can all be understood easily.

Note that in other definitions of expression equality program equality has to be formalized as well, because the safety of replacing equal expressions is a main requirement that must be proved anyway. Formalizing program equality is thus not an additional effort.

- **Formally showing that expressions are unequal is easy.**
  It suffices to find one program embedding for which the observational reduction behavior (which can be obtained intuitively) changes when the expressions are interchanged. This program can often be found easily.

- **Formally showing that expressions are equal is difficult.**
  Now, for all possible program embeddings it has to be checked that the observational reduction behavior does not change when the expressions are interchanged. This is both difficult to get an intuitive understanding of (how many embeddings need to be checked?) and difficult to show formally (how can you prove this?).

  However, by means of tactics convenient mechanisms to prove expression equality can still be defined. The correctness proofs of these tactics will be slightly more difficult, however.

- **Program equality cannot deal with partial applications.**
  Unfortunately, in case partial applications occur in a program, semantics and semantical equality cannot be determined on the basis of observational reduction behavior alone. Consider for example the following programs:

  \[
  \text{sum} \quad , \quad \text{sum} \circ \text{reverse} \quad , \quad \text{sum} \circ \text{tl}
  \]

  These three programs will either all have different observational reduction behaviors (if the execution output is simply the textual representation of the partial application) or all have the same observational reduction behavior (if the execution output is a notification that a partial application was encountered). In either case, it is not possible to determine that the first and second program are equal, but the second and the third are not.
In this thesis, program equality is only used as an intermediate stage for determining expression equality. Fortunately, it turns out that choosing to consider all partial applications as semantically equal on the program level, which can be realized by observing them all as some constant ‘opartial’, suffices to build a correct semantical equality on top of program equality.

• **Expression equality can deal with partial applications.**

When determining the theoretical equality between expressions \( e_1 \) and \( e_2 \) that contain partial applications, the following happens:

- For each program embedding \( \psi \) that does not completely remove the partial applications within both \( e_1 \) and \( e_2 \), the derived programs \( \psi[e_1] \) and \( \psi[e_2] \) are automatically equal: they can only differ in partial applications and these are always observed identically.

- Therefore, only program embeddings \( \psi \) that do remove all the partial applications within both \( e_1 \) and \( e_2 \) have to be considered. There can now be two possibilities:
  - For one of these program embeddings, the derived programs \( \psi[e_1] \) and \( \psi[e_2] \) are semantically different. Because \( \psi \) is a valid program embedding, this thus shows the inequality of \( e_1 \) and \( e_2 \).
    
    **Example:** The expressions ‘sum’ and ‘sum o tl’ are shown to be unequal by the program embedding ‘\( \bullet [1] \)’, where ‘\( \bullet \)’ denotes where the expressions should be filled in.
  
    - For all of these program embeddings, the derived programs \( \psi[e_1] \) and \( \psi[e_2] \) are semantically equal. This then shows that \( e_1 \) and \( e_2 \) are equal as well. Note that this means that the equality between partial applications is determined by means of the equality of all their possible total applications. This is a form of *extensionality* that is usual in the semantics of functional languages.
      
    **Example:** The expressions ‘sum’ and ‘sum o reverse’ are equal, which is shown sufficiently by all program embeddings of the form ‘\( \bullet xs \)’.

In other words: by observing all partial applications within programs as equal, theoretical expression equality works correctly for all kinds of expressions, including those containing partial applications.

• **Expression equality trivially implies referential transparency.**

Referential transparency is a main requirement of expression equality. In our approach it is satisfied easily, because equality requires expressions to be embedded into arbitrary programs already. This does make the definition of our expression equality slightly more complex, however.

Other valid descriptions of expression equality are possible, but they only make the definition of equality easier at the cost of making the proof of referential transparency more difficult. An example of such an alternative approach is to define equality recursively, and to only embed (by means
of an application on an arbitrary additional argument) when a top-level partial application is reached.

We have chosen to incorporate referential transparency into the definition of equality by means of program embeddings.

Defining expressions equality in terms of program embeddings thus has several important advantages, but due to its complexity some disadvantages as well. Other operational approaches by which expression equality can be determined are not less difficult, however.

Another alternative is to define expression equality \textit{denotationally}. This requires the use of much ‘heavier’ mathematics (see for instance [Lau93]). The definition given in this thesis only makes use of ‘simple’ mathematics (most importantly, it completely avoids the use of infinite terms), but still describes expression equality concisely and elegantly.

\section{8.4 External behavior of \textsc{Clean} programs}

From the user’s point of view, a compiled program written in \textsc{Clean} is perceived as a \textit{black box}, of which only two things are known: (1) it can be executed on a computer; and (2) during this execution, which may either stop at some point or go on indefinitely, the program gradually produces units of ‘physical’ output which can be observed by the user. Programs that require input from the user will not be considered in this chapter and other characteristics of execution (such as the consumption of resources) will be ignored.

The \textit{meaning} of a total program is determined completely by the output that it produces over the course of time. Because the program is a black-box, this meaning can only be determined by the user by means of observation. However, only the \textit{partial output} that has been produced by the program up to a certain point can be observed. This implies that the meaning of a total program can only be determined by means of observation if it terminates. If the program does not terminate, there are two possibilities:

- \textit{The program ‘hangs’ and stops producing output from a certain point.}
  In this case, the last observation before it hangs actually determines the meaning of the program, but this is not known to the user, because he does not know that the program will never produce any more output.

- \textit{The program is ‘productive’ and keeps producing output indefinitely.}
  In this case, the total output produced by the program will be infinite and can thus never be observed in practice.

The meaning of a non-terminating program can be determined by means of observation on the \textit{theoretical level}, however, by assuming that the user: (1) is able to observe programs infinitely long; and (2) is able to observe infinite outputs. Using this assumption of a \textit{super-user}, the following theoretical model of program result can be constructed:
• Describe the observations of the user by a function $\text{Obs}$ from time units (which are assumed to be elements of a countable set $T = \{t_0, t_1, t_2, \ldots\}$) to accumulative outputs. The result of this function applied to a time unit $t$ is thus the total output that has been produced by the program up to $t$.

• Build the output sequence of the program, which is the infinite sequence of observations $\langle \text{Obs}(t_0), \text{Obs}(t_1), \text{Obs}(t_2), \ldots \rangle$. Because the accumulated output of a program grows in time, each output sequence is ascending.

• The total execution output of a program, call this the program result, is the least upper bound of the output sequence $\left(\bigcup \{\text{Obs}(t_i) \mid i \in \mathbb{N}\}\right)$. This program result is a finite object for terminating and hanging programs and an infinite object for productive programs. The program result completely determines the behavior of the (total) program.

By comparing program results, it is now possible to determine if two (total) programs behave the same. Note that in practice, deciding that two programs behave the same is only possible for terminating programs. However, deciding that two programs do not behave the same, does not require a super-user and is usually possible in practice (you just have to wait long enough).

8.5 Internal behavior of Clean programs

More details can be added to the observational model of program behavior by no longer regarding the Clean program as a black-box. Instead, the program is now assumed to consist of two concrete components: a start graph $G$, which will actually be evaluated and which is assumed not to contain any partial applications; and a list of (function) definitions $L$, which are required for the evaluation of (function) symbols within $G$.

Execution was described in the previous section as an abstract process that takes place within the black-box of the program and of which only is known that it gradually produces output. Now that the black-box has been opened, this process and the output that it produces can be described in more detail by:

• **Execution**: the continuous reduction of $G$ in the context of $L$ according to the functional reduction strategy. From an abstract point of view, this strategy evaluates the left-most outer-most redex to root normal form and on success continues recursively with the next left-most outer-most redex. On failure, the reduction as a whole is terminated with an error message. More formally, execution can now be represented by the execution sequence $\langle G_0, G_1, G_2, \ldots \rangle$, where $G_0 = G$ and $G_{i+1}$ is obtained out of $G_i$ by applying one reduction step according to the functional reduction strategy. Note that this execution sequence is finite for terminating programs and infinite for halting and productive programs.

• **Production of output**: the execution output of the program is displayed (pretty-printed) on the console window. Each time execution successfully
evaluates a left-most outer-most redex to root normal form, this output is extended with the root of the produced root normal form. Theoretically, the output can therefore be regarded as a flat list of output units, where each output unit is either a basic value (all basic values are in root normal form), a constructor symbol (all total constructor applications are in root normal form) or the special symbol opartial (for all partial applications). Note that variables and error values are also root normal forms, but can never be produced as the result of successful reduction.

In the case of partial applications, this output differs from the console output that real CLEAN programs produce in practice. As was discussed earlier, this is necessary to enable expression equality to handle partial applications correctly.

The abstract time units, which were required in the previous section to enforce the progression of execution and observation, can now be replaced by indexes in the execution sequence. These indexes represent the number of reduction steps that have been applied to \( G \) so far. Using these indexes, observation can now be described in more detail by:

- Observation is represented by a function \( \text{Obs} \) from execution indexes to execution outputs. The result of this function applied to an index \( i \) is the total output that has been produced by execution up to \( G_i \).

- Each execution output is represented by a tuple of a list of output units and an execution status, where the execution status is either ‘busy’, ‘finished successfully’ or ‘finished with error <msg>’.

- Define an output graph to be a graph in which:
  - all leafs contain either a basic value, the symbol opartial, the execution status ‘busy’ or the execution status ‘finished with error <msg>’;
  - all nodes contain a constructor symbol.

Each execution output can now alternatively be represented by a single output graph. This output graph is obtained as follows:

- If the execution status is ‘finished successfully’, then the constructed list of output units uniquely describes a complete output graph (in which no execution status occurs).

- If the execution status is ‘busy’ or ‘finished with error <msg>’, then the list of output units describes the head of a fully evaluated graph. By inserting dummy leafs with the execution status at the missing locations, this head can then be completed to an output graph.

Note that (backwardly) this output graph uniquely identifies a tuple of a list of output units and an execution status.
Section 8.6: Behavior of Core programs

- Again, the output sequence of a program can be built. It is represented by \( \langle \text{Obs}(0), \text{Obs}(1), \text{Obs}(2), \ldots \rangle \), using the representation of observation in terms of output graphs.

Finally, the total execution output of the program can still be represented by the least upper bound of the output sequence. This program result completely determines the observational behavior of the program.

8.6 Behavior of Core programs

In this section, the observational model of program behavior will be formally applied to Core. In Section 8.6.1, the set of executable programs is formally defined. In Sections 8.6.2 and 8.6.3, execution and observation will be formalized respectively. In Core, this results in execution and output sets, rather than sequences. In Section 8.6.4, equivalence of output sets is formalized. This will be done without using least upper bounds or infinite program results. Finally, in Section 8.6.5, expression equality is formalized in Core.

8.6.1 Executable programs in Core

As was established earlier, an executable program in Clean consists of a start graph and a list of symbol definitions. Both these components can be expressed in Core easily: the start graph can be associated with an expression \( \in E \) and the list of definitions can be associated with a program \( \in \Psi \). Additionally, the start expression of an executable program is required to be closed, because execution is not considered to be defined for free variables.

Definition 8.6.1: (executable programs)

The set of executable programs \( \Psi_{\text{run}} \) is defined by:

\[
\Psi_{\text{run}} = \{ \text{run } e \text{ with } \psi \mid e \in \mathcal{E}_{\text{closed}}, \psi \in \Psi \}
\]

Note that there is a distinction in Core between ordinary programs in \( \Psi \), which cannot be executed, and executable programs in \( \Psi_{\text{run}} \), which can be executed. Both will usually be denoted by the letter \( \psi \), however.

8.6.2 Execution within Core

Both execution and observation within Clean heavily depend on the functional reduction strategy. This strategy dictates in which order the start graph should be reduced, and therefore also which parts of the reduced graph will produce execution output that can then be observed from the outside. In Core, however, this dependency will be removed totally. This has the following consequences:

- Execution is no longer modeled by the reduction sequence according to one particular reduction strategy, but simply by the unordered set of all possible reduction results, regardless of the strategy used.
This allows execution to be formalized in terms of reduction systems which leave the choice of redex free, such as the one defined in Chapter 7 of this thesis.

- The program result, which is the least upper bound of all observations, may contain more information than in the dependent case, because parts of the program get evaluated which would normally have been left alone by the functional reduction strategy. This thus changes the semantics of program equality, which becomes more restrictive.

Fortunately, this has no effect on the semantics of expression equality. This is because:

- Although execution is now able to evaluate any redex regardless of its location, output can and will (see next section) only be produced for parts of the reduced start expression that are on a head position, meaning that they are separated from the root by total constructor applications only.

In other words, the results of the evaluation of redexes which are not on head positions will be ignored completely by observation and therefore do not affect the semantics at all.

- Even though in CLEAN program equality is only determined on the basis of evaluation according to the functional reduction strategy, expression equality still enforces the evaluation of all redexes on head positions, because for all head positions there exists a surrounding program which selects and then evaluates its redex. Take for example the expression \texttt{Cons (Pair 7 E) Nil}. Reduction of \(E\) according to the functional reduction strategy can now be enforced by embedding the expression in the program \texttt{snd (hd •)}.

In other words, when determining expression equality, the embedding in programs enforces the evaluation of all redexes on head positions, regardless of the strategy used to determine program equality.

- Observation can no longer be linked to execution (see next section).

To summarize: removing the dependency of the functional reduction strategy makes program equality more restrictive, but does not affect expression equality.

Execution can therefore now be formalized in \textsc{Core} by the set of all possible reduction results:

\textbf{Definition 8.6.2:} \textit{(execute a program)}

The function \(\text{Execute} : \mathcal{Ψ}^{\text{run}} \to \wp(\mathcal{Ψ}^{\text{run}})\) is defined by:

\[
\text{Execute}(\text{run } e \text{ with } \psi) = \{ \text{run } e' \text{ with } \psi \mid e' \in \text{Reducts}_{\psi}(e) \} 
\]

\section*{8.6.3 Observation in \textsc{Core}}

In the ‘opened’ informal model, observation was defined by means of a function from execution indexes to output graphs. These execution indexes were used
as a means of selecting elements of the execution sequence. Because execution in CORE produces a set instead of a sequence, it is no longer possible to define observation as a function from indexes. Instead, it will therefore be defined as a function operating directly on the intermediate program results.

Analogously to CLEAN, the result of observation will be represented by an output expression. This output expression is basically a flattened representation of an output graph, but differs from it in two places:

- Each execution status ‘finished with error <msg>’ will be represented by the special data symbol `oerror`, of which there is only one. This means that in CORE there will be no semantical difference between different kinds of erroneous reductions.

- The execution status ‘busy’ will be represented by the special symbol `oerror` as well, which in this case acts as a worst-case approximation of each redex in the intermediate program result that has not yet been evaluated to root normal form. There are now two possibilities:

  - *The redex can eventually be reduced to a root normal form.*
    
    Because the execution set contains all possible reduction results, it will now contain at least one element in which the redex has actually been evaluated to a root normal form. Due to the mechanism of the least upper bound (and due to the fact that `oerror` is considered the smallest possible output expression), the semantics of the redex will now be determined completely by this root normal form and will not depend in any way on the smaller intermediate observations.
    
    Therefore, for these redexes, the intermediate observations in terms of `oerror` have no effect at all on the final semantics.

  - *The redex can never be reduced to a root normal form.*
    
    In this case the redex will always remain a redex. It is thus always observed as `oerror` and is therefore semantically equal to an erroneous reduction. This is precisely the intention of the semantics in CORE, because *everything that cannot be reduced to a root normal form will be considered erroneous and equal to each other.*

An output expression is thus an extended normal expression, consisting of basic values and total constructor applications and extended with the special data symbols `opartial` and `oerror`:

**Definition 8.6.3:** (execution outputs)

The set of output expressions \( E_{output} \) is defined by:

\[
E_{output} = \{ \text{basic } b \mid b \in B_v \} \\
\cup \{ \text{ocons } c \ es \mid c \in C, \ es \in (E_{output}) \mid \text{Arity}^2(c) = |es| \} \\
\cup \{ \text{opartial} \} \\
\cup \{ \text{oerror} \}
\]

Observation will now be formalized by means of the function `Observe`, which operates on an intermediate program result (and is overloaded for expressions
and list of expressions too) and produces an output expression. The basic idea of this observation function is to:

- replace all basic expressions by applications of \texttt{obasic};
- replace all total constructor applications by applications of \texttt{ocons} (and also continue recursively on the arguments);
- replace all partial applications by \texttt{opartial};
- continue recursively on the let expression of a lazy let;
- replace all erroneous expressions by \texttt{oerror};
- replace all expressions that are not in root normal form by \texttt{oerror}.

The function \texttt{Observe}, which is overloaded for executable programs, expressions and lists of expressions, can now be defined as follows:

**Definition 8.6.3:2: (observe an expression)**

The function \texttt{Observe : E → E\textsuperscript{output}} is defined by:

\[
\texttt{Observe}(e) = \begin{cases} 
\texttt{obasic}\ b & \text{if } e = (\texttt{basic}\ b) \\
\texttt{ocons}\ c\ \texttt{Observe}(es) & \text{if } e = (\texttt{symbol}\ c\ \texttt{as}\ es) \in \text{Total}(C) \\
\texttt{opartial} & \text{if } e \in \text{Partial}(S) \\
\texttt{Observe}(e') & \text{if } e = (\texttt{let}\ \texttt{binds}\ \texttt{in}\ e') \\
\texttt{oerror} & \text{otherwise}
\end{cases}
\]

**Note that:** this definition differs from the one in the mathematical foundation\cite{dvP07a}, in which unraveling was added explicitly to the observation function. This was needed due to the specialized behavior of its non-standard reduction mechanism, which does not allow variables to be unshared on all positions. In this thesis, it is assumed that the underlying reduction mechanism always allows variables to be unshared, which is the behavior of standard systems, such as Launchbury’s\cite{Lau93} and our system defined in Chapter 7.

**Definition 8.6.3:3: (observe a list of expressions)**

The function \texttt{Observe : \langle E \rangle → \langle E\textsuperscript{output} \rangle} is defined by:

\[
\texttt{Observe}(\texttt{cons}\ e\ es) = \texttt{cons}\ \texttt{Observe}(e)\ \texttt{Observe}(es) \\
\texttt{Observe}(\texttt{nil}) = \texttt{nil}
\]

**Definition 8.6.3:4: (observe an intermediate program result)**

The function \texttt{Observe : Ψ\textsubscript{run} → E\textsuperscript{output}} is defined by:

\[
\texttt{Observe}(\texttt{run}\ e\ \texttt{with}\ \psi) = \texttt{Observe}(e)
\]

Note that the recursive nature of \texttt{Observe} causes observation to stop as soon as something other than a root normal, lazy let definition or variable expression is encountered. Only evaluated redexes at the remaining positions (which will be called the \textit{head positions}) are displayed.

By applying observation on the elements of the execution set, the \textit{output set} of a program can now be obtained:
Definition 8.6.3: (produce the output set of a program)
The function \( \text{Output} : \mathcal{P}_{\text{run}} \rightarrow \mathcal{P}(\mathcal{E}_{\text{output}}) \) is defined by:
\[
\text{Output}(\psi) = \{ \text{Observe}(\psi') \mid \psi' \in \text{Execute}(\psi) \}
\]

In order to determine equivalence between output sets, an ordering relation on output expressions also needs to be defined. The idea behind this ordering is that \( E_1 \) should only be smaller or equal than \( E_2 \) if \( E_2 \) can be obtained out of \( E_1 \) by replacing \( \text{oerror} \)’s. In other words: \( \text{oerror} \) is smaller or equal than anything else and all other outputs are only smaller or equal than themselves. Intuitively, \( E_1 \) is thus smaller or equal than \( E_2 \) iff \( E_2 \) is more specific than \( E_1 \).

The ordering relation will be defined in two steps. First, a function \( \toprel \subseteq \mathcal{E}_{\text{output}} \to \mathcal{E}_{\text{output}} \) is defined, which produces all output expressions that are greater or equal than its argument. Then the relation \( \subseteq \) itself is defined in terms of this function. Note that typing is not considered here at all; therefore, an application of \( \subseteq \) is only really meaningful if the expressions compared are also equally typed.

Definition 8.6.3: (produce all greater output expressions)
The function \( \toprel \) defined by:
\[
\toprel(\text{basic} b) = \{ \text{basic} b \}
\]
\[
\toprel(\text{ocons} c e s) = \{ \text{ocons} c e s' \mid e' \in \toprel(e) \}
\]
\[
\toprel(\text{opartial}) = \{ \text{opartial} \}
\]
\[
\toprel(\text{oerror}) = \mathcal{E}_{\text{output}}
\]

Definition 8.6.3: (produce all lists of greater output expressions)
The function \( \toprel \) defined by:
\[
\toprel(\text{nil}) = \{ \text{nil} \}
\]
\[
\toprel(\text{cons} e e s) = \{ \text{cons} e' e s' \mid e' \in \toprel(e), e s' \in \toprel(e) \}
\]

Definition 8.6.3: (ordering relation on output expressions)
The binary infix relation \( \subseteq \subseteq \mathcal{E}_{\text{output}} \times \mathcal{E}_{\text{output}} \) is defined by:
\[
e_1 \subseteq e_2 \iff e_2 \in \toprel(e_1)
\]

The \( \subseteq \)-relation obeys the standard properties of ordering relations: it is reflexive, transitive and anti-symmetric. Moreover, it also ensures that observations of reduction results are always greater or equal than observations of their originals, which in turn implies that each output set has a least upper bound:

Observations 8.6.3: (observations about \( \subseteq \))

1. The ordering relation \( \subseteq \) is reflexive, transitive and anti-symmetric:
   (a) \( \forall e \in \mathcal{E}_{\text{output}} [e \subseteq e] \);  
   (b) \( \forall e_1, e_2, e_3 \in \mathcal{E}_{\text{output}} [(e_1 \subseteq e_2) \Rightarrow (e_2 \subseteq e_3) \Rightarrow (e_1 \subseteq e_3)] \);  
   (c) \( \forall e_1, e_2 \in \mathcal{E}_{\text{output}} [(e_1 \subseteq e_2 \land e_2 \subseteq e_1) \Leftrightarrow e_1 = e_2] \).
2. A reduce is always observed to be greater or equal than its original:
   \( \forall \psi \in \mathcal{P}_{\text{run}} \forall e_1, e_2 \in \mathcal{E}_{\text{output}} [(e_1 \rightarrow \psi e_2) \Rightarrow (\text{Observe}(e_1) \subseteq \text{Observe}(e_2))] \)
3. In the set of execution outputs, a common upperbound always exists:
   \( \forall \psi \in \mathcal{P}_{\text{run}} \forall e_1, e_2 \in \mathcal{E}_{\text{output}} [(e_1 \subseteq e_3) \land (e_2 \subseteq e_3)] \)
Chapter 8: Semantics of Expression Equality

Rationale:
The first three observations (1a-c) are not of great importance and can all easily be shown by structural induction, using the definitions of $\sqsubseteq$ and More.

The fourth observation (2) depends on the underlying reduction system: the observation of the left-hand-side of each reduction rule should be smaller or equal than the observation of its right-hand-side. This property can easily be verified to hold for the reduction system defined in the mathematical foundation [dvP07a], and also for the reduction system defined in Chapter 7 of this thesis.

The fifth observation (3) is a consequence of confluence, which we assume to hold, and the previous observation: (1) assume that $e_1$ is produced by observing the intermediate reduction result $e'_1$ and that $e_2$ is produced by observing $e'_2$; (2) then confluence ensures that $e'_1$ and $e'_2$ have a common reduct $e'_3$ which is a member of the execution set; (3) finally, the previous observation ensures that both $e_1$ and $e_2$ are smaller or equal than the observation of $e'_3$.

The last observation mentioned above is very important, because it ensures that the execution outputs of reducts according to non-standard evaluation strategies are not in the way of the ‘standard’ execution outputs. Specifically, it ensures that, as in Clean, the program result can be defined as the theoretical least upper bound of the set of execution outputs and that each execution output is a finite approximation of this program result.

8.6.4 Equivalence of output sets in Core

The final step towards formalizing program equality in Core is the formal definition of equivalence between produced output sets. In the informal model, this was realized by means of computing and comparing the least upper bounds of the output sets. Unfortunately, this is difficult to formalize, because these least upper bounds may very well be infinite objects which can therefore not be expressed as elements of inductively defined sets such as $E_{\text{output}}$.

Fortunately, equivalence between output sets can also be formalized without using infinite objects at all. This can be accomplished by means of the following observation, which somewhat resembles bi-simulation:

iff for all $x \in X$ there exists a $y \in Y$ such that $x \leq y$,
and for all $y \in Y$ there exists a $x \in X$ such that $y \leq x$,
then $\sqcup X = \sqcup Y$

Formally applying this observation to output sets in Core yields:

Definition 8.6.4: (equivalence of output sets)
The binary infix relation $\sim \subseteq \wp(E_{\text{output}}) \times \wp(E_{\text{output}})$ is defined by:

\[ O_1 \sim O_2 \iff \forall o_1 \in O_1 \exists o_2 \in O_2 [o_1 \sqsubseteq o_2] \land \forall o_2 \in O_2 \exists o_1 \in O_1 [o_2 \sqsubseteq o_1] \]
Definition 8.6.4: (equality of executable programs)
The binary infix relation \( \cong \subseteq \Psi^{\text{run}} \times \Psi^{\text{run}} \) is defined by:
\[
\psi_1 \cong \psi_2 \iff \text{Output}(\psi_1) \sim \text{Output}(\psi_2)
\]

Note that this program equality only makes sense when both programs agree on the constructor definitions.

The \( \sim \)-relation on output sets is reflexive, symmetric and transitive (and thus an equivalence relation, using the word ‘equivalence’ in a different context), as is shown by the following observations. Consequently, the \( \cong \)-relation on programs is also reflexive, symmetric and transitive.

Observations 8.6.4: (observations about \( \sim \) and \( \cong \))

1. On output sets, \( \sim \) is reflexive, symmetric and transitive:
   (a) \( \forall O \in \mathcal{P}(\mathcal{E}^{\text{output}}) [O \sim O] \);
   (b) \( \forall O_1, O_2 \in \mathcal{P}(\mathcal{E}^{\text{output}}) [O_1 \sim O_2 \Rightarrow O_2 \sim O_1] \);
   (c) \( \forall O_1, O_2, O_3 \in \mathcal{P}(\mathcal{E}^{\text{output}}) [O_1 \sim O_2 \Rightarrow O_2 \sim O_3 \Rightarrow O_1 \sim O_3] \).

2. On executable programs, \( \cong \) is also reflexive, symmetric and transitive:
   (a) \( \forall \psi \in \Psi^{\text{run}} [\psi \cong \psi] \);
   (b) \( \forall \psi_1, \psi_2 \in \Psi^{\text{run}} [\psi_1 \cong \psi_2 \Rightarrow \psi_2 \cong \psi_1] \);
   (c) \( \forall \psi_1, \psi_2, \psi_3 \in \Psi^{\text{run}} [\psi_1 \cong \psi_2 \Rightarrow \psi_2 \cong \psi_3 \Rightarrow \psi_1 \cong \psi_3] \).

Rationale:
The reflexivity\( (1a) \) and transitivity\( (1c) \) of \( \sim \) on output sets trivially follow from the reflexivity and transitivity of \( \subseteq \) (see Observation 8.6.3:9).
The symmetry\( (1b) \) of \( \sim \) on output sets follows trivially from the symmetry in the definition of \( \sim \).
The reflexivity\( (2a) \), symmetry\( (2b) \) and transitivity\( (2c) \) of \( \cong \) on executable programs are implied by their counterparts on output sets.

8.6.5 Expression equality in \textsc{Core}

As was established earlier this chapter, semantical equality between expressions can be informally defined by:

Definition A:
Two expressions \( e_1 \) and \( e_2 \) are semantically equal iff for all executable programs \( \psi \) holds that its observational behavior remains unchanged when \( e_1 \) is replaced by \( e_2 \).

Because in the previous section the notions of executable programs (by means of the set \( \Psi^{\text{run}} \)), execution (by means of the function \textit{Execute}), observation (by means of the function \textit{Observe}), observational behavior (by means of the function \textit{Output}) and equivalence between observational behaviors (by means of the relation \( \sim \)) have already been formalized, it is now possible to formalize expression equality in \textsc{Core} according to this informal definition.

There is, however, one problem with the definition above that must still be solved: it is imprecise regarding symbol semantics:
• The formal meaning of an expression equality can only be determined in the context of a set of symbol definitions, which is required to assign a semantics to the function (and constructor) symbols that occur within the expressions compared. Expressions can be equal in the context of one set of definitions and unequal in the context of another set of definitions. The informal definition above quantifies over all executables programs in which the expressions compared can be filled in. It therefore also quantifies over all possible function definitions, which is undesirable. Instead, the definition should focus on one single set of symbol definitions and quantify over start expressions only.

Correcting this problem, the informal definition can be adjusted to:

Definition B:
In the context of a set of symbol definitions $\psi$, two expression $e_1$ and $e_2$ are semantically equal iff for all embeddings $(e, x)$ holds that the observational behaviors of ‘$\text{run } e_{x \rightarrow e_1}$ with $\psi$’ and ‘$\text{run } e_{x \rightarrow e_2}$ with $\psi$’ are identical.

This informal definition can now be formalized easily. Note that semantical equality is only defined for closed expressions, because execution is only defined for programs with closed start expressions.

Definition 8.6.5\*1: (semantical expression equality)
For all $\psi \in \Psi$, the relation $\text{Equal}_\psi \subseteq \mathcal{E}_{\text{closed}} \times \mathcal{E}_{\text{closed}}$ is defined by:

$$\text{Equal}_\psi(e_1, e_2) \iff \forall x \in V_e \forall e \in \mathcal{E}_{\text{open}}(x)[\text{Output}(\text{run } e_{x \rightarrow e_1} \text{ with } \psi) \sim \text{Output}(\text{run } e_{x \rightarrow e_2} \text{ with } \psi)]$$

Because $\sim$ was already shown to be reflexive, symmetric and transitive, $\text{Equal}$ is reflexive, symmetric and transitive as well:

Observations 8.6.5\*2: (observations about Equal)
1. The semantical equality $\text{Equal}$ is reflexive, symmetric and transitive:
   (a) $\forall \psi \in \Psi \forall e \in \mathcal{E}_{\text{closed}}[\text{Equal}_\psi(e, e)]$;
   (b) $\forall \psi \in \Psi \forall e_1, e_2 \in \mathcal{E}_{\text{closed}}[\text{Equal}_\psi(e_1, e_2) \Rightarrow \text{Equal}_\psi(e_2, e_1)]$;
   (c) $\forall \psi \in \Psi \forall e_1, e_2, e_3 \in \mathcal{E}_{\text{closed}}[\text{Equal}_\psi(e_1, e_2) \Rightarrow \text{Equal}_\psi(e_2, e_3) \Rightarrow \text{Equal}_\psi(e_1, e_3)]$

Rationale:
These observations follow immediately from Observation 8.6.4\*3(2a-c).

The following theorem translates an application of $\text{Equal}$ to concrete statements about observations of the reducts of the expressions compared. Basically, the theorem step-by-step decomposes the application of $\text{Equal}$:

Theorem 8.6.5\*3: (decompose Equal until Observe is reached (1))

$$\forall \psi \in \Psi \forall e_1, e_2 \in \mathcal{E}_{\text{closed}}[\text{Equal}_\psi(e_1, e_2) \Rightarrow \forall e_3 \in \text{Reducts}_\psi(e_1) \exists e_4 \in \text{Reducts}_\psi(e_2)[\text{Observe}(e_3) \subseteq \text{Observe}(e_4)]]$$
Section 8.7: Examples of expression equalities

Proof:
Assume $\psi \in \Psi$ and $e_1, e_2 \in E_{\text{closed}}$.
Assume [1] $\text{Equal}_\psi(e_1, e_2)$, which by expanding $\text{Equal}$ simplifies to:

[2] $\forall x \in \mathcal{V}_e \forall e \in E_{\text{opened}}(x) \left[ \text{Output}(\text{run } e_{x \to e_1} \text{ with } \psi) \sim \text{Output}(\text{run } e_{x \to e_2} \text{ with } \psi) \right]$.

Instantiate [2] with an arbitrary $x \in \mathcal{V}_e$ and $e = (\text{var } x)_{x \to e}$. Also simplifying $e_{x \to e_1}$ and $e_{x \to e_2}$, assumption [2] then implies that [3] $\text{Output}(\text{run } e_1 \text{ with } \psi) \sim \text{Output}(\text{run } e_2 \text{ with } \psi)$ holds.

By expanding $\text{Output}$, $\sim$ and pattern matching, [2] simplifies to:

[4] $\forall e_3 \in \text{Reducts}_\psi(e_1) \exists e_4 \in \text{Reducts}_\psi(e_2) \left[ \text{Observe}(e_3) \subseteq \text{Observe}(e_4) \right]$

and

[5] $\forall e_4 \in \text{Reducts}_\psi(e_2) \exists e_3 \in \text{Reducts}_\psi(e_1) \left[ \text{Observe}(e_4) \subseteq \text{Observe}(e_3) \right]$.

To prove: $\forall e_3 \in \text{Reducts}_\psi(e_1) \exists e_4 \in \text{Reducts}_\psi(e_2) \left[ \text{Observe}(e_3) \subseteq \text{Observe}(e_4) \right]$. This is simply assumption [4].

Corollary 8.6.5.4: (decompose $\text{Equal}$ until $\text{Observe}$ is reached [2])

$\forall \psi \in \Psi \forall e_1, e_2 \in E_{\text{closed}}$

$\text{Equal}_\psi(e_1, e_2) \
\Rightarrow \exists e_4 \in \text{Reducts}_\psi(e_2) \exists e_3 \in \text{Reducts}_\psi(e_1) \left[ \text{Observe}(e_4) \subseteq \text{Observe}(e_3) \right]$

Finally, two more observations can be made about the relation between between semantical equality on programs and semantical equality on expressions. Firstly, terms that are unequal as programs are also unequal as expressions. Secondly, terms that are equal as expressions are also equal as programs:

Observations 8.6.5.5: (observations about relation between $\text{Equal}$ and $\cong$)

1. $\forall \psi \in \Psi \forall e_1, e_2 \in E_{\text{closed}} \neg (\text{run } e_1 \text{ with } \psi \cong \text{run } e_2 \text{ with } \psi) \Rightarrow \neg \text{Equal}_\psi(e_1, e_2)$.

2. $\forall \psi \in \Psi \forall e_1, e_2 \in E_{\text{closed}} \left[ \text{Equal}_\psi(e_1, e_2) \Rightarrow (\text{run } e_1 \text{ with } \psi) \cong (\text{run } e_2 \text{ with } \psi) \right]$.

Rationale:

The first observation holds by the following (trivial) reasoning:

Assume $\neg (\text{run } e_1 \text{ with } \psi) \cong (\text{run } e_2 \text{ with } \psi)$, which simplifies to $\neg (\text{Output}(\text{run } e_1 \text{ with } \psi) \sim \text{Output}(\text{run } e_2 \text{ with } \psi))$ and can then be rewritten to $\neg (\text{Output}(\text{run } (\text{var } x)_{x \to e_1} \text{ with } \psi) \sim \text{Output}(\text{run } (\text{var } x)_{x \to e_1} \text{ with } \psi))$.

This implies $\neg (\text{Equal}_\psi(e_1, e_2))$.

The second observation holds by the following (trivial) reasoning:

Assume $\text{Equal}_\psi(e_1, e_2)$. This implies $\text{Output}(\text{run } (\text{var } x)_{x \to e_1} \text{ with } \psi) \sim \text{Output}(\text{run } (\text{var } x)_{x \to e_1} \text{ with } \psi)$, which can be rewritten to $\text{Output}(\text{run } e_1 \text{ with } \psi) \sim \text{Output}(\text{run } e_2 \text{ with } \psi)$.

8.7 Examples of expression equalities

In this section, some examples will be given of expressions that are equal or unequal according to the given formal semantics. To increase readability, these
examples will be presented on the level of Clean. The formal definitions will still be assumed to be applicable, however.

Next to standard notations (such as \([\_:\_]\) and \([\]\) for lists) and standard functions (such as \(++\) for list concatenation), the following functions will also be used in the examples:

\[
\begin{align*}
\text{repeat} & \:: \ a \rightarrow [a] \\
\text{repeat} \ x & \ = \ [x: \ \text{repeat} \ x] \\
\text{repeatn} & \:: \ \text{Int} \ a \rightarrow [a] \\
\text{repeatn} \ n \ x & \ = \ \text{case} \ (n > 0) \ of \\
& \quad \text{True} \rightarrow [x: \ \text{repeatn} \ (n-1) \ x] \\
& \quad \text{False} \rightarrow []
\end{align*}
\]

The following five expressions all conceptually produce the infinite list of 1’s. Because there does not exist a program embedding which produces different execution output for these expressions, they are semantically equal to each other:

**Example 8.7.1:** \((\text{example of equal expressions (1)})\)

The following pseudo-expressions are all semantically equal:

- \(\text{repeat} 1\)
- \(\text{let} \ x = [1,1:x] \ \text{in} \ [1:x]\)
- \(\text{let} \ x = [1,1,1,1:x] \ \text{in} \ x\)
- \(\text{repeat} 1 \ ++ \ [2,3,4]\)
- \(\text{repeat} 1 \ ++ \ (\text{let} \ x = x \ \text{in} \ x)\)

The following four expressions all conceptually produce the singleton list \([\text{error}]\).

As in the previous example, there does not exist a program embedding which produces different execution output for these expressions, making them thus again semantically equal to each other.

**Example 8.7.2:** \((\text{example of equal expressions (2)})\)

The following pseudo-expressions are all semantically equal:

- \([\text{let} \ x = x \ \text{in} \ x]\)
- \([\text{case} \ 7 \ \text{of} \ 6 \rightarrow 6]\)
- \([\text{undef}]\)
- \([\text{length} \ (\text{repeat} \ 1)]\)

In the following three examples, the inequality between expressions is shown by means of providing a program embedding that distinguishes between them:

**Example 8.7.3:** \((\text{example of unequal expressions (1)})\)

The expressions of examples 8.7.1 and 8.7.2 are semantically unequal. \(\text{Evidence:}\) the program embedding ‘hd \(\bullet\)’, which produces the theoretical program result 1 for all expressions of example 8.7.1, and the theoretical program result \(\text{error}\) for all expressions of example 8.7.2.

**Example 8.7.4:** \((\text{example of unequal expressions (2)})\)

The expressions of example 8.7.1 are semantically unequal to ‘repeatn 47261 1’. \(\text{Evidence:}\) the program embedding ‘\(\bullet\) !! 50000’ (select the
50000th element), which produces the theoretical program result 1 for all expression of example 8.7:1, and the theoretical program result \texttt{error} for \texttt{\textquotesingle\textbackslash repeatn 47261 1\textquotesingle}.

\textbf{Example 8.7:5: (example of unequal expressions (3))}

The expression \texttt{bot} (where \texttt{bot} is defined by \texttt{bot \ x = ⊥}) is unequal to \texttt{⊥}.

\textit{Evidence:} the program embedding \texttt{\textquotesingle let! \ x = \_ in \_\textquotesingle}, which produces the theoretical program result 7 for \texttt{bot} (because \texttt{bot} is a partial function application and therefore a valid root normal form), and the theoretical program result \texttt{error} for \texttt{⊥}. In our system, \texttt{Ω} and \texttt{λ\_\textbackslash x.Ω} are thus not equal.

\textit{Note that:} for any \textit{E}, (\texttt{bot \ E}) reduces to (and is thus equal to) \texttt{⊥}.

The examples were formulated on the semantical level. With two exceptions, they can be proved with \textsc{Sparkle} on the practical level as well. The first exception is the equality of the expressions of example 8.7:1, which cannot be proved in \textsc{Sparkle} because no co-induction reasoning step is available. The second exception is formed by the subexpressions \texttt{let \ x = \_ \ in \ _\textbackslash x} and \texttt{length (\textbackslash repeat 1)} in example 8.7:2. These expressions reduce infinitely with no result, which cannot be detected by \textsc{Sparkle}. They can therefore not be proved equal (or unequal) to anything on the practical level. After removing these expressions, examples 8.7:2-8.7:5 can be proved in \textsc{Sparkle} without problems.

\section{Section 8.8: Characteristics of equality: reducibility}

The semantical equality on expressions obeys several interesting properties. One is \textit{referential transparency}, which is formally proved in chapter 8 of the mathematical framework\cite{dvP07a}. Another is \textit{reducibility}, which states that if $e_1$ reduces to $e_2$ then $e_1$ and $e_2$ must be semantically equal. This property will be formally proved in this section.

First, reducibility for programs will be proved. The basic idea of the proof is as follows: (1) if $ψ_1$ reduces to $ψ_2$, then the output set of $ψ_2$ is a subset of the output set of $ψ_1$; (2) the difference between these output sets is made by ‘small’ elements (meaning that greater elements are available); and (3) these ‘small’ elements can safely be removed from the output set without altering its least upper-bound. This idea will be formalized by means of the following lemma’s.

The first two lemma’s state that the execution and output sets of a reduct are subsets of the execution and output sets of their originals:

\textbf{Lemma 8.8:1: (inclusion of execution sets)}

$∀ψ_1 ∈ Ψ^{\text{run}}∀ψ_2 ∈ \text{Execute}(ψ_1)[\text{Execute}(ψ_2) ⊆ \text{Execute}(ψ_1)]$

\textbf{Proof:}

Assume $ψ_1 = (\text{run \ e_1 with } ψ) ∈ Ψ^{\text{run}}$.

Assume $ψ_2 ∈ \text{Execute}(\text{run \ e_1 with } ψ)$, which simplifies to $ψ_2 ∈ \{\text{run \ e with } ψ \mid e ∈ \text{Reducts}_ψ(e_1)\}$.

Assume thus that $ψ_2 = (\text{run \ e_2 with } ψ)$ for some $\{1\} e_2 ∈ \text{Reducts}_ψ(e_1)$. 

To prove: \(\text{Execute}(\text{run } e_2 \text{ with } \psi) \subseteq \text{Execute}(\text{run } e_1 \text{ with } \psi)\).

By expanding the applications of \(\text{Execute}\), this simplifies to:

\[\text{To prove: } \{\text{run } e \text{ with } \psi \mid e \in \text{Reducts}_{\psi}(e_2)\} \subseteq \{\text{run } e \text{ with } \psi \mid e \in \text{Reducts}_{\psi}(e_1)\} \subseteq \]

By means of pattern matching, this can be simplified to:

\[\text{To prove: } \text{Reducts}_{\psi}(e_2) \subseteq \text{Reducts}_{\psi}(e_1).\]

This is implied by assumption [1].

**Lemma 8.8.2:** (inclusion of output sets)

\[\forall \psi_1 \in \Psi^{\text{run}} \forall \psi_2 \in \text{Execute}(\psi_1) [\text{Output}(\psi_2) \subseteq \text{Output}(\psi_1)]\]

**Proof:**

Assume \(\psi_1 \in \Psi^{\text{run}}\) and [1] \(\psi_2 \in \text{Execute}(\psi_1)\).

**To prove:** \(\text{Output}(\psi_2) \subseteq \text{Output}(\psi_1)\).

By expanding the applications of \(\text{Output}\), this simplifies to:

\[\text{To prove: } \{\text{Observe}(\psi) \mid \psi \in \text{Execute}(\psi_2)\} \subseteq \{\text{Observe}(\psi) \mid \psi \in \text{Execute}(\psi_1)\}.\]

This follows trivially by applying Lemma 8.8.1 on assumption [1].

The third lemma states that for each element of \(\text{Output}(\psi_1)\) which is not an element of \(\text{Output}(\psi_2)\), there exists a greater element in \(\text{Output}(\psi_2)\):

**Lemma 8.8.3:** (difference in output sets)

\[\forall \psi_1 \in \Psi^{\text{run}} \forall \psi_2 \in \text{Execute}(\psi_1) \forall o_1 \in (\text{Output}(\psi_1) \setminus \text{Output}(\psi_2)) \exists o_2 \in \text{Output}(\psi_2) [o_1 \subseteq o_2]\]

**Proof:**

Assume \(\psi_1 = (\text{run } e_1 \text{ with } \psi) \in \Psi^{\text{run}}\).

Assume \(\psi_2 \in \text{Execute}(\text{run } e_1 \text{ with } \psi)\), which simplifies to \(\psi_2 \in \{\text{run } e \text{ with } \psi \mid e \in \text{Reducts}_{\psi}(e_1)\}\).

Assume thus that \(\psi_2 = (\text{run } e_2 \text{ with } \psi)\) for some [1] \(e_2 \in \text{Reducts}_{\psi}(e_1)\).

Assume \(o_1 \in \text{Output}(\text{run } e_1 \text{ with } \psi) \cap \text{Output}(\text{run } e_2 \text{ with } \psi)\), therefore \(o_1 \in \{\text{Observe}(e) \mid e \in \text{Reducts}_{\psi}(e_1) \setminus \text{Reducts}_{\psi}(e_2)\}\).

Assume thus that \(o_1 = \text{Observe}(e_3)\) for some [2] \(e_3 \in \text{Reducts}_{\psi}(e_1)\) and [3] \(e_3 \notin \text{Reducts}_{\psi}(e_2)\).

**To prove:** \(\exists o_2 \in \text{Output}(\text{run } e_2 \text{ with } \psi) [\text{Observe}(e_3) \subseteq o_2]\).

Expanding the application of \(\text{Output}\), this simplifies to:

**To prove:** \(\exists o_2 \in \{\text{Observe}(e) \mid e \in \text{Reducts}_{\psi}(e_2)\} [\text{Observe}(e_3) \subseteq o_2]\).

By means of pattern matching, this can be simplified to:

**To prove:** \(\exists e \in \text{Reducts}_{\psi}(e_2) [\text{Observe}(e_3) \subseteq \text{Observe}(e)]\).

Apply confluence of reduction on [1] and [2] to obtain expressions \(e'_2\) and \(e'_3\) such that [4] \(e'_2 \in \text{Reducts}_{\psi}(e_2)\), [5] \(e'_3 \in \text{Reducts}_{\psi}(e_3)\) and [6] \(e'_2 =_{\alpha} e'_3\).

Now, choose \(e'_2\) as witness for \(e \in \text{Reducts}_{\psi}(e_2)\):

**To prove:** \(\text{Observe}(e_3) \subseteq \text{Observe}(e'_2)\).

Observation yields the same result for \(\alpha\)-equal expressions. Using assumption [6], \(\text{Observe}(e'_2)\) can thus safely be replaced by \(\text{Observe}(e'_3)\):  

**To prove:** \(\text{Observe}(e_3) \subseteq \text{Observe}(e'_3)\).

This is proved by repeatedly applying Observation 8.6.3:9(2) on assumption [5].
An element of a superset \((a \in A, B \subseteq A)\) is called **minimal** if there is at least one element of \(B\) which is greater \((b \in B, a < b)\). The fourth lemma now states that if \(A\) and \(B\) are sets of expression outputs (presumably obtained by means of observation) and all elements of \(A \setminus B\) are minimal, then \(A\) and \(B\) are equivalent.

**Lemma 8.8.4:** (remove minimal elements from an output set)

\[
\forall O_1, O_2 \subseteq E^{\text{output}} [\ O_2 \subseteq O_1 \Rightarrow \forall o_1 \in (O_1 \setminus O_2) \exists o_2 \in O_2 [o_1 \subseteq o_2] \Rightarrow O_1 \sim O_2 \ ]
\]

**Proof:**

Assume \(O_1, O_2 \subseteq E^{\text{output}}.\) Assume [1] \(O_2 \subseteq O_1\) and [2] \(\forall o_1 \in (O_1 \setminus O_2) \exists o_2 \in O_2 [o_1 \subseteq o_2].\)

To prove: \(O_1 \sim O_2.\)

Expanding the application of \(\sim\) on output sets, this simplifies to:

**To prove:** \(\forall o_1 \in O_1, \exists o_2 \in O_2 [o_1 \subseteq o_2] \land \forall o_2 \in O_2, \exists o_1 \in O_1 [o_2 \subseteq o_1].\)

The right-hand-side of this conjunction is trivial: first, assume \(o_2 \in O_2;\) then choose \(o_1 \in O_1\) equal to \(o_2\) (possible by assumption [1]) and then use the reflexivity of \(\subseteq\) (see Observation 8.6.3 8.9(1a)) to ensure \(o_2 \subseteq o_2.\)

**To prove:** \(\forall o_1 \in (O_1 \setminus O_2) \exists o_1 \in O_2 [o_1 \subseteq o_2].\)

Using the same reflexivity argument, this is trivial for all \(o_1 \in O_2.\)

It thus suffices to prove the statement above for \(o_1 \in (O_1 \setminus O_2)\) alone:

**To prove:** \(\forall o_1 \in (O_1 \setminus O_2) \exists o_2 \in O_2 [o_1 \subseteq o_2].\)

This is simply assumption [2].

The reducibility of executable total programs is now a trivial consequence of these four lemma’s and can therefore be proved easily:

**Theorem 8.8.5:** (reducibility of executable programs)

\[
\forall \psi_1 \in \Psi^{\text{run}} \forall \psi_2 \in \text{Execute}(\psi_1) [\text{Output}(\psi_1) \sim \text{Output}(\psi_2)]
\]

**Proof:**

Assume \(\psi_1 \in \Psi^{\text{run}}\) and \(\psi_2 \in \text{Execute}(\psi_1).\)

To prove: \(\text{Output}(\psi_1) \sim \text{Output}(\psi_2).\)

This is proved immediately by applying Lemma 8.8.4, using Lemma’s 8.8.2 and 8.8.3 to prove its conditions.

The reducibility of expressions can now be formally proved as well, translating it to the reducibility of (sets of) executable programs:

**Theorem 8.8.6:** (reducibility of expressions)

\[
\forall \psi \in \Psi \forall e_1, e_2 \in E^{\text{closed}} [e_2 \in \text{Reducts}_\psi (e_1) \Rightarrow \text{Equal}_\psi(e_1, e_2)]
\]

**Proof:**

Assume \(\psi \in \Psi\) and \(e_1, e_2 \in E^{\text{closed}}.\) Assume [1] \(e_2 \in \text{Reducts}_\psi(e_1).\)

To prove: \(\text{Equal}_\psi(e_1, e_2).\)

Expanding the application of \(\text{Equal},\) this simplifies to:

**To prove:** \(\forall x \in V_e \forall e \in E^{\text{opened}}(x) [\text{Output} (\text{run } e \rightarrow e_1\text{ with } \psi) \sim \text{Output} (\text{run } e \rightarrow e_2\text{ with } \psi)].\)

Assume \(x \in V_e\) and \(e \in E^{\text{opened}}(x).\)
To prove: \( \text{Output}(\operatorname{run} e_{x \rightarrow e_1} \text{ with } \psi) \sim \text{Output}(\operatorname{run} e_{x \rightarrow e_2} \text{ with } \psi). \)

Apply Theorem 8.8.5.

To prove: \( (\operatorname{run} e_{x \rightarrow e_2} \text{ with } \psi) \in \operatorname{Execute}(\operatorname{run} e_{x \rightarrow e_1} \text{ with } \psi). \)

Expanding the application of \( \operatorname{Execute} \), this simplifies to:

To prove: \( (\operatorname{run} e_{x \rightarrow e_2} \text{ with } \psi) \in \{ \operatorname{run} e' \text{ with } \psi \mid e' \in \text{Reducts}_\psi(e_{x \rightarrow e_1}) \}. \)

By means of pattern matching, this can be further simplified to:

To prove: \( e_{x \rightarrow e_2} \in \text{Reducts}_\psi(e_{x \rightarrow e_1}). \)

By expanding the application of \( \text{Reducts} \), this can be simplified to:

To prove: \( e_{x \rightarrow e_1} \vdash e_{x \rightarrow e_2}. \)

This is proved by assumption [1].

**Corollaries 8.8.7:** (make use of reducibility)

1. \( \forall e_1, e_2, e_3 \in E_{\text{closed}} \left[ \text{Equal}_\psi(e_1, e_2) \Rightarrow e_3 \in \text{Reducts}_\psi(e_2) \Rightarrow \text{Equal}_\psi(e_1, e_3) \right] \)
2. \( \forall e_1, e_2, e_3 \in E_{\text{closed}} \left[ \text{Equal}_\psi(e_1, e_2) \Rightarrow e_3 \in \text{Reducts}_\psi(e_1) \Rightarrow \text{Equal}_\psi(e_3, e_2) \right] \)
3. \( \forall e_1, e_2, e_3 \in E_{\text{closed}} \left[ \text{Equal}_\psi(e_1, e_2) \Rightarrow e_2 \in \text{Reducts}_\psi(e_3) \Rightarrow \text{Equal}_\psi(e_1, e_3) \right] \)
4. \( \forall e_1, e_2, e_3 \in E_{\text{closed}} \left[ \text{Equal}_\psi(e_1, e_2) \Rightarrow e_1 \in \text{Reducts}_\psi(e_3) \Rightarrow \text{Equal}_\psi(e_3, e_2) \right] \)

### 8.9 Conclusions

In this chapter, a fully formalized semantics for expression equality has been defined. This equality is complete and is able to deal with all kinds of expressions that are allowed in Core-CLEAN, including cyclic expressions and arbitrary non-terminating expressions, whether they are productive or not.

Our definition is completely operational and is formulated entirely in terms of the observable behavior of program reduction. Expressions are first embedded into programs, and then the execution outputs that these programs produce are compared. This concept is straightforward to understand, because it makes use of visible program output. An additional advantage is that partial applications are supplied with sufficient arguments automatically. Because of the need for program embeddings and explicit observations, however, the definitions themselves become slightly more complicated.

The definition of our equality does not make use of infinite expressions or least-upper-bounds in any way. Instead, an approach similar to bi-simulation is used, and it is required that all finite reduction results are related.

We have proved several interesting properties of our expression equality, including reducibility, which states that equality is invariant under reduction. In chapter 8 of the formal framework, referential transparency of our equality is proved as well. The proof is rather straightforward and can be looked up in [dvP07a]. It was not included in this chapter because it is formulated on the proposition level.
Chapter 9

**Abstract.** We present a proof rule and an effective tactic for proving properties about Haskell type classes by proving them for the available instance definitions. This is not straightforward, because instance definitions may depend on each other. The proof assistant Isabelle handles this problem for single parameter type classes by structural induction on types. However, this does not suffice for an effective tactic for more complex forms of overloading. We solve this using an induction scheme derived from the instance definitions. The tactic based on this rule is implemented in the proof assistant Sparkle.

9.1 Introduction

It is often stated that formulating properties about programs increases robustness and safety, especially when formal reasoning is used to prove these properties. Robustness and safety are becoming increasingly important considering the current dependence of society on technology. Research on formal reasoning has spawned many general purpose proof assistants, such as Coq [The06], Isabelle [NPW02], and Pvs [OSRS01]. Unfortunately, these general purpose tools are geared towards mathematicians and are hard to use when applied to more practical domains such as actual programming languages.

Because of this, proof assistants have been developed that are geared towards specific programming languages. This allows proofs to be conducted on the source program using specifically designed proof rules. Functional languages are especially suited for formal reasoning because they are referentially transparent. Examples of proof assistants for functional languages are Evt [NFG01] for Erlang [AV91], Sparkle [dvP02] for Clean [Pv01], and Era [Win98] for Haskell [Pey03].
class Eq a where
  (==) :: a -> a -> Bool

instance Eq Int where
  x == y = predefinedeqint x y

instance Eq Char where
  x == y = predefinedeqchar x y

instance (Eq a) => Eq [a] where
  [] == [] = True
  (x:xs) == [] = False
  [] == (y:ys) = False
  (x:xs) == (y:ys) = x == y && xs == ys

Figure 9.1: A type class for equality in HASKELL

9.1.1 Type classes

A feature that is commonly found in functional programming languages is overloading structured by type classes [WB89]. Type classes essentially are groups of types, the class instances, for which certain operations, the class members, are implemented. These implementations are created from the available instance definitions and may be different for each instance. The type of an instance definition is called the instance head. The equality operator will be used as a running example throughout this paper (figure 9.1).

In the most basic case, type classes have only one parameter and instance heads are flat, that is, a single constructor applied to a list of type variables. Furthermore, no two instance definitions may overlap.

Several significant extensions have been proposed, such as multiple parameters [PJM97], overlapping instances, and instantiation with constructors [Jon93], that have useful applications such as collections, coercion, isomorphisms and mapping. In this paper, the term general type classes is used for systems of type classes that support these extensions and non-flat instance heads. Figure 9.2 shows a multi parameter class for the symmetric operation eq2.

An important observation regarding type classes is that, in general, the defined instances should be semantically related. For example, all instances of the equality operator usually implement an equivalence relation. These properties can be proven for all instances at once by proving them for the available instance definitions. Unfortunately, this is not straightforward because the instance definitions may depend on each other and hence so will the proofs. For example, equality on lists is only symmetric if equality on the list members is so as well.
Section 9.1.2: Contributions

The only proof assistant with special support for overloading that we know of is ISABELLE [Nip93, Wen97], which essentially supports single parameter type classes and a proof rule for it based on structural induction on types. However, we show that for general type classes, an effective tactic is not easily derived when structural induction is used. We use an induction scheme on types based on the instance definitions to solve this problem. Using this induction scheme, a proof rule and tactic are defined that are both strong enough and effective.

As a proof of concept, we have implemented the tactic in the proof assistant SPARKLE for the programming language CLEAN. The results, however, are generally applicable and can, for example, also be used for HASKELL and ISABELLE, if ISABELLE would support the specification of general type classes. In fact, the examples here are presented using HASKELL syntax. SPARKLE is dedicated to CLEAN, but can also be used to prove properties about HASKELL programs by translating them to CLEAN using the HACLE translator [Nay04].

9.1.3 Outline

The rest of this paper is structured as follows. First, the proof assistant SPARKLE is presented (section 9.2). Then, basic definitions for instance definitions, evidence values, and class constrained properties are introduced (section 9.3). After showing why structural induction does not suffice (section 9.4), the proof rule and tactic based on the instance definitions are defined (section 9.5) and extended to multiple class constraints (section 9.6). We end with a discussion of the implementation (section 9.7), related and future work (section 9.8), and a summary of the results (section 9.9).
9.2 Sparkle

The need for this work arose whilst improving the proof support for type classes in Sparkle. Sparkle is a proof assistant specifically geared towards Clean, which means that it can reason about Clean concepts using rules based on Clean’s semantics. Properties are specified in a first order predicate logic extended with equality on expressions. An example of this, using a slightly simplified syntax, is:

\[ \forall n: \text{Int} \neq \bot \forall a \forall xs:a \left[ \text{take } n \hspace{1em} \text{drop } n \hspace{1em} \text{++ } \text{xs } = \text{xs} \right] \]

These properties can be proven using tactics, which are user friendly operations that transform a property into a number of logically stronger properties, the proof obligations or goals, that are easier to prove. A tactic is the implementation of (a combination of) theoretically sound proof rules. Whereas in general a proof rule is theoretically simple but not very prover friendly, a tactic is prover friendly but often theoretically more complex. The proof is complete when all remaining proof obligations are trivial. Some useful tactics are, for example, reduction of expressions, induction on expression variables, and rewriting using hypotheses.

In Sparkle, properties that contain member functions can only be proven for specific instances of that function. For example:

\[ \text{sym}_{\text{Int}}: \forall x: \text{Int} \forall y: \text{Int} \left[ x = y \rightarrow y = x \right] \]

can be easily proven by induction on lists using symmetry of equality on integers. Proving that something holds for all instances, however, is not possible in general. Consider for example symmetry of equality:

\[ \text{sym}: \forall a[\text{Eq} :: a \Rightarrow \forall x:a \forall y:x \left[ x = y \rightarrow y = x \right]] \]

where \( \text{Eq} :: a \) denotes the, previously not available, constraint that equality must be defined for type \( a \). This property can be split into a property for every instance definition, which gives among others the property for the instance for lists:

\[ \text{sym}_{[a]}: \forall a[\text{Eq} :: a \Rightarrow \forall x:a \forall y:a \left[ x = y \rightarrow y = x \right]] \]

It is clear that this property is true as long as it is true for instance \( a \). Unfortunately, this hypothesis is not available. Using an approach based on induction, however, we may be able to assume the hypotheses for all instances the instance definition depends on, and hence will be able to prove the property.

Internally, Sparkle translates type classes to evidence values or dictionaries [WB89], that make the use of overloading explicit. The evidence value for a class constraint \( c :: a \) is the evidence that there is an (implementation of the) instance of class \( c \) for type \( a \). Hence, an evidence value exists if and only if the class constraint is satisfied. As usual, we will use the implementation itself as the
Section 9.3: Preliminaries

A program is translated by converting all instance definitions to functions (distinct names are created by suffixes). In expressions, the evidence value is substituted for member applications. When functions require certain classes to be defined, the evidence values for these constraints are passed as a parameter. Figure 9.3 shows an example of the result of the translation of the equality class from figure 9.1.

9.3 Preliminaries

Instead of defining a proof rule that operates on the example properties from section 9.2, we define both instances and properties at the level that explicitly uses evidence values. In this section, basic definitions for instance definitions, evidence values, and class constrained properties are given.

9.3.1 Instance definitions

Because we intend to support constructor classes, types are formalized by a language of constructors [Jon93]:

\[ \tau ::= \alpha \mid \mathcal{X} \mid \tau \tau' \]

where \( \alpha \) and \( \mathcal{X} \) range over a given set of type variables and type constructors respectively. For example, \( \tau \) can be \( \text{Int} \), \( \text{[Int]} \), and \( \text{Tree Char} \), but also the \( [\] \), \( \text{Tree} \), and \( \rightarrow \) constructors that take types as an argument and yield a list, tree, or function type respectively. \( TV(\tau) \) denotes the set of type variables occurring in \( \tau \). The set of closed types \( T^c(\tau) \) is the set of types for which \( TV(\tau) \) is empty.

Predicates are used to indicate that an instance of a certain class exists. An instance can be identified by an instantiation of the class parameters. The predicate \( c :: \bar{\tau} \) denotes that there is an instance of the class \( c \) for instantiation \( \bar{\tau} \) of the class parameters. For example, \( \text{Eq} :: \text{[Int]} \) and \( \text{Eq} :: (\text{Int}, \text{Int}) \) denote that there is an instance of the \( \text{Eq} \) class for types \( \text{[Int]} \) and \( (\text{Int}, \text{Int}) \).
respectively:

\[ \pi ::= c :: \bar{\tau} \]

Because these predicates are used to constrain types to a certain class, they are called class constraints. Class constraints in which only type variables occur in the type, for example \( \text{Eq} ::= a \), are called simple. For reasons of simplicity, it is assumed that all type variables that occur in a class constraint are distinct.

Without loss of generality, throughout this paper we restrict ourselves to type classes that have only one member and no subclasses. Multiple members and subclasses can be supported using records of expressions for the evidence values. An instance definition:

\[ \text{inst } \bar{\pi} \Rightarrow c :: \bar{\tau} = e \]

defines an instance \( \bar{\tau} \) of class \( c \) for types that satisfy class constraints \( \bar{\pi} \). The instance definition provides the translated expression \( e \) for the class member \( c \).

The functions \( \text{Head}(\text{inst } c :: \bar{\pi} \Rightarrow \tau = e) = \tau \) and \( \text{Context}(\text{inst } c :: \bar{\pi} \Rightarrow \tau = e) = \bar{\pi} \) will be used to retrieve the instance head and context respectively.

The program context \( \psi \), that contains the function and class definitions, also includes the available instance definitions. The function \( \text{Idefs}_\psi(c) \) returns the set of instance definitions of class \( c \) defined in program \( \psi \).

### 9.3.2 Evidence values

From the translation from type classes to evidence values, as briefly summarized in section 9.2, the rule for evidence creation is important for our purpose. Two definitions are required before it can be defined.

Firstly, because instance definitions are allowed to overlap, a mechanism is needed that chooses between them. Since the exact definition is not important for our purpose, we assume that the function \( \text{Ai}_\psi(c :: \bar{\tau}) \) determines the most specific instance definition applicable for instance \( \bar{\tau} \) of class \( c \). \( \text{Ai}_\psi \) is also defined for types that contain variables as long as it can be determined which instance definition should be applied.

Secondly, the dependencies of an instance are the instances it depends on:

\[ \text{Deps}(c :: \bar{\tau}, i) = *_{\text{Head}(i) \rightarrow \bar{\tau}}(\text{Context}(i)) \]

where \( *_{\bar{\tau} \rightarrow \bar{\tau}'} \) denotes the substitutor that maps the type variables in \( \bar{\tau} \) such that \( *_{\bar{\tau}} = \bar{\tau}' \). When \( i \) is not provided, \( \text{Ai}_\psi(c :: \bar{\tau}) \) is assumed for it.

Evidence values are now straightforwardly created by applying the expression of the most specific instance definition to the evidence values of its dependencies:

\[
\text{Deps}(\pi) = (\pi_1, \ldots, \pi_n) \\
\text{Ai}_\psi(\pi) = \text{inst } c :: \bar{\tau}' \Rightarrow \bar{\tau}' = e \\
\text{Ev}_\psi(\pi) = e \text{ Ev}_\psi(\pi_1) \ldots \text{ Ev}_\psi(\pi_n)
\]
In proofs, evidence values will be created assuming the evidence values for the dependencies are already assigned to expression variables:

\[ \text{Deps}(\pi, i) = \langle \pi_1, \ldots, \pi_n \rangle \]
\[ i = \text{inst } c :: \bar{\tau} \Rightarrow \bar{\tau}' = e \]
\[ \text{Ev}_\psi(\pi, i) = e \text{ ev}_{\pi_1} \ldots \text{ ev}_{\pi_n} \]

assuming that the evidence for \( \pi \) is assigned to the variable \( \text{ev}_\pi \). A specific instance definition \( i \) can be provided, because \( A_i(\pi) \) might not be known in proofs.

### 9.3.3 Class constrained properties

In Sparkle, properties are formalized by a first order predicate logic extended with equality on expressions. The equality on expressions is designed to handle infinite and undefined expressions well.

We extend these properties with class constraints, that can be used to constrain types to a certain class. These properties will be referred to as class constrained properties. For example, consider symmetry and transitivity of equality:

sym: \[ \forall a [\text{Eq} :: a \Rightarrow \forall x,y:a [\text{ev}_{\text{Eq} :: a} x \rightarrow \text{ev}_{\text{Eq} :: a} y x]] \]

trans: \[ \forall a [\text{Eq} :: a \Rightarrow \forall x,y,z:a [\text{ev}_{\text{Eq} :: a} x \rightarrow \text{ev}_{\text{Eq} :: a} y z \rightarrow \text{ev}_{\text{Eq} :: a} x z]] \]

The property \( c :: \bar{\tau} \Rightarrow p \) means that in property \( p \) it is assumed that \( \bar{\tau} \) is an instance of class \( c \) and the evidence value for this class constraint is assigned to \( \text{ev}_{c :: \bar{\tau}} \). Thus, the semantics of the property \( \pi \Rightarrow p \) is defined as \( p[\text{ev}_{\pi} \mapsto \text{ev}_\psi(\pi)] \).

### 9.4 Structural induction

The approach for proving properties that contain overloaded identifiers taken in Isabelle essentially is structural induction on types. In this section it is argued that the proof rule for general type classes should use another induction scheme.

Structural induction on types seems an effective approach because it gives more information about the type of an evidence value. This information can be used to expand evidence values. For example, \( \text{ev}_{\text{Eq} :: a} \) can be expanded to eqlist \( \text{ev}_{\text{Eq} :: a} \) (see figure 9.3).

\[ A_{i\psi}(\pi) = i \]
\[ \forall TV(\pi) [\text{Deps}(\pi) \Rightarrow p(\text{Ev}_\psi(\pi))]] \]
\[ \forall TV(\pi) [\pi \Rightarrow p(\text{ev}_\pi)]] \]

More importantly, structural induction allows the property to be assumed for structurally smaller types. Ideally the hypothesis should be assumed for all
dependencies on the same class. Unfortunately, structural induction does not always allow this for multi parameter classes.

Consider for example the multi parameter class in figure 9.2. The instance of Eq2 for [Int] (Char, Char) depends on the instance for Char Int, which is not structurally smaller because Char is not structurally smaller than [Int], and Int is not structurally smaller than (Char, Char). Hence, the hypothesis cannot be assumed for this dependency. This problem can be solved by basing the induction scheme on the instance definitions.

9.5 Induction on instances

The induction scheme proposed in the previous section can be used on the set of defined instances of a class. In this section, a proof rule and tactic that use this scheme are defined and applied to some examples.

9.5.1 Proof rule and tactic

We first define the set of instances of a class and an order based on the instance definitions on it. The well-founded induction theorem applied to the defined set and order yields the proof rule. Then, the tactic is presented that can be derived from this rule.

Remember that the instances of a class are identified by sequences of closed types. \( \bar{\tau} \) is an instance of class \( c \) if an evidence value can be generated for the class constraint \( c :: \bar{\tau} \). Hence, the set of instances of class \( c \) can be defined as:

\[
\text{Inst}_\psi(c) = \{ \bar{\tau} | \forall c' :: \bar{\tau}' \in \text{Deps}(c :: \bar{\tau}) [\bar{\tau}' \in \text{Inst}_\psi(c')] \}
\]

For example, \( \text{Inst}_\psi(\text{Eq}) = \{ \text{Int}, \text{Char}, [\text{Int}], [\text{Char}], [[\text{Int}]], \ldots \} \).

An order on this set is straightforwardly defined. Because the idea is to base the order on the instance definitions, an instance \( \bar{\tau}' \) is considered one step smaller than \( \bar{\tau} \) if the evidence for \( \bar{\tau} \) depends on the evidence for \( \bar{\tau}' \), that is, if \( c :: \bar{\tau}' \) is a dependency of the most specific instance definition for \( c :: \bar{\tau} \). For example, \( \text{Int} <^1_{(\psi,\text{Eq})} [\text{Int}] \) and \( [\text{Char}] <^1_{(\psi,\text{Eq})} [[\text{Char}]] \).

\[
\bar{\tau} <^1_{(\psi,c)} \bar{\tau}' \Leftrightarrow c :: \bar{\tau}' \in \text{Deps}(c :: \bar{\tau})
\]

Note that there is a specific set of instances for each class and therefore also a specific order for each class.

Well-founded induction requires a well-founded partial order, for which we use the reflexive transitive closure of \( <^1_{(\psi,c)} \). It can be easily derived from the way evidence values are generated that this is indeed a well-founded partial order. Applying this order, \( \leq_{(\psi,c)} \), to the well-founded induction theorem yields the following proof rule:

\[
\forall \bar{\tau} \in \text{Inst}_\psi(c) [\forall \bar{\tau}' \leq_{(\psi,c)} \bar{\tau} [p(\bar{\tau}')] \rightarrow p(\bar{\tau})] \quad \forall \bar{\alpha} \in \text{Inst}_\psi(c) [p(\bar{\alpha})] \quad \text{(inst-rule)}
\]
Section 9.5.2: Results

Rewriting the proof rule using the definitions of $\text{Inst}_\psi(c)$, $\leq_{(\psi,c)}$, evidence creation, and class constrained properties results in a tactic that can be directly applied to class constrained properties. For all class constraints $c :: \bar{\alpha}$:

$$
\forall i \in \text{Idefs}_\psi(c) \forall \text{Head}(i) \in (\bar{T}) [\text{Dep}(c :: \text{Head}(i), i) \Rightarrow \forall c' :: \bar{T} \in \text{Dep}(c :: \text{Head}(i), i)[c = c' \Rightarrow p(\text{ev}_{c :: \bar{T}}, \bar{T})] \\
\Rightarrow \forall \bar{\alpha} :: \bar{T} \in \text{Dep}(c :: \bar{\alpha}, i)[c :: \bar{\alpha} \Rightarrow p(\text{ev}_{c :: \bar{\alpha}}, \bar{\alpha})] (\text{inst-tactic})
$$

where it is assumed that all variables in $\text{Head}(i)$ are fresh. When the tactic is applied to a class constrained property, it generates a proof obligation for every available instance definition with hypotheses for all dependencies on the same class.

9.5.2 Results

The result is both a proof rule and a user friendly tactic. The tactic is nicely illustrated by symmetry of equality (figure 9.1 and 9.3). When (inst-tactic) is applied to:

$$
\text{sym: } \forall a [\text{Eq} :: a \Rightarrow \forall x, y :: [\text{ev}_{\text{Eq} :: a} x y \rightarrow \text{ev}_{\text{Eq} :: a} y x]]
$$

it generates the following three proof obligations (one for each instance definition):

$$
\text{sym}_{\text{Int}:} \forall x :: \text{Int} \forall y :: [\text{eqint} x y \rightarrow \text{eqint} y x]
$$

$$
\text{sym}_{\text{Char}:} \forall x :: \text{Char} \forall y :: [\text{eqchar} x y \rightarrow \text{eqchar} y x]
$$

$$
\text{sym}_{[a]}: \forall a [\text{Eq} :: a \Rightarrow \forall x, y :: [\text{ev}_{\text{Eq} :: a} x y \rightarrow \text{ev}_{\text{Eq} :: a} y x]]
$$

$$
\Rightarrow \forall x, y :: [\text{eqlist} \text{ev}_{\text{Eq} :: a} x y \rightarrow \text{eqlist} \text{ev}_{\text{Eq} :: a} y x]
$$

which are easily proven using the already available tactics.

The previous step could also have been taken using a tactic based on structural induction on types. However, (inst-tactic) can also assume hypotheses for dependencies that are possibly not structurally smaller. Consider for example the symmetry of $\text{eq2}$ in figure 9.2:

$$
\text{sym}_{2:} \forall a, b [\text{Eq2} :: a b \Rightarrow \text{Eq2} :: b a \\
\Rightarrow \forall x, y :: [\text{ev}_{\text{Eq2} :: a} x y \rightarrow \text{ev}_{\text{Eq2} :: b} a b x y]]
$$
Applying \texttt{(inst-tactic)} to this property generates a proof obligation for every instance definition, including one for the fourth instance of Eq2 in figure 9.2, where \texttt{eq2list} is the translation of that instance definition:

\[
\text{sym2}_{[a]} : \forall a, b, c \exists \ E q 2 :: b \ a \Rightarrow E q 2 :: c \ a

\Rightarrow [E q 2 :: a \ b \Rightarrow \forall x : b \ [e v _{E q 2 :: b} \ a \ x \ y \Rightarrow e v _{E q 2 :: a} \ b \ y \ x]]

\Rightarrow [E q 2 :: a \ c \Rightarrow \forall x : c \ [e v _{E q 2 :: c} \ a \ x \ y \Rightarrow e v _{E q 2 :: a} \ c \ y \ x]]

\Rightarrow E q 2 :: (b, c) [a] \Rightarrow \forall x : [a] \ [e v _{E q 2 :: (b, c)} \ [e v _{E q 2 :: b} \ a \ x \ y \Rightarrow e v _{E q 2 :: c} \ a \ y \ x

[eq2list ev_{E q 2 :: b} \ a \ ev_{E q 2 :: c} \ a \ x \ y \Rightarrow e v _{E q 2 :: (b, c)} \ [a] \ y \ x]]
\]

In this proof obligation, the hypotheses could not have been assumed when using structural induction on types (see section 9.4), hence our tactic is useful in more cases.

\section*{9.6 Multiple class constraints}

The proof rule and tactic presented in the previous section work well when the property has only one class constraint. In case of multiple class constraints, however, the rules might not be powerful enough. In this section it is shown that this problem does indeed occur. Therefore, a more general proof rule and tactic are defined and applied to some examples.

\subsection*{The problem}

Consider the two class definitions in figure 9.4. The translated instance definitions are respectively called \texttt{fint}, \texttt{flist}, \texttt{ftree}, \texttt{gint}, \texttt{gtree}, and \texttt{glist} at the level of dictionaries. Given the property:

\[
\text{same} : \forall a [f :: a \Rightarrow g :: a \Rightarrow [e v _{f :: a} \ x = e v _{g :: a} \ x]]
\]

Applying \texttt{(inst-tactic)} yields among others the goal:

\[
\text{same}_{[a]}: \forall a [g :: [a] \Rightarrow \forall x : [a] [f l i s t e v _{g :: a} \ x = e v _{g :: a} \ x]]
\]

This goal has a non-simple class constraint, which can only be removed by evidence expansion \texttt{(expand)}, resulting in:

\[
\text{same}_{[a]}^{*} : \forall a [f :: a \Rightarrow g :: a \Rightarrow \forall x : [a] [f l i s t e v _{g :: a} \ x = g l i s t e v _{f :: a} \ ev _{g :: a} \ x]]
\]

After some reduction steps, this can be transformed into:
Section 9.6.1: Proof rule and tactic

We take the same approach as in the previous section. We first define the set of instances, the order, the proof rule and the tactic. Then, in section 9.6.2, it is shown that the new tactic solves the problem.

First, the set of type sequences that are instances of all classes that occur in a list of class constraints is defined. $\bar{\tau}$ is a member of the set if all class constraints

\[
\begin{align*}
\text{data Tree a = Leaf | Node a (Tree a) (Tree a)} & \\
\text{class f a where f :: a -> Bool} & \\
\text{instance f Int where} & \\
\text{f x = x == x} & \\
\text{instance (g a) => f [a] where} & \\
f [] = True & \\
f (x:xs) = g x == g x & \\
\text{instance (f a, g a) => f (Tree a) where} & \\
f Leaf = True & \\
f (Node x l r) = f x == g x & \\
\text{class g a where g :: a -> Bool} & \\
\text{instance g Int where} & \\
g x = x == x & \\
\text{instance (f a) => g (Tree a) where} & \\
g Leaf = True & \\
g (Node x l r) = f x == f x & \\
\text{instance (g a, f a) => g [a] where} & \\
g [] = True & \\
g (x:xs) = g x == f x
\end{align*}
\]

Figure 9.4: Problematic class definitions

This proof obligation is true when $\text{ev}_{f::a} x = \text{ev}_{g::a} x$. Unfortunately, the induction scheme did not allow us to assume this hypothesis. Since this problem is caused by the fact that the type variables occur in more than one class constraint, the natural solution is to take multiple class constraints into account in the induction scheme.

9.6.1 Proof rule and tactic

We take the same approach as in the previous section. We first define the set of instances, the order, the proof rule and the tactic. Then, in section 9.6.2, it is shown that the new tactic solves the problem.

First, the set of type sequences that are instances of all classes that occur in a list of class constraints is defined. $\bar{\tau}$ is a member of the set if all class constraints
π are satisfied when all variables TV(π) are replaced by the corresponding type from ɹ. We assume here that TV(π) is a linearly ordered, for example lexicographically, sequence and that the elements of ɹ are in the corresponding order. For example, SetInstψ( f :: a, g :: a) = {Int, [Int], Tree Int, [[Int]], ...}.

SetInstψ(π) = { ɹ | ∀a′ ∈ [TV(π) → ɹ(α′) ∈ Instψ(α)]}

The order on this set is an extension of the order for single class constraints to sets. A sequence of types ɹ is considered one step smaller than ɹ′ if TV(ɹ) → TV(ɹ′) is a subset of the dependencies of TV(ɹ) → TV(ɹ′). For example, [Int] <₁(ψ, f :: a, g :: a) [[[Int]]] because {f :: [Int], g :: [[Int]]} is a subset of Deps(g :: [[Int]]) ∪ Deps(f :: [[Int]]). Here, sequences of class constraints are lifted to sets when required:

\[ ɹ <₁ ψ, ɹ′ \iff TV(ɹ) \subseteq TV(ɹ′) \]

Again, it can be derived from the evidence creation that the reflexive transitive closure of this order, ≤₁(ψ, ɹ), is a well-founded partial order.

Applying the well-founded induction theorem to this set and order yields the proof rule for multiple class constraints. For every sequence of simple class constraints ɹ:

\[
\forall ɹ ∈ SetInstψ(ɹ) \left[ \forall ɹ′ ≤₁ ψ, ɹ \rightarrow p(ɹ′) \rightarrow p(ɹ) \right] \quad \text{(multi-rule)}
\]

Because multiple class constraints are involved, defining the final tactic is a bit more complicated. Instead of all instance definitions, every combination of instance definitions, one for each class constraint, has to be tried. All of these instance definitions make assumptions about the types of the type variables, and these assumptions should be unifiable. Therefore, we define the most general unifier that takes the sharing of type variables across class constraints into account:

\[ SetMgu(c_1 :: α_1, ... , c_n :: α_n, ɹ_1, ..., ɹ_n) = * \iff \forall 1 ≤ i ≤ n [\exists *′ ψ ′(α_i') = ɹ_i] \land \forall *′′ [\exists *″ ψ″(α_i″) = ɹ_i] \Rightarrow \exists *′′′ [ψ′ = ψ″ \circ *] \]

Furthermore, for readability of the final tactic, some straightforward extensions of existing definitions to vectors are used:

\[
\begin{align*}
Idefs_ψ(ɹ_1, ..., ɹ_n) & = \{i_1, ..., i_n \mid i_j ∈ Idefs_ψ(ɹ_j)\} \\
Head(ɹ_1, ..., ɹ_n) & = \langle Head(ɹ_1), ..., Head(ɹ_n)\rangle \\
Evψ(ɹ_1, ..., ɹ_n, ɹ_1, ..., ɹ_n) & = \{Evψ(ɹ_1, ɹ_1), ..., Evψ(ɹ_n, ɹ_n)\} \\
ev(ɹ_1, ..., ɹ_n) & = \{ev_1, ..., ev_n\} \\
Deps(ɹ_1, ..., ɹ_n, ɹ_1, ..., ɹ_n) & = \langle Deps(ɹ_1, ɹ_1), ..., Deps(ɹ_n, ɹ_n)\rangle
\end{align*}
\]

Finally, using the presented definitions, evidence creation, class constrained properties, and the proof rule, the tactic can be defined. For every sequence of
simple class constraints $\bar{\pi}$:

$$\forall \bar{t} \in \text{Defs}_x(\bar{\pi}) \exists s = \text{SetMgu}(\bar{\pi}, \text{Head}(\bar{t})) \forall (\text{Head}(\bar{t})) \in \langle T^* \rangle$$

$$[\text{Dep}(s(\bar{\pi}), \bar{t})$$

$$\Rightarrow \forall s'(\bar{\pi}) \subseteq \text{Dep}(s(\bar{\pi}), \bar{t}) [p(e_{s'}(\bar{\pi}), \bar{t})]$$

$$\rightarrow p(E_{s'}(s(\bar{\pi}), \bar{t}), s(\text{Head}(\bar{t})))$$

$$]$$

$$\forall TV(\bar{\pi}) [\bar{\pi} \Rightarrow p(e_{TV(\bar{\pi})})]$$

(multi-tactic)

Note that applying this tactic may result in non-simple class constraints when non-flat instance types are used. For non-simple class constraints, the induction tactics cannot be applied, but the (expand) rule might be used. However, in practice most instance definitions will have flat types.

This solution for multiple class constraints has some parallels to the constraint set satisfiability problem (CS-SAT), the problem of determining if there are types that satisfy a set of class constraints. The general CS-SAT problem is undecidable. However, recently, an algorithm was proposed [CFV04] that essentially tries to create a type that satisfies all constraints by trying all combinations of instance definitions, as we have been doing in our tactic.

### 9.6.2 Results

In this section, we have generalized the proof rule and tactic from section 9.5 to multiple class constraints. In case of a single class constraint, the new rules behave exactly the same as (inst-rule) and (inst-tactic). However, now we can apply (multi-tactic) to multiple class constraints at once. Given the previously problematic property:

**same**: $\forall a [f :: a \Rightarrow g :: a \Rightarrow [ev_{f::a} x = ev_{g::a} x]]$

this yields three proof obligations, one for every unifiable combination of instance definitions:

**same_int**: $\forall a [fint x = gint x]$

**same_a**: $\forall a [f :: a \Rightarrow g :: a \Rightarrow \forall x_a [ev_{f::a} x = ev_{g::a} x]$

$$\rightarrow \forall x :: [f :: a \Rightarrow ev_{f::a} x = ev_{g::a} x]$$

**same_tree_a**: $\forall a [f :: a \Rightarrow g :: a \Rightarrow \forall x_a [ev_{f::a} x = ev_{g::a} x]$

$$\rightarrow \forall x :: [f :: a \Rightarrow ev_{f::a} x = ev_{g::a} x]$$

The goal **same_a** (and **same_tree_a**) now has a hypothesis that can be used to prove the goal using the already available tactics. Hence, by taking multiple class constraints into account the problem is solved.
9.7 Implementation

As a proof of concept, we have implemented the (multi-tactic) tactic extended for multiple members and subclasses in SPARKLE. Because of the similarity to the already available induction tactic and the clearness of the code, the implementation of the tactic took very little time. However, to implement the tactic, the typing rules had to be extended. The translation of type classes to dictionaries is only typeable in general using rank-2 polymorphism, which is currently not supported by SPARKLE. This was worked around by handling the dictionary creation and selection in a way that hides the rank-2 polymorphism. Ideally, the use of dictionaries should be completely hidden from the user as well.

The tactic has been used to prove, amongst others, the examples in this paper. The implementation is available at:

http://www.student.kun.nl/ronvankesteren/SparkleGTC.zip

9.8 Related and future work

As mentioned in section 9.1, the general proof assistant ISABELLE [NPW02] supports overloading and single parameter type classes. ISABELLE’s notion of type classes is somewhat different from HASKELL’s in that it represents types that satisfy certain properties instead of types for which certain values are defined. Nevertheless, the problems to be solved are equivalent. ISABELLE [Nip93, Wen97] uses a proof rule based on structural induction on types, which suffices for the supported type classes. However, if ISABELLE would support more extensions, most importantly multi parameter classes, it would be useful to define our proof rule and a corresponding tactic in ISABELLE.

Essentially, the implementation of the tactic we proposed extends the induction techniques available in SPARKLE. Leonard Lensink proposed and implemented extensions of SPARKLE for induction and co-induction for mutually recursive functions and data types [Lv04]. The main goal was to ease proofs by making the induction scheme match the structure of the program. Together with this work this significantly increases the applicability of SPARKLE.

Because generics is often presented as an extension of type classes [HP00], it would be nice to extend this work to generics as well. Currently, in CLEAN generics are translated to normal type classes where classes are created for every available data type [AP01]. There is a library for HASKELL that generates classes with boilerplate code for every available data type [LP03]. The tactic presented here can already be used to prove properties about generic functions by working on these generated type classes. However, the property is only proven for the data types that are used in the program and a separate proof is required for each data type. That is, after all, the main difference between normal type classes and generics. Hence, it remains useful to define a proof rule specifically for generics.
9.9 Conclusion

In this paper, we have presented a proof rule for class constrained properties and an effective tactic based on it. Although structural induction on types is theoretically powerful enough, we showed that for an effective tactic an induction scheme should be used that is based on the instance definitions. The tactic is effective, because, using the defined proof rule, it allows all sensible hypotheses to be assumed. The rule and tactic were first defined for single class constraints and then generalized to properties with multiple class constraints.

As a proof of concept, the resulting tactic has been implemented in a customized version of SPARKLE for the programming language CLEAN, but it can also be used for proving properties about HASKELL programs. This is, to our knowledge, the first implementation of an effective tactic for general type classes. If ISABELLE would support extensions for type classes, the tactic could be implemented in ISABELLE as well.

Acknowledgements

We would like to thank Sjaak Smetsers for his suggestions and advice concerning this work, especially on the semantics of type classes in CLEAN, and Fermín Reig for a valuable discussion on generic proofs at the TFP2004 workshop.
Chapter 10

Peter Achten, Marko van Eekelen, Maarten de Mol, Rinus Plasmeijer:
A Common Arrow Based Semantics for GEC and iData Applications
Under submission to Journal of Functional Programming.

Abstract. State-based interactive applications, whether they run on the desktop or as a web application, can be considered as collections of interconnected editors of structured values that allow users to manipulate data. This is the view that is advocated by the GEC and iData toolkits, which offer a high level of abstraction to programming desktop and web GUI applications respectively. Special features of these toolkits are that editors have shared, persistent state, and that they handle events individually. In this paper we cast these toolkits within the Arrow framework and present a single, unified semantic model that defines shared state and event handling. We study the properties of this EditorArrow model, and of editors in particular. Furthermore, we present the definedness properties of the combinators. A reference implementation of the EditorArrow model is given with some small program examples. We discuss formal reasoning about the model using the proof assistant Sparkle.

10.1 Introduction

Building Graphical User Interfaces (GUI) is a labor intensive endeavor, whether they are being programmed based on a desktop widget set, or based on the web. Consider the effort of creating a frequently occurring application-user dialog, in which the user is required to enter a number of data items in order to advance the program. When programming for the desktop, the programmer needs to declare, create, manage, and eventually destroy widgets (at least one for each input element, and typically several to contain them and provide proper layout); for each widget several callback routines need to be programmed that implement both the behavior of the widget, and its effect to other widgets. Callback functions must terminate timely (the 1/10s rule) to provide the application user the impression that the application is sufficiently responsive to her actions.
When programming for the web, the programmer needs to create the proper HTML pages containing the forms that hold the input elements; the state of these elements needs to be managed by the programmer because of the stateless nature of the web; the communication, which is typically string based, between client browser and server application has to be programmed, and is untyped, which is a known source of errors. The code that computes the page needs to terminate sufficiently fast, otherwise the browser will give up. In both situations, the resulting code can easily result in hundreds of lines of code that is intricately interdependent. How can you convince yourself, or other stakeholders, that the program is correct with respect to its requirements? Ideally, one would like to prove (once and for all) that the program satisfies well stated properties in a formal, and computer supported, way. Unfortunately, even if the host programming language supports formal reasoning, neither the desktop nor the web has a formally specified reasoning model. Without some kind of underlying model one will have to resort to informal reasoning or to (model-based) testing. Model-based testing does not require a formal model of the implementation, but only a formal specification of the required properties. Due to automation, model based test systems can rapidly and repeatedly explore vast numbers of test scenarios, and generate reports when issues are being found. Model based testing can be an extremely valuable tool to increase the confidence in the correctness of an implementation, but it still does not provide a proof.

In this paper we create a common underlying formal model for a certain class of desktop and web programs making formal reasoning applicable for such programs. Reconsider the task of creating an application user dialog in which the user needs to provide several data items to the application. Another way of looking at this task is to consider the data items to be an instance of a structured data type, and to derive the corresponding GUI automatically from this data type. The derived GUI acts effectively as an editor of values of the given structured type. This reduces the programming effort to specifying a suitable structured data type, and invoking the derivation mechanism to create its GUI. What remains to be done is to interconnect the elements of the data type in a suitable way. This avenue has been explored in our previous work on generating GUIs for the desktop resulting in the GEC toolkit [AvP03, AvP04, AvPv04] as well as for the web, resulting in the iData toolkit [PA05, PA06]. The host language is the pure and lazy functional programming language CLEAN (for readers who are more familiar with the functional language HASKELL, we refer to [Ach07] for a concise overview of the differences between CLEAN and HASKELL). We use a functional language because they are known to support formal reasoning well; we use CLEAN because it comes with the interactive proof assistant SPARKLE [dvP02, dvP08a], which allows us to reason about CLEAN programs. Furthermore, because the above mentioned toolkits have been implemented in CLEAN, we wish to reason about the programs, and not a derived model of a program. Finally, the built in generic programming support in CLEAN is used for the automatic derivation of GUIs.

With the GEC and iData toolkit, the programmer creates dialogs, or forms, by means of designing a structured data type that identifies the values that
can be edited by the user. Whenever such a value has been edited, it may invoke an effect on other dialogs, or forms. Put in other words, these dialogs are interconnected. In the toolkits, we have explored two paths to define this interconnection relation.

- In the first approach, the ‘freestyle’ approach, editors are interconnected by means of a function that invokes editors when needed. This provides the programmer with the full expressive power of the host language.

- In the second approach, we capture the interconnection relation with the Arrow framework [Hug00]. In this way, the programmer exchanges freedom of expressiveness with the rigor of a small set of combinator functions.

It is our goal to reason formally about interactive GUI programs written in either the GEC or the iData toolkit. Eventually, we want to be able to do this for programs written in either of the above styles, but for now we restrict ourselves to the combinator based approach. The point-free style of Arrow combinators makes them particularly amenable to formal reasoning. We will use the proof assistant SPARKLE, not only because it will aid us in managing with the proofs, but also because every complete SPARKLE proof takes definedness properties into account, i.e. reasoning about how a program deals with undefined values (⊥) and under which conditions ⊥ values are yielded [vd06, vd07]. With the aid of SPARKLE, we have been able to formalize definedness relations of the Arrow combinators of our framework.

Our framework is an event handling system, where the events model user edit operations on editors. This is different from the standard approach to Arrow based systems, where the value of a system is determined by evaluating the Arrow system from the start until the end. Event based systems necessarily need to ‘break into’ the circuit that is created by the arrow expression, because an event causes an effect only after the targeted editor. Another unusual feature of interactive applications is sharing editor states. Editors are identified objects. Two (or more) editor objects with the same identifier conceptually refer to the same object, and hence, the same state. In the realization, any two shared editors are mapped to a single appearance in the concrete user interface that is presented to the user. In this way, complex interconnection patterns can be constructed. Despite these differences, we show that our EditorArrow model satisfies the standard set of laws that are imposed on Arrow models. In addition, we identify a number of specific laws for editors and we identify definedness properties for our editor arrows.

In summary, we propose a common formal semantic model for interactive GEC and iData programs written in an Arrow combinator style. We define both denotational and operational semantics for the EditorArrow model. Programming applications of EditorArrow combinators are expressed in CLEAN, which allowed us to use the interactive proof assistant SPARKLE. In some cases, this pointed to situations that were clearly undesired, but that had escaped our scrutiny. This has led to changes both on the semantical level and in the specifi-
cation of the properties. We handle some programming examples and reasoning case studies, some of which concern reasoning about definedness.

The remainder of this paper is structured as follows. We first present the two toolkits in Sect. 10.2 discussing the differences and correspondences. A common EditorArrow model abstracting from these toolkits is defined denominationally and operationally in Sect. 10.3. Sect. 10.4 gives the standard Arrow laws, and identifies iteration and editor laws, as well as definedness laws for the basic combinators of this semantic model. In Sect. 10.5 we present a reference implementation of the EditorArrow model in Clean. We give some small example EditorArrow programs and we also discuss formal reasoning about these programs with Sparkle. Related work is presented in Sect. 10.6 and we end with conclusions in Sect. 10.7.

10.2 The GEC and iData Toolkits

In this section we briefly introduce the two toolkits, GEC and iData, discuss their similarities and differences, and identify a common api, for which a semantic model will be defined and used in the remainder of this paper.

The GEC and iData toolkits have been designed for different contexts (widget based versus web-based GUIs), but with the same goals and design principles: to automatically generate GUIs from structured types, and to consider such a GUI as an editor for values of that type. Hence, an editor is a typed unit that provides the application user with a GUI to edit values of that given type only. The concept of type parameterized editors provides a strong abstraction mechanism to eliminate the differences of the two back-ends of the toolkits. In this way, they become closely related. There are however also many differences between these toolkits with respect to behavior, implementation, and use. Before we distill a common api, we first discuss the differences.

The GEC Toolkit has been designed and implemented to create desktop GUI applications. It has been implemented in the GUI toolkit of Clean, Object I/O [AP98]. An editor is an interactive element that resides in a window. In addition, the state of the editors is resident. Just like any other interactive element of Object I/O, editors are managed by the program: they can be created, altered, and closed. The internal implementation of an editor basically copies the generic decomposition of the editor’s value to a (large set of) GUI-fragment/receiver pairs. This allows to refresh only the significant parts of the GUI when values are modified by the user. The editor responds to such a user action by means of a callback function, as is usually done in desktop GUI applications, and Object I/O as well. In this callback function, the programmer has access to the full Object I/O library and all other editors.

The iData Toolkit has been designed and implemented to be a web application. An iData application can be opened within any browser, and navigated with the usual back and forth buttons. Editors are interactive
elements that reside within a browser window as form elements. With each user action, a new web page needs to be rebuilt. This is an essential difference with the GEC toolkit that also has its impact at the programmer’s level: in the GEC toolkit editors need to be closed explicitly, whereas this is not required of the programmer in the iData toolkit. For this reason, iData programming is much easier than GEC programming. In contrast with the GEC toolkit, the value of an iData element is not decomposed generically, but rather kept intact, and is ‘patched’ by a generic function whenever the user alters part of the state of the corresponding iData element. Editors in the iData toolkit have no callback functions to alter each other. Instead, it is the program that manipulates the editors and makes their values depend on each other. When computing web pages, the toolkit encodes the editor states, which can reside within the HTML page, or stay on the server side. An iData application computes HTML forms that are generated from the type of the corresponding editor. Within the program, the programmer has access to the generated HTML. She can define additional HTML content, and can control the layout of editors.

In order to illustrate the rather large differences between GEC programs and iData programs, we first give an off-the-shelf, ‘freestyle’, implementation in the two toolkits of a case study.

### 10.2.1 Example: Variable Sum List in GEC and iData

The case study is an interactive program that allows the user to enter a positive number in one input field. This number determines the total number of other integer input elements. These input elements can also be edited by the user. After each such edit action, their sum should be displayed in a final integer editor. This can be repeated as many times as the user likes. She can increase and decrease the number of integer input elements, alter their values, and is informed of the sum of their values.

For both toolkits we present the ‘freestyle’ versions of this case study. The GEC program (Fig. 10.1) needs to import the GEC library StdGEC, as well as the CLEAN prelude StdEnv (line 2). The main wrapper function of the GEC toolkit is startGEC, which expects a GEC function (varsumlist) that creates the GUI that belongs to this program. Being based on Object I/O, GEC editors are interactive elements that are parameterized with callback functions that define the response of each editor to a change of value. However, the GEC toolkit deviates from the Object I/O api convention that their constructor functions (gecEdit and gecHide) yield a GEC handle to the created GEC editor rather than being provided with one. The first editor that is created is the display of the sum of all values (line 5). Because it does not have to direct its output to another editor, its callback function is simply noUpdate, which does not change the environment. The second editor that is created stores the current list of argument editors, which is initially empty (line 6). (A store, created with hideNGEC, is just an invisible editor.) These editors are needed to close them.
module varsumlist_GEC_FreeStyle

import StdEnv, StdGEC

Start world = startGEC Void varsumlist world

varsumlist pSt

♯ (sumGEC.pSt) = gecEdit 0 OutputOnly noUpdate pSt
♯ (argGEC.pSt) = gecHide [] pSt
♯ (nrGEC, pSt) = gecEdit 0 Interactive (createNr sumGEC argGEC) pSt
= pSt

where

createNr sumGEC argGEC _ n pSt
♯ (argGECs.pSt) = argGEC.gecGetValue pSt
| n < length argGECs
♯ (keep, away) = splitAt n argGECs
♯ (vs.pSt) = seqList (map (λ gec → gec.gecGetValue) away) pSt
♯ pSt = foldr closeGECGUI pSt away
♯ pSt = set gec NoUpdate keep pSt
= sumField sumGEC Enquire (~(∑ sum vs)) pSt
| otherwise
♯ (new.pSt) = seqList [ gecEdit 0 Interactive (sumField sumGEC)
\ i ← [1..n-currNrArgs] ] pSt
= set gec NoUpdate (argGECs ++ new) pSt

sumField sumGEC _ v pSt
♯ (curSum.pSt) = sumGEC.gecGetValue pSt
= set sumGEC YesUpdate (curSum + v) pSt
gecEdit v d f = createNGEC "Example" d True v f
gecHide v = hideNGEC title OutputOnly True v noUpdate
noUpdate _ _ env = env
closeGECGUI gec = gec.gecClose o (gec.gecCloseGUI SkipCONS)
set gec = gec.gecSetValue

Figure 10.1: The GEC varsumlist program in ‘free-style’.

afterwards. In line 7, the integer editor is created in which the user can enter
the desired number of input fields. Its callback function is createNrFields which
is parameterized with the GEC references sumGEC (the sum display) and argGEC
(the stored list of current editors). When its value is altered, it checks whether
the new number is less than the current number of editors. In that case, it closes
the appropriate editors (lines 13-18). In the other case, new editors should be
created (lines 19-21). Each sum argument editor has the same callback function,
(sumField sumGEC) (lines 22-24) which first determines the current value of the
sum display, and updates it with the new value.

The iData program (Fig. 10.2) needs to import the iData toolkit, besides
the CLEAN prelude (line 2). Its main wrapper function is doHtmlWrapper, which
expects a function (varsumlist) that computes a HTML page, that may contain
Section 10.2.2: Abstractions towards Editor Arrows

module varsumlist_iData_FreeStyle

import StdEnv, StdiData

Start world = doHtmlWrapper varsumlist world

varsumlist hSt

% (nrF, hSt) = mkEditForm (Init.nFormId "nr" 0) hSt
% (argFs,hSt) = seqList [ mkEditForm (Init.nFormId ("arg " <+ i)) 0 ]
% i ← [0..value nrF-1]
% (sumF, hSt) = mkEditForm (Set.ndFormId "sum" (sum (map value argFs)))

\ = mkHtml "Example"

[mkColForm (map (BodyTag o form) ([nrF <+ argFs <+ [sumF]])) [value form = form.value]

form form = form.form

Figure 10.2: The \textit{iData} varsumlist program in ‘free style’.

forms, created as editors for \textit{iData}. In \textit{iData}, an editor is created with the
function \texttt{mkEditForm}. \textit{iData} follows the \textit{Object I/O} convention to parameterize
constructor functions with their handles, rather than yielding such a value.
When an editor’s value depends on the value of other editors, then its value
must be \texttt{Set}. An example is in line 9, where the sum display is defined: its
value must be the sum of the values of the argument editors. The other editors
have just initial values (line 5 and 6). The HTML page (lines 11-12) that is
computed is a single column of all form renderings of all editors, starting with
the number editor, followed by the list of value editors, and closed with the sum
display.

The two programs behave similarly, yet their specifications are very different
and the implementation of the underlying toolkits are even more different. The
\textit{iData} version is much shorter and more declarative than the \textit{GEC} version,
because it only specifies which editors depend on which other editors: if the user
enters a lower number of editors, then the \textit{iData} toolkit only includes the re-
main ing editors in the HTML page. This is very different from the \textit{GEC} version,
in which the program must close the editors itself.

Having discussed the main differences, we can now turn our attention to the
similarities and then extract a common core.

10.2.2 Abstractions towards Editor Arrows

Despite the above mentioned differences, these programs have a lot in common:
both use editors to interact with the user, and both programs specify the same
interconnection relation: an integer value is displayed that is the sum of the
values of \textit{n} integer editors, where \textit{n} is the value edited by the user in some
first editor. Clearly, the \textit{iData} example in Fig. 10.2 has the closest match with this interconnection relation. We can rearrange the \textit{GEC} toolkit in such a way that its behavior is similar to that of an \textit{iData} program: immediately after each user event, all editor GUIs are closed, and are reopened only if they are created during execution. In fact, this behavior is already implemented at the level of each individual \textit{GEC}: the user can switch between the constructors of a value of an algebraic data type without having to reconstruct the intermediate values. The other change that needs to be made is that editors are identified by means of a label instead of a \textit{GEC} value. Again, the adapted toolkit can maintain an administration in which labels are associated with \textit{GEC} values. With these arrangements the freestyle \textit{GEC} version can be expressed much shorter and results in the code displayed in Fig. 10.3.

\begin{verbatim}
module varsumlist_GEC_FreeStyle2
import StdEnv, StdGEC

Start world = startGEC Void varsumlist world

varsumlist pSt
  \# (sumGEC, pSt) = gecEdit 0 OutputOnly noUpdate pSt
  \# (nrGEC, pSt) = gecEdit 0 Interactive (createNrFields sumGEC) pSt
  = pSt
where
  createNrFields sumGEC _ n

Even though the adapted \textit{GEC} version still uses callback functions to specify the interconnection of editors, its resemblance with the \textit{iData} version has increased significantly. We continue to eliminate the differences between the two toolkits by means of abstraction. These abstractions are:

- \textit{We ignore all layout issues.}\n  In the \textit{GEC} toolkit, editors can reside in different windows. In the \textit{iData} toolkit, all editors reside in the same browser window. We also ignore where the editors within a window appear, what they look like, and what their dimensions are.

- \textit{We abstract from residence of state.}\n  We simply assume that every editor has access to its state value.
Section 10.2.2: Abstractions towards Editor Arrows

- We abstract from representation (widgets versus forms).
  We are only concerned with editors that respond to value changes. We know that we can derive a rendering for each and every type and do not wish to reason about these renderings.

- We abstract from the communication method (events versus post/get).
  Instead, we consider user actions to be just editing actions which can be modeled conveniently as a whole new value of the same type of value that is maintained by the editor.

- We abstract from specific typing and type classes issues.
  There are many different (generic) classes in the two toolkits, but essentially they all make sure that an editor can be rendered, its value (de)serialized and changed.

As a result of these abstractions we can consider editors of values of any type as basic building blocks. The next step to undertake is to unify editor declarations and the means to interconnect them. The examples in Fig. 10.1 and Fig. 10.2 illustrate that it is very unlikely that we will succeed in doing this for ‘freestyle’ programs (even for the modified GEC toolkit example in Fig. 10.3). As stated in Sect. 10.1, we use a combinator approach based on the Arrow framework for this. Hence, the editor declaration will become a basic Arrow combinator. We adopt the conventions of the iData toolkit:

- Editors are identified by means of a unique label and initial value.
  In the GEC toolkit, the programmer needs to use the handle to a GEC editor for this purpose, which is only available after creating the editor. This leads in many cases to reversed editor creation, as is also illustrated in Fig. 10.1 in which the sum display editor needs to be created first because it is manipulated by the other editors. The iData approach is actually similar to the one taken in Object I/O, in which identifiers are created independently of the elements that they identify.

- Editors are shared by means of declaring an editor with the same identifier.
  In the GEC toolkit, sharing is realized by manipulating the handle to the GEC editor. Again, the use of two different means to identify the same editor is uncomfortable, and we prefer the uniform approach of the iData toolkit.

- Editor values are read and set by subsequent declarations of editors with the same identifier.
  In the GEC toolkit, the value of an editor can be read and set via its handle, and when the editor is created. Because we do not want to have two different forms of access, we combine reading and setting the value of an editor with its declaration.

Both the GEC and iData toolkit have one primitive generic editor creation function (gGEC used by createNGEC and hideNGEC in the GEC toolkit, and mkViewForm
used by \texttt{mkEditForm} in the \textit{iData} toolkit). We can choose to use a single editor creation combinator function as well, but instead we prefer to emphasize the \textit{two} different ways of using an editor, each expressed with a separate combinator function, viz. \texttt{editread} and \texttt{editset}. They have slightly different signatures: both receive an identifier value (unique label and initial value) via the arrow state, but \texttt{editset} is also provided with the new value of the editor. Both editors behave the same when manipulated by the user: they receive a new value and emit that value via the arrow state. The difference shows up when an editor that appears earlier within the arrow relation has been manipulated by the user: the \texttt{editread} editor simply \textit{echoes} its current value via the arrow state, whereas the \texttt{editset} editor \textit{copies} the value that is received via the arrow state as its new value, and emits that new value via the arrow state. Note that the editor that appears earlier within the arrow relation can be \textit{the same} editor, by using the same identifier. In this way, intricate relationships can be defined via sequential composition rather than a cyclic combination of editors.

### 10.2.3 Example: Variable Sum List in \textit{Arrow} style

In Sect. 10.2.1 we have presented the ‘freestyle’ versions of the variable sum list case study in both the \textit{GEC} and \textit{iData} toolkit. We now show what the respective solutions look like in the two toolkits with the \textit{Arrow api}. The solutions are shown in Fig. 10.4 and Fig. 10.5. The most important aspect is that the \texttt{varsumlist \textit{Arrow}} expression is \textit{identical} in both programs. They still need different wrapper functions (\texttt{startGEC} and \texttt{doHtmlWrapper} respectively) and need to import different toolkits.

```haskell
module varsumlist_GEC_ArrowStyle where

import StdEnv, StdGEC

Start world = startGEC Void (startCircuit varsumlist 0) world

varsumlist = arr (λx → nrId)
  >>> editread
  >>> arr (λn → (n,0))
  >>> iterateN (first (arr argId >>> editread))
    >>> arr (uncurry (+))
  >>> arr (λt → (sumId,t))
  >>> editset

nrId = ("nr", 0)
sumId = ("sum",0)
argId n = ("arg +n",0)
```

Figure 10.4: The \textit{GEC varsumlist} program in \textit{Arrow} style.

The \texttt{varsumlist} expression uses all standard \textit{Arrow} combinators: \texttt{arr f} which
Section 10.3: Editor Arrows

Both in the GEC and in the iData toolkit an editor can be regarded as a uniquely named, typed storage for a single value. It presents a GUI to the application user...
to alter this value. When connected to another editor, the editor communicates its stored value both when its value is changed by the user and when a change of another editor is received.

An interactive application is a collection of such connected editors. We will define and use EditorArrow combinators to define the connections between editors in a point-free style.

### 10.3.1 Denotational Semantics for Editor Arrows

In the classic approaches to functional reactive programming [EH97, CNP03, HCNP03] the basic building block is formed by signals, defined as time-varying values:

\[
\text{Signal } a = \text{Time} \rightarrow a
\]

Signals are therefore well suited to define values that vary smoothly over time. They can also be used to accommodate the discrete nature of events as they occur in GUIs [CE01]: at time \( t \) either an event \( e \) is available (Just \( e \)) or it is not (Nothing). Hence, by defining

\[
\text{Event } a = \text{Maybe } a
\]

event streams can be included as Signal (Event \( a \)) functions.
Section 10.3.1: Denotational Semantics for Editor Arrows

From the account in Sect. 10.2, it follows that in the case of editors we are only concerned with events and event streams. Therefore, in our framework a Signal (Event a) simplifies to a list based event stream. In the EditorArrow framework, an interactive program processes a stream of events, EditEvents, which is modeled conveniently as a list of events.

\[ \text{EditEvents} = [\text{EditEvent}] \]

Interactive programs consist of arbitrarily many editors, each having a value of possibly different type. If we would model this with a strongly typed programming language (as we will in Sect. 10.5) this would lead to the use of existential or dependent types or some other mechanism. Here, we just assume a Value domain, and use lists of values abstracting from the way this is specified in a programming language.

When the user manipulates an editor that is identified via \( \text{eid} : \text{ID} \), he eventually generates a new value \( v : \text{Value} \). This event is modeled as a pair of the \( \text{eid} : \text{ID} \) value of the editor, and the new value \( v : \text{Value} \) that the user has generated. The ID consists of the name of the editor and its initial value which it will have as long as no event for it has occurred.

\[ \text{EditEvent} = \text{ID} \times \text{Value} \]
\[ \text{ID} = \text{Name} \times \text{Value} \]

As stated above, an interactive program consists of arbitrarily many editors that have a data value that can be manipulated by the user. We collect these editable data in a set of pairs:

\[ \text{EditableData} = \wp(\text{ID} \times \text{Value}) \]

We want all values in the EditableData domain to be fully defined since these are the values that are to be displayed. We can “read” and “write” pair values from this set using two primitives, \( \text{read} \) and \( \text{write} \). We assume an access function \( \text{initvalue} \) to take from an event identifier of type ID the value part which holds its initial value. Note that these primitives require their arguments to be fully defined since the resulting EditableData domain is fully defined.

\[ \text{read } \text{eid } s = \begin{cases} d & \text{if } (\text{eid},d) \in s \\ \text{initvalue } \text{eid} & \text{if } (\text{eid},d) \notin s \end{cases} \]
\[ \text{write } \text{eid } v s = \begin{cases} (\text{eid},v) \cup s \setminus (\text{eid},d) & \text{if } (\text{eid},d) \in s \\ (\text{eid},v) \cup s & \text{if } (\text{eid},d) \notin s \end{cases} \]

The ID values serve as unique keys in \( s : \text{EditableData} \):

\[ \forall \text{eid} : \text{ID}, s : \text{EditableData}. (\text{eid},d) \in s \land (\text{eid},d') \in s \Rightarrow d = d' \]

In Sect. 10.2 we stated that we want to construct programs by means of the Arrow combinators. An Arrow program fragment processes an event. This is modeled by \( \text{EventStatus} = \{ \text{Pending}, \text{Processed} \} \). We define two predicates
pending and processed that hold only if their EventStatus argument has the corresponding value. Processing an event possibly updates the existing editable data. In addition, it expects an incoming value of type a, and emits an outgoing value of type b. The editable data together with an incoming or outgoing value and the status of event processing are put in one triplet: the EState. A program fragment is an Editable Data and Event Transformer function, abbreviated as EDET:

\[ EState \ a \ = \ EditableData \times a \times EventStatus \]
\[ EDET \ a \ b \ = \ Event \to EState \ a \to EState \ b \]

In contrast to classic reactive programming with Signals, where state is always local (introduced by the use of loop), we are modeling a situation where essentially global data are edited. Hence, we take as the basis of our Arrow modelling the type EDET a b.

The arrow expressions that we allow are built in the following way:

\[ EdArrow \ ::= \ arr \ Fun \mid EdArrow \ggg EdArrow \mid first \ EdArrow \]
\[ \mid left \ EdArrow \mid iterate \ EdArrow \]
\[ \mid editread \mid editset \]

where Fun represents functions as expressed in a functional language.

Denotationally, we define a partial function \( [-] \) from these arrow expressions to the functions on the EDET domain. Why this is a partial function will be explained later in section 10.4.4.

\[ [-] : EdArrow \to EDET \ a \ b \]

Below we specify the meaning for each of the combinators denotationally. We use tuples and lists for lambda arguments and standard case, if and non-recursive let constructs to keep the definitions concise and readable.

The basic classic combinators (arr, \( \ggg \) and first) are easily defined. For the meaning of \( f \) in the arr rule we rely on standard lazy functional language semantics \( [-]_\lambda \) [CD82], using domains that are lifted by adding \( \bot \) to them as domain value. It is important to note that the specific domains for this model (EditEvent, EditableData and their components) are not lifted.

\[
\begin{align*}
[\ arr \ f \] & = \ \lambda e.\lambda(s,a,p).(s,[f]_{\lambda \bot} a,p) \\
[f \ggg g] & = \ \lambda e.([g] e) \circ ([f] e) \\
[first \ f] & = \ \lambda e.\lambda(s,bd,p).let \ (b,d) = bd \\
& \ \ \ \ \ \ \ \ \ \ \ \ \ \ (s',c,p') = [f] e (s,b,p) \in (s',(c,d),p')
\end{align*}
\]

The definition of first has an interesting aspect. If the pattern \((b,d)\) is undefined then the result of the meaning function may still be a triplet with a defined or undefined second triplet element, all depending on the meaning of \( f \).

For our purposes, we also need some choice combinator. The standard way to do this is to use a left combinator. Based on left, different kinds of choice
combinators can be created using the lifted standard Either type. Since this domain is lifted, the result of the case definition can be a partially defined function.

\[
[\text{left } f] = \lambda e.\lambda (s, \text{either}lr, p).
\]

\text{case eitherlr of}

\[
\begin{align*}
\text{Left } a &= \text{ let } (s', b, p') = f e (s, a, p) \text{ in } (s', \text{Left } b, p') \\
\text{Right } c &= (s, \text{Right } c, p)
\end{align*}
\]

The meaning of the two combinators for the basic editor variants, \textit{editread} and \textit{editset}, are defined straightforwardly using the \textit{read}, \textit{write} and \textit{pending} functions. We follow the intuitive meaning described in the previous section on page 172 quite closely. Since the \textit{eid} and \textit{eida} event identifiers are lifted and they are passed to the \textit{read} and \textit{write} primitives which require non lifted values, this is a partial definition.

\[
[\text{editread}] = \lambda (eid', v).\lambda (s, eid, p).
\]

\[
\begin{cases}
\text{write eid } v s, v, \text{Processed} & \text{if } \text{pending}(p) \land eid = eid' \\
(s, \text{read } eid s, p) & \text{if } \neg \text{pending}(p) \lor eid \neq eid'
\end{cases}
\]

\[
[\text{editset}] = \lambda (eid', v).\lambda (s, eida, p).
\]

\[
\begin{cases}
\text{write eid } v s, v, \text{Processed} & \text{if } \text{pending}(p) \land eid = eid' \\
\text{write eid } a s, a, p & \text{if } \neg \text{pending}(p) \lor eid \neq eid'
\end{cases}
\]

We will also need some kind of recursion. Both the \textit{GEC} and the \textit{iData} toolkit have the property that they have a single arrow expression called by a wrapper (which is essentially an event loop that deals with consecutive events recursively). These editor arrow expressions build a finite, fully evaluated interface for the user. This interface may be dynamic in the sense that the user can influence its values and its size but it will always be finite and fully evaluated. For modeling recursion on the level of such editor arrow expressions we need nothing more than primitive recursion. As the basic building block for primitive recursion we use the \textit{iterate} combinator that iterates its argument arrow a finite number of times using a lifted natural number \(n\). Analogous to the choice combinator \textit{left} the result may be partially defined since the \((n, a)\) value is in a lifted domain.

\[
[\text{iterate } f] = \lambda e.\lambda (s, (n, a), p).
\]

\[
\begin{cases}
(s, a, p) & \text{if } n = 0 \\
\text{let } (s', a', p') = [f] e (s, (n, a), p) & \text{if } n > 0 \\
\text{in } [\text{iterate } f] e (s', (n - 1, a'), p')
\end{cases}
\]

The above denotational semantics states what the meaning is of an arrow expression on a single event. To define what happens with an event stream, consisting of a list of \textit{EditEvents} we need to model the toolkit wrappers’ event loops.
Chapter 10: A Common Arrow Based Semantics for GEC and iData

\[ f \]_{\text{eventstream}} = [\text{eventloop } f]

The toolkit wrappers are modeled by a loop combinator as is introduced by Paterson. The loop combinator is defined using the standard least fixed point combinator \( \text{Y} \). In our case however, this loop combinator will occur exactly once (note that it is not part of the definition of \( \text{EdArrow} \)), on the outside of an editor arrow expression. To avoid confusion we have not called this a loop combinator but an \text{eventloop} combinator.

\[
[\text{eventloop } f] = \text{Y}
\left( \lambda \text{evloopf}. \lambda (s,a). \lambda \text{es}.
\begin{cases}
  s & \text{if } \text{es} = [] \\
  \text{let } (s', b, p) = [f] (\text{hd } \text{es}) (s, a, \text{Pending}) & \text{if } \text{es} \neq [] \\
  \text{in } \text{evloopf} (s', a) (\text{tl } \text{es})
\end{cases}
\right)
\]

Using \text{iterate} within arrow expressions and one single \text{eventloop} on the outside we have exactly the right expressive power for the \text{EditorArrow} model.

10.3.2 Operational Semantics for Editor Arrows

For implementing the \text{EditorArrow} model we also need operational semantics. They are derived straightforwardly from the denotational semantics. We take again the same domains. The operational semantics are defined in the standard way using ‘big-step’ semantics. The relation \( \rightarrow \) is suffixed with the handled event \( e : (id, v) \) which is assumed to be always defined. It relates the argument triplet \((s, a, p)\) of store, value and boolean to a result triplet. The rules define what the semantics is for defined triplets. For other cases the semantics is undefined.

The rules for the basic combinators are given below. With \( \rightarrow_{\lambda\perp} \) we denote the standard reduction from functional languages.

\[
\frac{f \ a \rightarrow_{\lambda\perp} a'}{\text{arr} \ f \ (s, a, p) \rightarrow_{c:(id,v)} (s, a', p)} \quad (arr)
\]

\[
\frac{f \ (s, a, p) \rightarrow_{c:(id,v)} (s', a', p') \quad g \ (s', a', p') \rightarrow_{c:(id,v)} (s'', a'', p'')}{f \gggg g \ (s, a, p) \rightarrow_{c:(id,v)} (s'', a'', p'')} \quad (seq)
\]

The first rule requires two alternatives since the value domain is lifted and we want a lazy variant of \text{first} consistent with the denotational definition.

\[
\frac{f \ (s, a, p) \rightarrow_{c:(id,v)} (s', a', p')}{\text{first} \ f \ (s, (a, c), p) \rightarrow_{c:(id,v)} (s', (a', c), p')} \quad (\text{first})
\]

\[
\frac{f \ (s, \perp, p) \rightarrow_{c:(id,v)} (s', a', p')}{\text{first} \ f \ (s, \perp, p) \rightarrow_{c:(id,v)} (s', (a', \perp), p')} \quad (\text{first}_\perp)
\]
Section 10.3.2: Operational Semantics for Editor Arrows

Operationally, we need for the left combinator the following choice rules (we do not have an extra undefined rule here, we use a partial definition instead):

\[
f (s, a, p) \rightarrow e_{c(id, v)} (s', a', p')
\]

(choice_left)

\[
\text{left } f (s, \text{Left } a, p) \rightarrow e_{c(id, v)} (s', \text{Left } a', p')
\]

(choice_left)

\[
\text{left } f (s, \text{Right } a, p) \rightarrow e_{c(id, v)} (s, \text{Right } a, p)
\]

(choice_right)

Both the editor combinators distinguish between the case where the event is pending (in which case it has to be processed when it matches the id of the editor) or not. The operational semantics employs the same primitives (pending, read and write) as the denotational semantics.

\[
s' = \text{write } id \; v \; s \quad \text{pending}(p)
\]

(editread_pending)

\[
\text{editread } (s, id, p) \rightarrow e_{c(id, v)} (s', v, \text{Processed})
\]

\[
a = \text{read } id \; s \quad \text{id} \neq \text{id}' \lor \neg \text{pending}(p)
\]

(editread_other)

\[
\text{editset } (s, (id, a), p) \rightarrow e_{c(id, v)} (s', a, \text{Processed})
\]

(editset_pending)

\[
s' = \text{write } id \; a \; s \quad \text{id} \neq \text{id}' \lor \neg \text{pending}(p)
\]

(editset_other)

Iteration is defined through two rules (using a natural number). We have one rule for the base case and another for the iterating case using the natural number to count the number of iterations.

\[
\text{iterate } f (s, (0, a), p) \rightarrow e(s, a, p)
\]

(iter_0)

\[
f (s, (n+1, a), p) \rightarrow e(s', a', p') \quad \text{iterate } f (s', (n, a'), p') \rightarrow e(s'', a'' , p'')
\]

(iter_n)

Finally, the event loop is defined straightforwardly dealing with events one by one and passing the resulting store to the next event. We only yield the store as result since, at each new event the store is augmented to a triplet with the same initial value and the same boolean indicating that the event has not been processed yet.

\[
f (s, a, \text{Pending}) \rightarrow \eta s
\]

(events_end)
\[
\begin{align*}
\text{events}_\text{next} & \quad \leftarrow \quad f(s, a, \text{Pending}) \rightarrow_f (s', a', p') \\
& \quad \text{events}_\text{next} & \quad \leftarrow \quad \left(f(s, a, \text{Pending}) \rightarrow_{\epsilon \cdot \alpha} s''\right)
\end{align*}
\]

It is easy to prove that the operational semantics is sound with respect to the denotational semantics. The operational semantics will be used as the basis for a reference implementation of the framework in the programming language Clean in Sect. 10.5.

### 10.4 Properties of Editor Arrows

In this section we state the basic properties of the semantic model that has been presented in the previous section. The “classic” Arrow laws, as described by Hughes and Paterson, are valid for this model. These laws are given as Def. 10.4.1.

In Sect. 10.4.1 we introduce “iterate” laws and in Sect. 10.4.2 we give properties of the “eventloop”. We introduce basic “editor” laws in Sect. 10.4.3. Finally, we provide “definedness” laws in Sect. 10.4.4.

**Definition 10.4.1:** *(Classic Arrow Laws)*

\[
\begin{align*}
\text{arr id} > > > f & \quad \text{(Left unit)} \quad \equiv \quad f \quad \text{arr id} \\
\quad & \quad \text{(Right unit)} \\
\quad f \quad > > > (g \quad > > > h) & \quad \text{(associativity of } > > > \text{)} \quad \equiv \quad (f \quad > > > g) \quad > > > h \\
\quad \quad & \quad \text{(assoc preserves } > > > \text{)} \\
\quad \quad \text{arr} (g \circ f) & \quad \equiv \quad \text{arr} f \quad > > > \text{arr} g \\
\quad \quad \text{arr} (g \circ f) & \quad \equiv \quad \text{arr} f \quad > > > \text{arr} g \\
\quad \quad \text{first} (\text{arr} f) & \quad \equiv \quad \text{arr} (f \times \text{id}) \\
\quad \quad \text{first} (\text{arr} f) & \quad \equiv \quad \text{first} f \quad > > > \text{first} g \\
\quad \quad \text{first} f \quad > > > \text{arr} (\text{id} \times g) & \quad \text{(assoc eliminates } \text{first}) \quad \equiv \quad \text{arr} \quad \text{fst} \quad > > > f \\
\quad \quad \text{first} f \quad > > > \text{arr} \quad \text{fst} & \quad \text{(assoc eliminates } \text{first}) \quad \equiv \quad \text{arr} \quad \text{fst} \quad > > > f \\
\quad \quad \text{first} f \quad > > > \text{arr} \quad \text{assoc} & \quad \text{(assoc eliminates } \text{first}) \quad \equiv \quad \text{arr} \quad \text{assoc} \quad > > > \text{first} f \\
\quad \quad \text{left} (\text{arr} f) & \quad \equiv \quad \text{arr} (f \oplus \text{id}) \\
\quad \quad \text{left} (\text{arr} f) & \quad \equiv \quad \text{left} f \quad > > > \text{left} g \\
\quad \quad \text{left} f \quad > > > \text{arr} (\text{id} \oplus g) & \quad \text{(left exchanges)} \quad \equiv \quad \text{arr} (\text{id} \oplus g) \quad > > > \text{left} f \\
\quad \quad \text{arr} \quad \text{Left} \quad > > > \text{left} f & \quad \text{(left unit)} \quad \equiv \quad \text{f} \quad > > > \text{arr} \quad \text{Left} \\
\quad \quad \text{left} (\text{left} f) \quad > > > \text{arr} \quad \text{assocsum} & \quad \text{(left association)} \quad \equiv \quad \text{arr} \quad \text{assocsum} \quad > > > \text{left} f
\end{align*}
\]

**where**

\[
\begin{align*}
\quad \text{fst} (a, b) & \quad \equiv \quad a \\
\quad f \times g (a, b) & \quad \equiv \quad (f \text{ a, } g \text{ b}) \\
\quad f \oplus g (\text{Left a}) & \quad \equiv \quad \text{Left} (f \text{ a}) \\
\quad f \oplus g (\text{Right b}) & \quad \equiv \quad \text{Right} (g \text{ b}) \\
\quad \text{assoc} ((a, b), c) & \quad \equiv \quad (a, (b, c)) \\
\quad \text{assocsum} (\text{Left} (\text{Left} a)) & \quad \equiv \quad \text{Left} a \\
\quad \text{assocsum} (\text{Left} (\text{Right} b)) & \quad \equiv \quad \text{Right} (\text{Left} b) \\
\quad \text{assocsum} (\text{Right} c) & \quad \equiv \quad \text{Right} (\text{Right} c)
\end{align*}
\]
10.4.1 Iterate Laws

Def. 10.4.1.2 states the two iterate laws. There is a rule for 0 and a rule for \( m+1 \). They are described nicely using an auxiliary function \( \odot \). This auxiliary function puts an argument number \( a \) in a pair with the arrow result value, that is being passed, such that \( \text{iterate} \) can use this number to count the iterations.

The \( \text{iterate base} \) law expresses the fact the argument is applied zero times. The \( \text{iterate next} \) law expresses the fact that the argument is applied \( m+1 \) times consecutively with decreasing values starting with \( m+1 \).

**Definition 10.4.1.2: (Iterate Laws)**

\[
\begin{align*}
\text{iterate} f \odot 0 \ (\text{iterate base}) &= \text{arr id} \\
\text{iterate} f \odot (m+1) \ (\text{iterate next}) &= f \odot (m+1) \triangleright \text{iterate} f \odot m
\end{align*}
\]

where \( f \odot a = \text{arr} (\lambda x \to (a, x)) \triangleright f \)

10.4.2 Eventloop Properties

The properties of the eventloop are given in Def. 10.4.2.3. There are two properties. The property \( \text{eventloop end} \) expresses that, when there is no event anymore, the result is the store. The \( \text{eventloop next} \) property expresses that the events are dealt with one after the other passing the state and using the same initial value and event status over and over again. This last property requires some auxiliary “plumbing” functions.

**Definition 10.4.2.3: (Eventloop Properties)**

\[
\begin{align*}
\text{eventloop} f (s, a) [] \ (\text{eventloop end}) &= s \\
\text{eventloop} f (s, a) [e: es] \ (\text{eventloop next}) &= \text{eventloop} f
\quad (\text{drop} ((\text{dfp} f) e (s, a, \text{Pending}))) \\
&\quad \triangleright es
\end{align*}
\]

where \( \text{dfp} f = \text{dupl} \triangleright \text{first} f \triangleright \text{pop} \)

\[
\begin{align*}
\text{dupl} &= \text{arr} (\lambda a \to (a, a)) \\
\text{pop} &= \text{arr} \text{snd} \\
\text{drop} &= \lambda (s, a, p) \to (s, a)
\end{align*}
\]

10.4.3 Editor Laws

The proofs of the classic arrow laws, the iterate laws and the eventloop properties do not rely essentially on the definitions of edit combinators, hence they are also valid for the \( \text{editread} \) and \( \text{editset} \) combinators. This means that we get already a lot of equivalences ‘for free’ when the edit combinators are involved.

In addition, we introduce ten laws that are specific to uses of \( \text{editread} \) and \( \text{editset} \). They are given as Def. 10.4.3.4.

- We distinguish four edit elimination laws (one for each combination of the two different edit combinators) expressing that editors behave as pure stores: it is harmless (and pointless) to store the very same data in the same location in sequence in two occurrences of the same editor (i.e. with
the same id). These laws are expressed again with the auxiliary function ⊙ to put the id at the right place for editset and another auxiliary function ⊚ to put the id at the right place for editread.

- The four edit swap laws express the property of independence of the order of two editors of values in the first and the second part of a tuple. In each of these laws it is assumed that i and j are different. The edit swap laws are expressed nicely in a symmetric way using the standard combinator *** and its “mirrored” variant ⋆⋆⋆.

Finally, we have two laws for often used standard application patterns of the edit arrow combinators: self and feedback:

- The self pattern is used to apply a function on the value that is edited by a user and store its result for this editor. In this way, editors can control the values that they contain. The self composition law states that function composition distributes over this self pattern.

- The feedback pattern is used for two editors to feed their results directly back to each other. In general, you cannot swap the order of different subsequent editors because they will respond differently to the same event sequence. The feedback swap law states that in the case of mutual feedback the order of the editors is irrelevant. In the case that i equals j this is a trivial consequence of applying the edit elimination laws.

### Definition 10.4.3: (Editor Laws)

<table>
<thead>
<tr>
<th>Law</th>
<th>Equation</th>
</tr>
</thead>
<tbody>
<tr>
<td>editread ⊚ i &gt; &gt; &gt; editread ⊚ i</td>
<td>(read read elimination) = editread ⊚ i</td>
</tr>
<tr>
<td>editset ⊚ i &gt; &gt; &gt; editset ⊚ i</td>
<td>(read set elimination) = editset ⊚ i</td>
</tr>
<tr>
<td>editset ⊚ i &gt; &gt; &gt; editread ⊚ i</td>
<td>(set read elimination) = editset ⊚ i</td>
</tr>
<tr>
<td>editset ⊚ i &gt; &gt; &gt; editset ⊚ i</td>
<td>(set set elimination) = editset ⊚ i</td>
</tr>
<tr>
<td>editread ⊚ i *** editread ⊚ j</td>
<td>(read read swap) = editread ⊚ j *** editread ⊚ i</td>
</tr>
<tr>
<td>editset ⊚ i *** editset ⊚ j</td>
<td>(read set swap) = editset ⊚ j *** editset ⊚ i</td>
</tr>
<tr>
<td>editset ⊚ i *** editset ⊚ j</td>
<td>(set read swap) = editread ⊚ j *** editset ⊚ i</td>
</tr>
<tr>
<td>editset ⊚ i *** editset ⊚ j</td>
<td>(set set swap) = editset ⊚ j *** editset ⊚ i</td>
</tr>
<tr>
<td>self f i &gt; &gt; &gt; self g i</td>
<td>(self composition) = self (g ° f) i</td>
</tr>
<tr>
<td>feedback i j</td>
<td>(feedback swap) = feedback j i</td>
</tr>
</tbody>
</table>

where

<table>
<thead>
<tr>
<th>Expression</th>
<th>Equation</th>
</tr>
</thead>
<tbody>
<tr>
<td>f ⊚ a</td>
<td>arr (λx → a) &gt; f</td>
</tr>
<tr>
<td>f ⊚ a</td>
<td>arr (λx → (a, x)) &gt; f</td>
</tr>
<tr>
<td>f *** g</td>
<td>first f &gt; second g</td>
</tr>
<tr>
<td>f *** g</td>
<td>second f &gt; first g</td>
</tr>
<tr>
<td>second f</td>
<td>arr swap &gt; first f &gt; arr swap</td>
</tr>
<tr>
<td>swap (x, y)</td>
<td>(y, x)</td>
</tr>
<tr>
<td>self f i</td>
<td>editread ⊚ i &gt; arr f &gt; editset ⊚ i</td>
</tr>
<tr>
<td>feedback i j</td>
<td>editread ⊚ i &gt; editset ⊚ j &gt; editset ⊚ i</td>
</tr>
</tbody>
</table>
10.4.4 Definedness Laws

In the EditorArrow model we have assumed that editors are only able to operate on values that are fully defined, which was modeled by restricting the access functions read and write to values from the Value domain. This has subsequent consequences for the entire model, which were left implicit in Sect. 10.3. In this section, these consequences will be made explicit by means of formulating definedness laws.

Modeling the definedness behavior of editors has consequences for both the used domains and the meaning function. On the domain level, the value part of an EState must be lifted by explicitly incorporating \( \perp \) in it. When values are constructed with tuples or orisers, multiple lifts may even be necessary. This affects the allowed input (and the produced output) of each editor arrow as follows:

**Definition 10.4.4-5:** (value transformation of editor arrows)

<table>
<thead>
<tr>
<th>editor arrow</th>
<th>allows input</th>
<th>and produces</th>
<th>assuming</th>
</tr>
</thead>
<tbody>
<tr>
<td>arr ( f )</td>
<td>( A )</td>
<td>( B )</td>
<td>( f \in A \rightarrow B )</td>
</tr>
<tr>
<td>( f \gg g )</td>
<td>( A )</td>
<td>( C )</td>
<td>( f \gg g \in B, g \in B \rightarrow C )</td>
</tr>
<tr>
<td>first ( f )</td>
<td>((A \times C)_\perp)</td>
<td>((B \times C)_\perp)</td>
<td>( f \in A \rightarrow B )</td>
</tr>
<tr>
<td>left ( f )</td>
<td>((\text{Either } A C)_\perp)</td>
<td>((\text{Either } B C)_\perp)</td>
<td>( f \in A \rightarrow B )</td>
</tr>
<tr>
<td>iterate ( f )</td>
<td>((\mathbb{N}<em>\perp \times A)</em>\perp)</td>
<td>( A )</td>
<td>( f \in A \rightarrow A )</td>
</tr>
<tr>
<td>editread</td>
<td>( ID_\perp)</td>
<td>Value</td>
<td>–</td>
</tr>
<tr>
<td>editset</td>
<td>((\text{ID}<em>\perp \times A)</em>\perp)</td>
<td>Value</td>
<td>–</td>
</tr>
</tbody>
</table>

Here, \( A_\perp \) denotes \( A \cup \{\perp\} \), and \( f : A \rightarrow B \) denotes that the arrow \( f \) transforms values of type \( A \) to values of type \( B \) (ignoring the other elements of the EState, which are of the same type for all editor arrows). For instance, if \( f : A \rightarrow A \) and \( a \in A \), then \((0, a), \perp\) and \((\perp, a)\) are all valid input for iterate \( f \). Note that editread and editset both produce an element of Value, which is assumed to be the unification set of the defined values of all allowed types. The ‘\( A \)’ input of editset, on the other hand, does not necessarily have to be defined.

The behavior of the editor arrows on all their allowed inputs was described in Sects. 10.3.1 and 10.3.2, and is the same for the denotational and operational semantics. In the case of \( \perp \) values, this behavior can be summarized as follows:

**Case 1:** It does not matter that (part of) the input is \( \perp \), because no structural information is required at that point. Now, computation can continue normally. This case covers the following situations:

\[
\text{arr } f \text{ on } \perp; \ f \gg g \text{ on } \perp; \ \text{first } f \text{ on } (\perp, x) \text{ and } (x, \perp); \\
\text{left } f \text{ on } \text{Left } \perp \text{ and } \text{Right } \perp; \text{ and iterate } f \text{ on } (n, \perp).
\]

**Case 2:** A \( \perp \) occurs where structural information is required, but it is possible to continue computation normally anyway. This case occurs only when first \( f \) is applied on \( \perp \), which is considered to be equal to applying first \( f \) on \((\perp, \perp)\).
Case 3: A ⊥ occurs where structural information is required, and it is not possible to continue computation normally. This case covers the following situations:

- left \( f \) on \( ⊥ \) (cannot decide whether to apply \( f \) or not);
- iterate \( f \) on \( ⊥ \) and \( (⊥, x) \) (cannot decide how many times to apply \( f \));
- editset on \( ⊥ \) (cannot obtain id and value).

In these situations, we have chosen not to produce any result at all.

Case 4: A \( ⊥ \) occurs when either a defined ID or a defined Value is required to access the editable data. This case covers the following situations:

- editread on \( ⊥ \);
- editset on \( (⊥, a) \); and
- editset on \( (id, ⊥) \) (when no event is pending for \( id \)).

Again, in these situations we have chosen not to produce any result at all.

Due to cases 3 and 4, the semantics of editor arrows becomes a partial function that does not always produce an \( EState \) triplet. In order to determine in which situations a result is produced, the following definedness laws can be used:

**Definition 10.4.4.6:** (definedness relation for editor arrows)
\[
\text{Def}(f, A, B) ⇔ ∀a ∈ A∀ev, s, p. ∃b ∈ B∃s′, p′.[f \ ev(s, a, p) = (s′, a′, p′)]
\]

**Definition 10.4.4.7:** (definedness laws for editor arrows)

<table>
<thead>
<tr>
<th>Law</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( f ∈ A → B )</td>
<td>( \text{Def}(f, A, B) )</td>
</tr>
<tr>
<td>( \text{Def}(f, A, B), \text{Def}(g, B, C) )</td>
<td>( \text{Def}(f &gt;&gt;= g, A, C) )</td>
</tr>
<tr>
<td>( \text{Def}(f, A, B) )</td>
<td>( \text{Def}(\text{first } f, A \times C, B \times C) )</td>
</tr>
<tr>
<td>( \text{Def}(f, {⊥}, B) )</td>
<td>( \text{Def}(\text{iterate } f, N \times A, A) )</td>
</tr>
<tr>
<td>( \text{Def}(f, A, B) )</td>
<td>( \text{Def}(\text{editread }, ID, Value) )</td>
</tr>
<tr>
<td>( \text{Def}(f, A, B) )</td>
<td>( \text{Def}(\text{editset }, ID \times Value, Value) )</td>
</tr>
</tbody>
</table>

**For any given editor arrow \( f \), these laws can be used to come up with sets \( A \) and \( B \) such that \( \text{Def}(f, A, B) \) can be inferred. This then shows that \( f \) produces a result as long as its input value is an element of \( A \).**

### 10.5 Programming with Editor Arrows

In this section, we build a direct implementation of the semantic EditorArrow model that was described in Sect. 10.3. The implementation is realized by means of a library in Clean and is named ‘EditorArrowCore’. The library serves two purposes. Firstly, it is a reference implementation: execution in EditorArrowCore results in the abstract desired behavior of an editor arrow, and
execution in \textit{GEC} and \textit{iData} must result in graphical representations of this same abstract behavior. Secondly, it is a basis for formal reasoning, because it allows the laws of Sect. 10.4 to be verified with \textsc{Clean}'s proof assistant \textsc{Sparkle}.

This section is structured as follows. First, we describe the realization of the base editor arrows in Sect. 10.5.1. Then, we define composed arrow operations in Sect. 10.5.2, which are used to make programming with arrows easier. In Sect. 10.5.3, we then express two example programs as editor arrows, and compare their execution behaviors in \textit{EditorArrowCore}, \textit{GEC} and \textit{iData}. Finally, in Sect. 10.5.4 we discuss the formalization in \textsc{Sparkle} of the earlier provided arrow laws, and we compare the definedness of \textit{EditorArrowCore} with respect to the \textit{EditorArrow} model.

### 10.5.1 Base editor arrows in the \textit{EditorArrowCore} library

The \textit{EditorArrowCore} library is a direct implementation of the \textit{EditorArrow} model that was already described concisely in Sect. 10.3. On the top level, it defines the concept of Editable Data and Event Transformers, by means of the following types:

```plaintext
:: EDET a b ::= Event \rightarrow (EState a) \rightarrow (EState b) 1.
:: EState a ::= (EditableData, a, EventStatus) 2.
:: EditableData ::= [(EditorId, SerializedValue)] 3.
:: EventStatus ::= Processed | Pending 4.
:: EditorId ::= (EditorName, InitialValue) 5.
:: EditorName ::= String 6.
:: Event ::= (EditorId, SerializedValue) 7.
:: InitialValue ::= SerializedValue 8.
:: SerializedValue ::= String 9.
```

With respect to the \textit{EditorArrow} model, there are only two differences. Firstly, an association list is used to represent \textit{EditableData} (line 3), instead of an association set. This is of no consequence, because \textit{EditableData} will only be operated on by functions that are guaranteed never to create duplicates.

Secondly, values are serialized to Strings (line 9) before they are stored in the \textit{EditableData} (line 3). Basically, this is a poor man’s solution to implementing stores in which the values can be of arbitrary different types. The serialize and deserialize functions must be provided by the user explicitly, by means of the following class:

```plaintext
class editable a 1.
where 2.
    serialize :: a \rightarrow String 3.
    deserialize :: String \rightarrow a 4.
```

In \textit{EditorArrowCore}, each editor must be overloaded with an instance of the \textit{editable} class. Furthermore, in order for serialized values to work correctly, the instance must also satisfy the following properties:

- \( \forall a.[a = \bot \iff \text{serialize } a = \bot] \); and
Chapter 10: A Common Arrow Based Semantics for GEC and iData

- $\forall s. [s = \perp \iff \text{deserialize } s = \perp]$; and

- $\forall a. [\text{deserialize} (\text{serialize } a) = a]$

The first two properties state that the definedness of serialized values is identical to the definedness of deserialized values, which is necessary to ensure that the definedness properties of the EditorArrow model carry over to EditorArrowCore. The third property is necessary to make sure that editors do not change values on their own. Unfortunately, it is not possible in CLEAN to enforce properties explicitly for all instances of a class. It is therefore the responsibility of the user to provide instances of the editable class that satisfy the required conditions.

In Sect. 10.3.1, a grammar was introduced for editor arrows ($\text{EdArrow ::= arr Fun | EdArrow \ggg EdArrow | ...}$), and a meaning function was defined on top of it. For type technical reasons, this approach cannot be translated to CLEAN directly. The problem is that explicit instantiation of $\text{EdArrow}$ is necessary for the meaning function (i.e. $\llbracket\rrbracket :: (\text{EdArrow } a\ b) \rightarrow EDET a\ b$), but can never be realized because the types of the arrow operations are not unifiable\(^1\).

In EditorArrowCore, each arrow operation is therefore defined directly by means of a function of the appropriate $EDET$ type. This approach is typeable, but has the disadvantage that argument editor arrows can only be typed by means of $EDET$ as well, and are therefore no longer restricted to wellformed arrows ($\in \text{EdArrow}$). This is corrected by making the $EDET$ type abstract. Finally, note that in EditorArrowCore arrows are not defined by means of classes, because in the context of editors we are only interested in the $EState$ instance.

The effect of the arrow operations is simply a transformation of the $EState$ based on an incoming $Event$. First, the standard operations $\ggg$, $arr$ and $first$ are defined:\(^2\)

\begin{align*}
\ggg :: (EDET a\ b) (EDET b\ c) & \rightarrow EDET a\ c & \text{(1)} \\
\ggg f\ g\ event\ state & = (\_,\_,\_); \text{event} & = g\ event\ (f\ event\ state) & \text{(2)} \\
arr :: (a\rightarrow b) & \rightarrow EDET a\ b & \text{(4)} \\
arr f\ event\ (data, a, status) & = (data, f\ a, status) & \text{(5)} \\
first :: (EDET a\ b) & \rightarrow EDET (a,c) (b,c) & \text{(7)} \\
first f\ event\ (data, ac, status) & \$ (data, b, status) = f\ event\ (data, \text{fst}\ ac, status) & \text{(9)} \\
& = (data, (b, \text{snd}\ ac), status) & \text{(10)}
\end{align*}

\(^1\)For instance, $first\ f$ can only be a member of $\text{EdArrow}$ if tuples are always produced, which is undesirable for the other arrow operations.

\(^2\)For reasons of clarity, we have simplified the types of the arrow operations; the actual types are more complex, because in CLEAN the number of type arguments must be equal to the number of function arguments, resulting in for instance: $\ggg :: (EDET a\ b) (EDET b\ c)\ Event\ (EState\ a) \rightarrow EState\ c$
Section 10.5.1: Base editor arrows in the EditorArrowCore library

These functions behave identically to their counterparts in Sect. 10.3. Note that analogously to the operational semantics, \( \ggg \) performs a pattern match on the \texttt{EState} triple (line 2), and \texttt{first} does not perform a pattern match on the input value \texttt{ac} (line 8). This has to do with desired definedness properties, and will be explained further in Sect. 10.5.4.

Next, the operations \texttt{left} (for the realization of choice) and \texttt{iterate} (for the realization of the most basic form of recursion) are defined. Note that \textsc{Clean} defines a function \texttt{iterate} in its standard environment already; the arrow operation is therefore renamed to \texttt{iterateN}.

\[
\begin{align*}
\text{:: } \texttt{Either a b} & = \texttt{Left a} | \texttt{Right b} \quad & (1) \\
\text{left} :: & (\texttt{EDET a b}) \to \texttt{EDET (Either a c) (Either b c)} & (2) \\
\text{left f } \text{event} (\text{data}, \texttt{Left a}, \text{status}) & \ggg (\text{data}, b, \text{status}) = f \text{event} (\text{data}, a, \text{status}) & (4) \\
& = (\text{data}, \texttt{Left b}, \text{status}) & (5) \\
\text{left f } \text{event} (\text{data}, \texttt{Right c}, \text{status}) & = (\text{data}, \texttt{Right c}, \text{status}) & (6) \\
\text{iterateN} :: & (\texttt{EDET (Int,a) a}) \to \texttt{EDET (Int,a) a} & (8) \\
\text{iterateN f } \text{event} (\text{data}, (n, a), \text{status}) & \ggg (\text{data}, a, \text{status}) = (\text{data}, a, \text{status}) & (10) \\
& = f \text{event} (\text{data}, (n, a), \text{status}) & (11) \\
& = \text{iterateN f } \text{event} (\text{data}, (n-1, a), \text{status}) & (12)
\end{align*}
\]

The definition of \texttt{left} is identical to the operational semantics. The definition of \texttt{iterateN} is slightly different, because \textsc{Clean} does not provide a type for natural numbers, but only one for whole numbers (\texttt{Int}). The base case therefore has to check for \( n \leq 0 \) (line 10) instead of \( n = 0 \), and the recursive case goes from \( n \) to \( n - 1 \) (line 12) instead of from \( n + 1 \) to \( n \). Note that the recursion in \texttt{iterateN} always terminates, because the loop variable cannot be changed by the recursive arrow (see line 11: \( n \) is input of \( f \), but not output).

Next, the accessor functions \texttt{read} and \texttt{write} will be defined, which will be used later to describe the operations \texttt{editread} and \texttt{editset}. In the \texttt{EditorArrow} model, the purpose of \texttt{read} and \texttt{write} is twofold: they are not only used to update the editable data, but they are also used to implicitly enforce definedness properties. The required definedness properties of \texttt{read} and \texttt{write} are as follows:

- In the \texttt{EditorArrow} model, \texttt{read} can be regarded as a partial function in the lifted domain that only produces a result for identifiers that are defined. This is then subsequently used to restrict the behavior of \texttt{editread}.

  In \textsc{Clean}, partial functions can be modeled by producing \( \perp \) for the input values that are not in its domain. In \texttt{EditorArrowCore}, \texttt{read} will therefore be defined in such a way that it produces \( \perp \) if \( \texttt{id} = \perp \), and performs the required read operation on the editable data otherwise.

- In the \texttt{EditorArrow} model, \texttt{write} can be regarded as a partial function in the lifted domain that only produces a result for identifiers and values
that are defined. This is then subsequently used to restrict the behavior of \textit{editset}.

In \textit{EditorArrowCore}, \textit{write} will be defined in such a way that it produces \perp if either \textit{id} = \perp or \textit{v} = \perp, and performs the required write operation on the editable data otherwise.

This leads to the following definitions of \textit{read} and \textit{write}:

\begin{align*}
\text{evalString} & : : !\text{String} \to \text{Bool} \\
\text{evalString} \ s & = \text{True} \\
\text{evalEditorId} & : : \text{EditorId} \to \text{Bool} \\
\text{evalEditorId} \ (\text{name}, \text{value}) & = \text{evalString name} \&\& \text{evalString value} \\
\text{read} & : : \text{EditorId} \to \text{EditableData} \to \text{SerializedValue} \\
\text{read} \ \text{id} \ \text{data} & | \not (\text{evalEditorId id}) = \perp \\
& | \text{evalString value} = \text{read}' \ \text{id} \ \text{data} \\
& \quad \text{where} \\
& \quad \text{read}' \ \text{id} \ [\text{record}: \text{data}] \\
& \quad \\
& \quad \\
& \quad | \text{fst record} = \text{id} = \text{snd record} \\
& \quad \text{otherwise} = \text{read}' \ \text{id} \ \text{data} \\
& \quad \text{read}' \ \text{id} \ [] \\
& \quad = \text{snd id} \\
\text{write} & : : \text{EditorId} \to \text{SerializedValue} \to \text{EditableData} \to \text{EditableData} \\
\text{write} \ \text{id} \ \text{value} \ \text{data} & | \not (\text{evalEditorId id}) = \perp \\
& | \not (\text{evalString value}) = \perp \\
& = \text{write}' \ \text{id} \ \text{value} \ \text{data} \\
& \quad \text{where} \\
& \quad \text{write}' \ \text{id} \ \text{value} \ [\text{record}: \text{data}] \\
& \quad \\
& \quad | \text{fst record} = \text{id} = \text{snd record} \\
& \quad \text{otherwise} = [\text{record}: \text{write}' \ \text{id} \ \text{value} \ \text{data}] \\
& \quad \text{write}' \ \text{id} \ \text{value} \ [] \\
& \quad = [(\text{id}, \text{value})]
\end{align*}

The definedness conditions are checked by \textit{read} and \textit{write} on lines 9, 19 and 20. For checking the definedness of a \textit{SerializedValue} (which is actually a \textit{String}), the function \textit{evalString} (lines 1-3) is used. By means of its strictness annotation, it produces \text{True} for defined values and \perp for undefined ones. The definedness of a \textit{EditorId} is checked with \textit{evalEditorId} (lines 4-6), which makes use of pattern matching and translates to two calls of \textit{evalString}. Because of the explicit pattern match, it does not need a strictness annotation in front of its \textit{EditorId} argument.

Using \textit{read} and \textit{write}, the operations \textit{editread} and \textit{editset} can now be defined in \textit{EditorArrowCore} as follows:
Section 10.5.1: Base editor arrows in the EditorArrowCore library

editread :: EDET EditorId a | editable a
| status == Pending && ev_id == id
  | ≠! data = write id v data
  = (data, deserialize v, Processed)
| otherwise
  ≠! read_v = read id data
= (data, deserialize read_v, status)

editset :: EDET (EditorId, a) a | editable a
| status == Pending && ev_id == id
  | ≠! data = write id v data
  = (data, deserialize v, Processed)
| otherwise
  ≠! data = write id (serialize a) data
= (data, a, status)

These functions model the operational semantics directly. The strict lets (denoted by ≠!) on lines 4, 7, 12 and 15 model the definedness conditions imposed by read and write. These strict lets compute a value, and if this value is ⊥ cause editread and editset to produce ⊥ as a whole. Note that as was discussed earlier, explicit conversion to and from SerializedValue is necessary in EditorArrowCore for storing values of different types in a single editable data.

Finally, the execution of an arrow on a scenario is realized by applying events one by one on the arrow. This eventloop is defined in a general way for all editor arrows of type EDET a b. It requires an initial value of type a, which is needed at every event to get started, and it throws away the result value of type b, assuming instead that the editable data is used for transferring information from one event to the next. It also requires an initial editable data.

:: Scenario ::= [Event]

eventloop :: (EDET a b) (EditableData, a) Scenario → EditableData
| event:events
  ≠! (data, _, _) = f event (data, a, Pending)
  = eventloop f (data, a) events
eventloop f (data, a) []
= data

To execute an arrow in EditorArrowCore, it must be wrapped in an application of eventloop. For the initial editable data, [] can be filled in to indicate that all editors should start at their specified initial values. The scenario input corresponds to user actions which must be processed by the arrow and can be chosen freely. The varsumlist arrow of Sect. 10.2.3 can be wrapped in EditorArrowCore as follows:

module varsumlist_EAC

import StdEnv, EditorArrowCore
Chapter 10: A Common Arrow Based Semantics for GEC and iData

Start = eventloop varsumlist ([], ⊥)
   [(nrId, "2"), (argId 1, "30"), (argId 2, "12"),
    (nrId, "1"), (nrId, "3"), (argId 3, "58")]

Note that varsumlist does not use its initial value, therefore ⊥ can be used for it safely (line 3). The user actions of Sect. 10.2.3 have been modeled by a list of six events (lines 3-5). Note that the value in each event must be provided in serialized format.

10.5.2 Derived editor arrows in the EditorArrowCore library

The base arrow operations of EditorArrowCore are sufficiently powerful to express many example programs, but are still rather unfriendly for programming purposes. In this section, a layer of derived arrow operations will therefore be defined on top of the base layer. The derived operations are applications of existing arrows only, and can be used in EditorArrowCore, GEC and iData. In Sect. 10.5.3, the derived operations will be used to construct example programs with ease.

The derived arrow operations consist of useful abbreviations for commonly used functionality, an operation for branching into separate computations, operations for performing choice based on the arrow state, and an arrow version of map. First, abbreviations are introduced for functions that are often lifted to the arrow level:

\[
dupl := \lambda x \rightarrow (x,x)\]
\[
set a := \lambda x \rightarrow a\]
\[
add1 a := \lambda b \rightarrow (a,b)\]
\[
add2 b := \lambda a \rightarrow (a,b)\]

The function \(\text{dupl}\) (line 1) duplicates an arrow state, which is useful if an operation is applied that unwantedly consumes its input. The functions \(\text{set}\), \(\text{add1}\) and \(\text{add2}\) (line 2-4) introduce a constant into an arrow state, which is useful for operations that need constant input only (apply \(\text{set}\) beforehand) and operations that need a combination of state and constant input (apply \(\text{add1}\) or \(\text{add2}\) beforehand).

The following abbreviations introduce special notations for specific applications of \(\text{arr}\) that are often needed in combined arrow expressions:

\[
\text{arr2} f := \text{arr} \ (\lambda (a,b) \rightarrow f\ a\ b)\]
\[
(\otimes) f\ g := \text{arr} \ g \gg f\]

The operation \(\text{arr2}\) (line 1) combines two separately computed values to a single one by means of the application of a function. The infix operation \(\otimes\) (line 2) inserts an \(\text{arr}\) before an arbitrary operation, which is useful if the operation requires a small state transformation to be applicable. In particular, it is handy for providing editor ids to \(\text{editread}\) and \(\text{editset}\) by means of \((\text{editread} \otimes \text{set} \ id)\) and \((\text{editset} \otimes \text{add1} \ id)\), which in Sect. 10.4.3 were even abbreviated further to \(\text{editread} \boxdot id\) and \(\text{editset} \odot id\).
Section 10.5.2: Derived editor arrows in the EditorArrowCore library

Arrows often require separate computations to be carried out independently, after which the results are combined again. This behavior can be achieved by means of first and its dual second, but they both require the arrow state to be a tuple in the first place. In order to conveniently start separate computations from a single value, the operation branch is defined:

\[
\text{second} :: \text{EDET} (a, b) \rightarrow \text{EDET} (c, a) (c, b)
\]

\[
\text{second } f = \text{arr swap} \ggg \text{first } f \ggg \text{arr swap}
\]

where \( \text{swap} (x, y) = (y, x) \)

\[
\text{branch} :: \text{EDET} (a, b) \rightarrow \text{EDET} a (b, c)
\]

\[
\text{branch } f g = \text{arr dupl} \ggg \text{first } f \ggg \text{second } g
\]

The well-known operation second (lines 1-3) is the dual of first and allows an arrow to be executed on the right-hand-side of a tuple only. The operation branch (lines 4-5) duplicates its input value, which in fact creates two separate branches, and executes its first argument on the first branch and its second argument on the second branch. Combining the values afterwards must be performed separately.

For programming purposes, it is important that an arrow operation is available that chooses between computations based on the contents of the arrow state. The base layer does not define such an operation, but it can be expressed in terms of left as follows:

\[
\text{right} :: \text{EDET} b c \rightarrow \text{EDET} (\text{Either } a b) (\text{Either } a c)
\]

\[
\text{right } f = \text{arr swap} \ggg \text{left } f \ggg \text{arr swap}
\]

where \( \text{swap} (\text{Left } a) = \text{Right } a \)

\( \text{swap} (\text{Right } b) = \text{Left } b \)

\[
\text{choice} :: \text{EDET} l b \rightarrow \text{EDET} r b \rightarrow \text{EDET} (\text{Either } l r) b
\]

\[
\text{choice } f g = \text{left } f \ggg \text{right } g \ggg \text{arr remove\_either}
\]

where \( \text{remove\_either} (\text{Left } x) = x \)

\( \text{remove\_either} (\text{Right } x) = x \)

\[
\text{ifthenelse} :: (a \rightarrow \text{Bool}) \rightarrow \text{EDET} a b \rightarrow \text{EDET} a b \rightarrow \text{EDET} a b
\]

\[
\text{ifthenelse } p f g = \text{arr } (\lambda a \rightarrow \text{if } (p a) (\text{Left } a) (\text{Right } a)) \ggg \text{choice } f g
\]

The operation right (lines 1-4) is the dual of left. The standard operation choice (lines 5-8) chooses between its arguments on the basis of the arrow state: a Left triggers execution of the first argument and a Right execution of the second. The operation ifthenelse (lines 9-11) lifts choice to predicates by internally converting to an Either based on the outcome of the predicate.

In a truly functional manner, it is possible to lift basic arrow operations to lists as well. We will demonstrate this by realizing a map in terms of iterateN. The idea is to repeatedly pop the first element of the list, apply the arrow to it and put the transformed element back at the end of the list. This must be iterated exactly as many times as the list is long:

\[
\text{mapA} :: \text{EDET} a a \rightarrow \text{EDET} [a] [a]
\]
map f = arr (λ as → (length as, as)) >>= iterateN (inner_app f)

where inner_app f = arr (λ(_, [a:as]) → (a,as))

>>= first f

>>= arr (λ(a,as) → as ++ [a])

Many other derived applications can of course be defined as well, and the actual EditorArrowCore library contains more operations than are defined in this section. It is not the purpose of this paper to list all these operations, however.

10.5.3 Some Small Editor Arrows Programs

In Sect. 10.2.3, an example editor arrow was described with which the sum of a variable number of editors was computed. Using the derived operations of EditorArrowCore, this editor arrow can now be expressed much more elegantly, as follows:

variable_sum_arrow :: EDET Int Int

variable_sum_arrow

= editread @ (set nrId)

>>= iterateN (first (editread @ argId) >>= arr2 (+)) @ (add2 0)

>>= editset @ (add1 sumId)

The main difference is that all applications of arr which were used to add a constant value to the arrow value have been replaced with applications of @. This is not only more compact, but also describes the intention of these constant values (they are used as fixed input for the next arrow) more clearly.

This editor arrow can be executed in EditorArrowCore. We will use the scenario of Sect. 10.2.3, modeling the user actions with a list of Events. By printing the events and the intermediate states, this results in the following output in EditorArrowCore:

[[]

→ Event(nr, 2)

[ nr=2; sum=0 ]

→ Event(arg 1, 30)

[ nr=2; arg 1=30; sum=30 ]

→ Event(arg 2, 12)

[ nr=2; arg 1=30; arg 2=12; sum=42 ]

→ Event(nr, 1)

[ nr=1; arg 1=30; arg 2=12; sum=30 ]

→ Event(nr, 3)

[ nr=3; arg 1=30; arg 2=12; sum=42 ]

→ Event(arg 3, 58)

[ nr=3; arg 1=30; arg 2=12; arg 3=58; sum=100 ]

The incoming events are shown on lines 2,4,6,8,10 and 12. The editable data, which contain the current values of the editors, are shown on lines 1,3,5,7,9,11 and 13. Note that editors that do not have an entry in the editable data are still at their initial value (which is 0 for all editors in this example). The states at lines 1, 7, 9 and 13 correspond with the screenshots in Sect. 10.2.3.
Another interesting example is a convertor between euro's and dollars. It consists of a euro editor and a dollar editor which are connected in such a way that a change in one editor causes the other editor to be updated. In arrow style, this can be realized by a shared feedback of the form \( \text{euro} \gg\gg \text{dollar} \gg\gg \text{euro} \), as follows:

```haskell
convert_arrow :: EDET a Real
convert_arrow
= editread 0 (set euroId)
  >>> arr toDollar
  >>> editset 0 (add1 dollarId)
  >>> arr toEuro
  >>> editset 0 (add1 euroId)
where
toDollar euro = euro * 1.592
  toEuro dollar = dollar / 1.592
```

Finally, the following editor arrow changes indicated values in a list. It consists of two editors, one to input the index of the element, and one to change its value. The list itself is stored in the arrow state, and is never sent to an editor. Therefore, this example works both for finite and for infinite lists.

```haskell
list_editor :: EDET [a] [a] | editable a
list_editor
= branch (editread 0 set indexId) skip
  >>> arr (λ(i,list) → (list!!i, (i,list)))
  >>> first (editset 0 (add1 fieldId))
  >>> arr (λ(n,(i,list)) → updateAt n in list)
```

## 10.5.4 Arrow laws for Sparkle

By implementing the EditorArrow model in Clean, it also becomes possible to make use of its integrated proof assistant Sparkle \([dvP08a]\). In this section, we will translate the laws of Sect. 10.4 to EditorArrowCore, which allows their correctness to be verified by proving them with Sparkle.

The realization of editor arrows in Clean follows the operational semantics as closely as possible. As a result, there is only one difference between the behavior of EditorArrowCore and EditorArrow. This difference is due to the lazy semantics of Clean, which makes it possible for an editor arrow to get an undefined event, editable data or event status as input. The behavior in these cases has not been defined by the semantics, and may falsify the laws of Sect. 10.4.

If the incoming event, editable data and event status are all defined, then editor arrows in EditorArrowCore behave exactly the same as in the EditorArrow model. By explicitly enforcing these definedness conditions, the laws can be transferred to Sparkle directly. For this purpose, we implement the following eval functions:

```haskell
evalEvent :: Event → Bool
```
194  Chapter 10: A Common Arrow Based Semantics for GEC and iData

evalEvent \( (\text{id}, \text{v}) \)
\begin{align*}
&= \text{evalEditorId} \land \text{evalValue v} \\
\end{align*}

\begin{align*}
evalEState :: (a \rightarrow \text{Bool}) \rightarrow \text{Bool} \\
\text{evalEState eval_a (data, a, status)} \\
&= \text{evalEditableData data} \land \text{eval_a a} \land \text{evalEventStatus status} \\
\end{align*}

\begin{align*}
evalEditableData :: \text{EditableData} \rightarrow \text{Bool} \\
\text{evalEditableData} \quad \text{[]} \\
&= \text{True} \\
\end{align*}

\begin{align*}
evalEventStatus :: \text{EventStatus} \rightarrow \text{Bool} \\
\text{evalEventStatus Pending} = \text{True} \\
\text{evalEventStatus Processed} = \text{True} \\
\end{align*}

Note that evalEditorId and evalValue were already defined in Sect. 10.5.1. The other eval functions are defined here in the same manner. The function evalEState (lines 4-6) has been augmented with a custom eval predicate for values because this additional predicate is needed for translating the definedness laws of Sect. 10.4.4.

The laws of Sect. 10.4 can now be transferred to SPARKLE directly. We demonstrate this for the following three laws:

**Law ‘\( \gg\gg \text{ def} \)’:** \( \text{Def}(f, A, B) \Rightarrow \text{Def}(g, B, C) \Rightarrow \text{Def}(f \gg\gg g, A, C) \)

**Sparkle:**
\begin{align*}
\text{evalEvent ev} \\
&\rightarrow \text{evalEState A state} \\
&\rightarrow ((s \text{[]}, e) \rightarrow \text{evalEState A s} \rightarrow \text{evalEState B (f e s)}) \\
&\rightarrow ((s \text{[]}, e) \rightarrow \text{evalEState B s} \rightarrow \text{evalEState C (g e s)}) \\
&\rightarrow \text{evalEState C ((f \gg\gg g) ev state)} \\
\end{align*}

**Notes:** With additional definedness conditions, the translation of \( \text{Def}(f, A, B) \)
\( \text{is [s] [s] evalEvent e} \rightarrow \text{evalEState A s} \rightarrow \text{evalEState B (f e s)} \). The law in SPARKLE can be obtained by applying this translation three times, and eliminating the outer universal quantors (which are optional in SPARKLE).

**Law ‘\( \ll\ll \text{ assoc eliminates first} \)’:** \( \text{first (first f)} \gg\gg \text{arr assoc} = \text{arr assoc} \gg\gg \text{first f} \)

**Sparkle:**
\begin{align*}
\text{evalEvent ev} \\
&\rightarrow \text{evalEState (A o \text{fst o fst}) state} \\
&\rightarrow ((s \text{[]}, e) \rightarrow \text{evalEState A s} \rightarrow \text{evalEState B (f e s)}) \\
&\rightarrow (\text{first (first f)} \gg\gg \text{arr assoc}) \text{ ev state} \\
&= (\text{arr assoc} \gg\gg \text{first f}) \text{ ev state} \\
\end{align*}

**Notes:** The original law can be found in the last line of the translation. The first two lines ensure that the incoming event and state are defined, and that \( A \) holds for the \( \text{fst} \) of the \( \text{fst} \) of the state. The third line corresponds to \( \text{Def}(f, A, B) \), and ensures that applying \( f \) on the \( \text{fst} \) of the \( \text{fst} \) of the state yields a defined result.

**Law ‘\( \text{read read elimination} \)’:** \( \text{editread} \oplus i \gg\gg \text{editread} \oplus i = \text{editread} \oplus i \)
Section 10.6: Related Work

We have presented a semantic model for interactive applications. The model is inspired on our work on high level toolkits for desktop GUI applications and web applications, viz. the GEC Toolkit and the iData Toolkit. The model uses the same level of abstraction as the toolkits by considering the elementary interactive components as being editors of arbitrary values that can be edited by the user. The elementary elements are glued together by means of the EditorArrow combinator functions. The advantage of using a functional style formalism is that integration of computation can be done within the framework, using functions. Other projects, such as Fruit [CE01] and Fran [HCNP03] have taken this route as well. These systems had to resort to Arrows in order to eliminate subtle performance problems. In our case, we use them chiefly to structure our programs in order to facilitate reasoning.

Another way of modeling interactive programs is to regard them as collections of communicating processes. From this point of view, it seems to be natural to provide a model in terms of a process algebra. There is a wide variety of process algebras available, such as CCS [Mil80], CSP [Hoa85], ACP [BW90], and μCRL [GR01]. Especially the latter might be interesting in this context because it augments ACP with algebraic data types in a spirit that is very similar to functional programming. In general, the fine grained control over concurrency that is usually provided by process algebraic models is not necessary when dealing with interactive applications. We hope to have demonstrated that the use of a disciplined, functional style is well suited to create intricate interactive applications that can still be reasoned about with traditional equational reasoning techniques.

10.7 Conclusions

We have introduced the formal EditorArrow semantic model of the GEC and iData toolkits. This model is based on the Arrow framework. It essentially extends the basic framework with iteration instead of loops and with primitive combinator functions, editread and editset, for creating editors with shared state.
Apart from the classic associated Arrow laws we have formulated a number of additional laws for iteration and for editors. Furthermore, we have introduced definedness laws for the semantic model. This is relevant because the edit combinators impose very strict requirements on their input values, output values and events that are passed through the system, which is in contrast with the requirements of the standard Arrow combinators.

The use of SPARKLE greatly increased confidence in the correctness of the proven laws. In addition, working with this proof assistant helped us to identify issues that escaped our attention in the process of specifying the model and its theorems.
Appendix A

Tactic Library of Sparkle

This appendix provides a short description of the tactics that can be used for building proofs in Sparkle (version 0.0.5a, June 2008). It extends on earlier versions of the tactic library, which appeared as local appendices in [dv99a] and [dvP08a]. These two papers are included as Chapters 2 and 4 of this thesis, but their local versions of the tactic library have been removed and are replaced by this global appendix.

The June 2008 version of Sparkle makes a library of 42 tactics available. In the next section, these tactics will shortly be described one by one. Of each tactic, its general purpose will be explained, and a small typical example of its syntax and its use in practice will be provided. Furthermore, each tactic is briefly categorized as follows:

Equivalence/Strengthening - an equivalence tactic creates new goals that are logically equivalent to the original goal; a strengthening tactic creates goals that are logically stronger.

Forwards/Backwards - a forwards tactic brings hypotheses closer to the current goal; a backwards tactic brings the current goal closer to the hypotheses.

Instantaneous - an instantaneous tactic proves a goal in one single step (and will not be categorized as equivalence/strengthening or forwards/backwards).

Programming/Logic - a programming tactic is based on the semantics of Clean; a logic tactic is based on the semantics of the logical connectives.

Note that the descriptions focus on the typical use of tactics only, and do not necessarily cover all their possible applications.
Absurd <Hyp1> <Hyp2>.  

Type: Instantaneous; logic.

Info: Proves a goal that contains contradictory (absurd) hypotheses.

Details: Hypotheses are contradictory if they are each other’s exact negation.

Example: ⟨H1:¬(p = 12), H2:p = 12⟩ ⊢ FALSE
  ◀Absurd H1 H2.▶
  Q.E.D.

AbsurdEquality <Hyp>.

Type: Instantaneous; programming.

Info: Proves a goal that contains a hypothesis stating an absurd equality.

Details: An equality between two different basic values is absurd, as well as an equality between applications of different lazy constructors.

Example: ⟨H1:True = False⟩ ⊢ FALSE
  ◀AbsurdEquality H1.▶
  Q.E.D.

Notes: True and False are constructors; FALSE is a constant proposition.

Apply <Fact>.

Type: Usually strengthening, depends on fact; backwards; logic.

Info: Applies a fact to the current goal.

Details: A fact is either an earlier proved theorem or an introduced hypothesis, and must be of the form ∀x₁...xₙ.P₁→...Pₘ→Q. It is only valid if r₁...rₙ can be found such that Q[xᵢ → rᵢ] equals the current goal. If this is the case, then the current goal is replaced with the conjunction P₁[xᵢ → rᵢ]∧...∧Pₘ[xᵢ → rᵢ].

Example: p, ⟨H1:∀x∀y∀z.x > 0 → y < z → x + y < x + z⟩ ⊢ 7 + p < 7 + 12
  ◀Apply H1.▶
  p, ⟨H1:∀x∀y∀z.x > 0 → y < z → x + y < x + z⟩ ⊢ 7 > 0 ∧ p < 12

Notes: This tactic can also be applied in a forwards manner. In that case, P₁ must match on a hypothesis R, which is then replaced by P₂ → ...Pₙ → Q.

Assume <Prop>.

Type: Equivalence; forwards; logic.

Info: Assumes the validity of a manually stated proposition.

Details: Two goals are created: one with the assumption as new hypothesis, and one with the hypothesis as goal itself.

Example: P, Q, R, ⟨H1:P → R⟩, ⟨H2:¬P → R⟩ ⊢ R
  ◀Assume P ∨ ¬P.▶
  (2) P, Q, R, ⟨H1:P → R⟩, ⟨H2:¬P → R⟩ ⊢ P ∨ ¬P

Notes: A name for the new hypothesis is generated automatically.
Appendix A: Tactic Library of Sparkle

Case <Hyp>.  

**Type:** Equivalence; backwards; logic.

**Info:** Breaks down an introduced disjunction.

**Details:** The hypothesis must be of the form $P \lor Q$. Two goals are created: one in which $P \lor Q$ is replaced with $P$, and one in which it is replaced with $Q$.

**Example:** $P, Q, \langle H_1: P \lor \neg P \rangle, \langle H_2: P \rightarrow Q \rangle, \langle H_3: \neg P \rightarrow Q \rangle \vdash Q$

- **Case H1.**
  - (1) $P, Q, \langle H_1: P \rangle, \langle H_2: P \rightarrow Q \rangle, \langle H_3: \neg P \rightarrow Q \rangle \vdash Q$
  - (2) $P, Q, \langle H_1: \neg P \rangle, \langle H_2: P \rightarrow Q \rangle, \langle H_3: \neg P \rightarrow Q \rangle \vdash Q$

Cases <Expr>.  

**Type:** Equivalence; programming.

**Info:** Performs a case distinction on a given expression.

**Details:** The expression must be of an algebraic type. New goals are created for each of its constructors, and one for $\bot$ as well. Each new goal is obtained by replacing all occurrences (also in the hypotheses) of the indicated expression with a generic application of the constructor.

**Example:** $xs, ys, \langle H_1: \text{length} (xs ++ ys) > 0 \rangle \vdash \neg(xs ++ ys = [])$

- **Cases (xs ++ ys).**
  - (1) $\langle H_1: \text{length } \bot > 0 \rangle \vdash \neg(\bot = [])$
  - (2) $\langle H_1: \text{length } [] > 0 \rangle \vdash \neg([] = [])$
  - (3) $x_1, x_2, \langle H_1: \text{length } [x_1:x_2] > 0 \rangle \vdash \neg([x_1:x_2] = [])$

**Notes:** Names for the newly introduced variables are generated automatically.

ChooseCase.  

**Type:** Equivalence; programming.

**Info:** Simplifies a case distinction in which only one pattern is valid.

**Details:** The goal must be of the form $E_1 = E_2$, where $E_1$ is a case distinction and $E_2$ is a basic value. A pattern is valid if its result is not statically unequal to $E_2$. The tactic succeeds only if there is exactly one valid pattern. The case is then simplified to the result of the single valid pattern, and its condition is introduced as a conjunction in the goal.

**Example:** $n \vdash \text{case } n \text{ of } (7 \mapsto 13; 13 \mapsto 7; n \mapsto 11) = 13$

- **ChooseCase.**
  - $n \vdash n = 7 \land 13 = 13$
Appendix A: Tactic Library of Sparkle

**Compare <Expr1> with <Expr2>**.  
(*#8*)

**Type:** Equivalence; backwards; logic.  
**Info:** Distinguishes between the possible compare results of two expressions.  
**Details:** The expressions must both be of type Int. Five new goals are created: one for \(E_1 = \bot\), one for \(E_2 = \bot\), one for \(E_1 < E_2\), one for \(E_1 = E_2\) (provided that \(E_1\) and \(E_2\) are not \(\bot\)), and one for \(E_2 < E_1\).  
**Example:** \(m, n \vdash \min m n \leq \max m n\)

\[\begin{align*}
1.\; m, n \vdash m = \bot &\rightarrow \min m n \leq \max m n \\
2.\; m, n \vdash n = \bot &\rightarrow \min m n \leq \max m n \\
3.\; m, n \vdash m < n &\rightarrow \min m n \leq \max m n \\
4.\; m, n \vdash \neg (m = \bot) &\rightarrow \neg (n = \bot) &\rightarrow m = n &\rightarrow \min m n \leq \max m n \\
5.\; m, n \vdash n < m &\rightarrow \min m n \leq \max m n
\end{align*}\]

**Contradiction.**  
(*#9*)

**Type:** Equivalence; backwards; logic.  
**Info:** Builds a proof by contradiction.  
**Details:** Replaces the current goal by the absurd proposition \(\text{FALSE}\) and adds its negation as a hypothesis in the context. If a double negation is produced, it will be removed automatically.  
**Example:** \(P, \langle H1: P \rightarrow \text{FALSE} \rangle \vdash \neg P\)

\[\begin{align*}
P, \langle H1: P \rightarrow \text{FALSE} \rangle, \langle H2: P \rangle &\vdash \text{FALSE}
\end{align*}\]

**Notes:** A name for the new hypothesis is generated automatically. This tactic can also be applied in a forwards manner on a hypothesis. In that case, the negation of the hypothesis simply becomes the new goal to prove.  

**Cut <Fact>.**  
(*#10*)

**Type:** Equivalence; backwards; logic.  
**Info:** Duplicates a fact.  
**Details:** A fact is either an earlier proved theorem or an introduced hypothesis. It is added to the to prove by means of a new implication.  
**Example:** \(\langle H1: \forall P. P \vee \neg P \rangle \vdash \text{FALSE}\)

\[\begin{align*}
\langle H1: \forall P. P \vee \neg P \rangle &\vdash (\forall P. P \vee \neg P) \rightarrow \text{FALSE}
\end{align*}\]
**Definedness.**  (#11)

**Type:** Instantaneous; logic.

**Info:** Uses contradictory definedness information to prove a goal.

**Details:** Two sets of expressions are determined: (1) those that are statically known to be equal to \(\perp\); (2) those that are statically known to be unequal to \(\perp\). These sets are determined by examining equalities in hypotheses and using strictness information. In addition, the totality of certain predefined functions is used. If an overlap between the two sets is found, the goal is proved immediately.

**Example:** \(xs, ys, zs, \langle H1: xs = \perp \rangle, \langle H2: xs ++ ys = [1: zs] \rangle \vdash \text{FALSE}\)

\[\text{Definedness.}\]

\[\text{Q.E.D.}\]

**Notes:** In the example, \(xs = \perp\) due to \(H1\), and \(\neg(xs = \perp)\) due to the strictness of ++ and the definedness of the result of \(xs ++ ys\) by means of \(H2\).

---

**Discard <Hyp>.**  (#12)

**Type:** Strengthening; logic.

**Info:** Deletes an introduced hypothesis.

**Example:** \(x, xs, \langle H1: \text{reverse } [] = [] \rangle \vdash \text{reverse } [x:xs] = \text{reverse } xs ++[x]\)

\[\text{Discard } H1.\]

\[x, xs \vdash \text{reverse } [x:xs] = \text{reverse } xs ++[x]\]

---

**Exact <Hyp>.**  (#13)

**Type:** Instantaneous; logic.

**Info:** Proves a goal that is identical to an introduced hypothesis.

**Example:** \(\langle H1: \forall P \forall Q. (P \land Q) \rightarrow P \rangle \vdash \forall P \forall Q. (P \land Q) \rightarrow P\)

\[\text{Exact } H1.\]

\[\text{Q.E.D.}\]

---

**ExFalso <Hyp>.**  (#14)

**Type:** Instantaneous; logic.

**Info:** Proves a goal that contains a hypothesis stating FALSE.

**Example:** \(\langle H1: \text{FALSE} \rangle \vdash 5 = 6\)

\[\text{ExFalso } H1.\]

\[\text{Q.E.D.}\]

---

**Extensionality <Name>.**  (#15)

**Type:** Equivalence; backwards; logic.

**Info:** Proves equality of functions by means of extensionality.

**Details:** The current goal must of the form \(E_1 = E_2\), and both \(E_1\) and \(E_2\) must be functions. The goal is then replaced with \(\forall \text{Name}. (E_1 \text{ Name}) = (E_2 \text{ Name})\).

**Example:** \(\vdash (++ []) = \text{id}\)

\[\text{Extensionality } xs.\]

\[\vdash \forall xs. [] ++ xs = \text{id} \text { xs}\]

**Notes:** To prevent proving \(\perp = \lambda x. \perp\), which is not valid, additional definedness conditions are created under certain conditions.
Appendix A: Tactic Library of Sparkle

**Generalize <Expr> to <Name>**. (#16)

**Type:** Strengthening; backwards; logic.

**Info:** Generalizes an arbitrary subexpression.

**Details:** In the to prove, replaces all occurrences of the indicated expression with the variable Name. Then, adds the quantor \(\forall Name\) in front of it.

**Example:**

\[ \text{xs} \vdash (\text{reverse xs} \; \odot \; []) = \text{reverse xs} \]

\[ \vdash \forall ys. \text{ys} \; \odot \; [] = \text{ys} \]

**Induction <Var>**. (#17)

**Type:** Strengthening; backwards; programming.

**Info:** Performs structural induction on a variable.

**Details:** The type of the indicated variable must be \(\text{Int}, \text{Bool}\) or algebraic. A goal is created for each root normal form (RNF) the variable may have, which includes \(\bot\). The RNFs of an algebraic type are determined by its constructors. In each created goal, the variable is replaced by its corresponding RNF. Universal quantors are created for new variables. Additionally, induction hypotheses are added (as implications) for all recursive variables.

**Example:**

\[ \vdash \forall xs. \text{xs} \; \odot \; [] = \text{xs} \]

\[ \vdash \forall x. \forall xs. (\text{xs} \; \odot \; [] = \text{xs}) \rightarrow [x : \text{xs}] \; \odot \; [] = [x : \text{xs}] \]

**Injective**. (#18)

**Type:** Strengthening; backwards; logic.

**Info:** Proves equality of applications by making use of injectivity.

**Details:** Replaces a goal of the form \((S E_1 \ldots E_n) = (S' E'_1 \ldots E'_n)\), where \(S\) is either a function or a constructor, with the conjunction \(E_1 = E'_1 \land \ldots \land E_n = E'_n\).

**Example:**

\[ \text{x}, \text{y} \vdash \text{xs} \; \odot \; [] = \text{xs} \; \odot \; \text{ys} \]

\[ \vdash \text{xs} = \text{xs} \land [] = \text{ys} \]

**Notes:** This tactic can also be applied in a forwards manner on a hypothesis.

**IntArith**. (#19)

**Type:** Equivalence; backwards; logic.

**Info:** Built-in simplification of arithmetic expressions.

**Details:** This tactic operates on expressions containing applications of \(+, -, \times\) on integers. It performs three simplifications: (1) \(a \times (b + c)\) is replaced with \(a \times b + a \times c\); (2) constants are moved to the right as much as possible; and (3) computations on constants are carried out statically.

**Example:**

\[ \text{x}, \text{y} \vdash 3 + 7 \times (12 + x) - 100 = \text{y} \]

\[ \vdash 7 \times x - 13 = \text{y} \]

**Notes:** This tactic can also be applied in a forwards manner on a hypothesis.
Appendix A: Tactic Library of Sparkle

IntCompare.  (#20)

Type: Instantaneous; logic.
Info: Proves goals with contradictory integer comparisons.
Details: Only hypotheses of the exact form \( x < y \) are used as input. If a chain \( x < y < \ldots < x \) can be found, then the goal is proved immediately.
Example: \( x, y, z, (H1:y < x), (H2:z < y), (H3:x < z) \vdash \text{FALSE} \)
\[ \text{IntCompare.} \]
Q.E.D.

Introduce <Name1> <Name2> ... <Namen>.  (#21)

Type: Equivalence; backwards; logic.
Info: Introduces universally quantified variables and hypotheses in the goal.
Details: The current goal must be of the form \( \forall x_1 \ldots x_a.P_1 \rightarrow \ldots \rightarrow P_b \rightarrow Q \), where \( a + b = n \). The quantors and implications may be mixed. The variables \( x_1 \ldots x_a \) and the hypotheses \( P_1 \ldots P_b \) are deleted from the current goal and are added to the goal context using the names given.
Example: \( \vdash \forall x.(x = 7 \rightarrow \forall y.(y = 7 \rightarrow x = y)) \)
\[ \text{Introduce p H1 q H2.} \]
\[ p, q, (H1:p = 7), (H2:q = 7) \vdash p = q \]

MakeUnique.  (#22)

Type: Equivalence; logic.
Info: Makes all variable names in the goal unique.
Details: Effects both introduced and bound variables. Also makes the names unique of bound variables that are not in the same scope.
Example: \( \langle H1:\forall n.n + 0 = n \rangle, \langle H2:m = m \rangle \vdash \forall n_m.m + n = n + m \)
\[ \text{MakeUnique.} \]
\[ \langle H1:\forall n.n + 0 = n \rangle, \langle H2:m + m \rangle \vdash \forall m_1.m_1 + n_1 = n_1 + m_1 \]

ManualDefinedness <Theorem-Name>.  (#23)

Type: Special.
Info: Extends the definedness analysis with manual definedness information.
Details: The indicated theorem must be of the form \( \forall x_1 \ldots x_n.\neg(x_1 = \bot) \rightarrow \ldots \rightarrow \neg(x_n = \bot) \rightarrow \neg(f \ x_1 \ldots x_n = \bot) \). Two rules are added to the definedness analysis:
(1) \( \{e_1 \ldots e_n\} \subseteq D \rightarrow (f \ e_1 \ldots e_n) \in D \) and (2) \( (f \ e_1 \ldots e_n) \in U \rightarrow \exists i [e_i \in U] \), where \( D \) means 'is known to be defined' and \( U \) 'is known to be undefined'.
Example: \( xs, ys, (H1:xs ++ ys = \bot), (H2:ys = \bot) \vdash ys = \bot \)
\[ \text{ManualDefinedness manual; Definedness.} \]
Q.E.D.
(assuming that theorem manual proves that \( \forall xs \forall ys.\neg(xs = \bot) \rightarrow \neg(ys = \bot) \rightarrow \neg(xs ++ ys = \bot) \) holds)

Notes: The extension of the definedness analysis only holds for the current branch of the current proof.
Appendix A: Tactic Library of Sparkle

**MoveQuantors <Dir>**.  
*Type:* Equivalence; backwards; logic.  
*Info:* Swaps implications and universal quantifications.  
*Details:* The direction argument is either ‘In’ or ‘Out’. When moving inwards, goals of the form $\forall x_1...x_n.P_1 \rightarrow ... \rightarrow P_m \rightarrow Q$ are transformed to $P_1 \rightarrow ... \rightarrow P_m \rightarrow \forall x_1...x_n.Q$, provided that none of the $x_i$ occur in any of the $P_j$. The outwards move is the opposite of the inwards move.  
*Example:* $R \vdash \forall P \forall Q.R \rightarrow \neg R \rightarrow P \land Q$  
\begin{verbatim}
\textbar MoveQuantors In.\textbar
R \vdash R \rightarrow \neg R \rightarrow \forall P \forall Q.P \land Q
\end{verbatim}  
*Notes:* This tactic can also be applied in a forwards manner on a hypothesis.

**Opaque <Fun>**.  
*Type:* Special.  
*Info:* Marks a function as non-expandable.  
*Details:* When a function is marked opaque, it will not be expanded by the reduction mechanism. Instead, reduction will stop.  
*Example:* $\vdash \text{zip}(\emptyset, \emptyset) = \emptyset$  
\begin{verbatim}
\textbar Opaque \text{zip}; Reduce NF All.\textbar
\textbar zip2 \text{ [] [] = []} \end{verbatim}  
*Notes:* The opaqueness only holds for the current branch of the current proof.

**Reduce NF All**.  
*Type:* Equivalence; backwards; programming.  
*Info:* Reduces all expressions in the current goal to normal form.  
*Details:* All redexes in the current goal are replaced by their reducts. This full reduction is accomplished by first using standard reduction to root normal form, and then continuing recursively on the top-level arguments.  
*Example:* $\vdash \text{reverse}[7 * 12, 100 - 12] = [89-1, 83+1]$  
\begin{verbatim}
\textbar Reduce NF All.\textbar
\textbar [88, 84] = [88, 84] \end{verbatim}  
*Notes:* This tactic can also be configured to reduce $n$ steps; or to reduce to root normal form; or to reduce a specific redex; or to reduce within a hypothesis. In order to safely handle non-terminating functions, an artificial limit is imposed on the maximum number of reduction steps.
**Appendix A: Tactic Library of Sparkle**

---

**RefineUndefinedness.**  
*Type:* Equivalence; backwards; logic.  
*Info:* Refines undefinedness equalities.  
*Details:* Attempts to refine all undefinedness equalities in the current goal of the form $(S \ E_1 \ldots E_n) = \bot$, where $S$ is either a constructor or a halting function. Replaces the equality with the disjunction of all $E_i = \bot$ where $E_i$ is on a strict position and not statically known to be defined.  
*Example:*  
\[ x, y \vdash (x + y) - 13 = \bot \]  
\[ \langle \text{RefineUndefinedness.} \rangle \]  
\[ x, y \vdash (x + y) = \bot \]  
*Notes:* This tactic can also be applied in a forwards manner on a hypothesis.

**Reflexive.**  
*Type:* Instantaneous; logic.  
*Info:* Utilizes the reflexivity of the built-in operators $=$ and $\leftrightarrow$.  
*Details:* Immediately proves any inner goal of the form $E = E$ or $P \leftrightarrow P$. The tactic automatically looks beyond unintroduced quantors and implications that appear in front of the statement.  
*Example:*  
\[ \vdash \forall x \exists y. x < y \rightarrow x + y = x + y \]  
\[ \langle \text{Reflexive.} \rangle \]  
Q.E.D.

**Rename <Name1> to <Name2>.**  
*Type:* Special.  
*Info:* Renames an introduced variable or an introduced hypothesis.  
*Example:*  
\[ x, y \vdash x < y \rightarrow \neg (x = y) \]  
\[ \langle \text{Rename x to z.} \rangle \]  
\[ z, y \vdash z < y \rightarrow \neg (z = y) \]

**Rewrite <fact>.**  
*Type:* Usually strengthening, depends on fact; backwards; logic.  
*Info:* Rewrites the current goal using an equality in a fact.  
*Details:* A fact is either an earlier proved theorem or an introduced hypothesis, and must be of the form $\forall x_1 \ldots x_n. P_1 \rightarrow \ldots P_m \rightarrow Q$, where $Q$ is either $L = R$ or $L \leftrightarrow R$. It is only valid if $r_1 \ldots r_n$ can be found such that $L[x_i \mapsto r_i]$ occurs within the to prove. If this is the case, then all occurrences of $L[x_i \mapsto r_i]$ are replaced with $R[x_i \mapsto r_i]$. Furthermore, goals are created for each condition of the fact; the i-th states $P_i[x_i \mapsto r_i]$.  
*Example:*  
\[ p, (H1: \forall x. \neg(x = \bot) \rightarrow x * 0 = 0) \vdash (p - 7) * 0 = 0 \]  
\[ \langle \text{Rewrite H1.} \rangle \]  
\[ (1) \ p, (H1: \forall x. \neg(x = \bot) \rightarrow x * 0 = 0) \vdash 0 = 0 \]  
\[ (2) \ p, (H1: \forall x. \neg(x = \bot) \rightarrow x * 0 = 0) \vdash -(p - 7) = \bot \]  
*Notes:* This tactic can also be configured to rewrite from right to left; or to rewrite at one specific location only; or to rewrite within a hypothesis.
Appendix A: Tactic Library of Sparkle

Specialize <Hyp> with <Expr>/<Prop>.

**Type:** Strengthening; forwards; logic.
**Info:** Specializes a universally quantified hypothesis.
**Details:** The hypothesis must be \( \forall x.P \), and the given expression/proposition \( r \) must have the same type as \( x \). Then, the hypothesis is replaced with \( P[x \mapsto r] \).

**Example:**
\[
\begin{align*}
\forall x \forall y \forall z, \langle H1: x < y \rangle & , \langle H2: y < z \rangle , \langle H3: x < y \rightarrow y < z \rightarrow x < z \rangle \vdash x < z \\
\text{Specialize H3 with y.} & \\
x, y, z, \langle H1: x < y \rangle , \langle H2: y < z \rangle , \langle H3: x < y \rightarrow y < z \rightarrow x < z \rangle \vdash x < z
\end{align*}
\]

Split.

**Type:** Equivalence; backwards; logic.
**Info:** Splits a conjunction into separate goals.

**Example:**
\[
\begin{align*}
P, Q, \langle H1: P \rangle , \langle H2: Q \rangle \vdash P \land Q \\
\text{Split.} & \\
(1) P, Q, \langle H1: P \rangle , \langle H2: Q \rangle \vdash P \\
(2) P, Q, \langle H1: P \rangle , \langle H2: Q \rangle \vdash Q
\end{align*}
\]

**Notes:** This tactic can also be applied in a forwards manner on a hypothesis.

SplitCase <Num>.

**Type:** Strengthening; backwards; programming.
**Info:** Splits a case expression into its alternatives.

**Details:** The case expression that will be split is indicated by means of an index (cases are numbered from left to right starting with 1). A new goal is created for each of the alternatives of the case, including one for \( \bot \) and one for the default. In each goal, the case expression is replaced by the result of the alternative. Hypotheses are introduced to indicate which alternative was chosen.

**Example:**
\[
\begin{align*}
xs, \langle H1: \neg(xs = \bot) \rangle \vdash \text{case } xs \text{ of } ([y: ys] \mapsto y; \_ \mapsto 12) > 0 \\
\text{SplitCase 1.} & \\
(1) xs, \langle H1: \neg(xs = \bot) \rangle , \langle H2: xs = \bot \rangle \vdash \bot > 0 \\
(2) xs, y, ys, \langle H1: \neg(xs = \bot) \rangle , \langle H2: xs = [y: ys] \rangle \vdash y > 0 \\
(3) xs, \langle H1: \neg(xs = \bot) \rangle , \langle H2: xs = [\_] \rangle \vdash 12 > 0
\end{align*}
\]

SplitIff.

**Type:** Equivalence; backwards; logic.
**Info:** Splits a \( \leftrightarrow \) into a \( \rightarrow \) and a \( \leftarrow \).

**Details:** The current goal must be of the form \( P \leftrightarrow Q \). Two goals are created, one for with \( P \rightarrow Q \) and one for \( Q \rightarrow P \).

**Example:**
\[
\begin{align*}
P, Q, \langle H1: P \rightarrow Q \rangle , \langle H2: Q \rightarrow P \rangle \vdash P \leftrightarrow Q \\
\text{SplitIff.} & \\
(1) P, Q, \langle H1: P \rightarrow Q \rangle , \langle H2: Q \rightarrow P \rangle \vdash P \rightarrow Q \\
(2) P, Q, \langle H1: P \rightarrow Q \rangle , \langle H2: Q \rightarrow P \rangle \vdash Q \rightarrow P
\end{align*}
\]

**Notes:** This tactic can also be applied in a forwards manner on a hypothesis.
Appendix A: Tactic Library of Sparkle

Symmetric. (§#35)

**Type:** Equivalence; backwards; logic.

**Info:** Utilizes the symmetry of the built-in operators = and ↔.

**Details:** Replaces any inner goal of the form $E_1 = E_2$ with $E_2 = E_1$, and any inner goal of the form $P \leftrightarrow Q$ with $Q \leftrightarrow P$. The tactic automatically looks beyond unintroduced quantors and implications that appear in front of the statement.

**Example:**
$x, \langle H1 : x = y \rangle \vdash y = x$

\[\begin{array}{c}
\text{Symmetric.}
\end{array}\]

$x, \langle H1 : x = y \rangle \vdash x = y$

**Notes:** This tactic can also be applied in a forwards manner on a hypothesis.

Transitive <Expr>/<Prop>. (§#36)

**Type:** Equivalence; backwards; logic.

**Info:** Utilizes the transitivity of the built-in operators = and ↔.

**Details:** Replaces any goal of the form $E_1 = E_2$ with $E_1 = E$ and $E = E_2$ (where $E$ is the argument of the tactic), or any goal of the form $P_1 \leftrightarrow P_2$ with $P_1 \leftrightarrow P$ and $P \leftrightarrow P_2$ (where $P$ is the argument of the tactic).

**Example:**
$P \vdash P \leftrightarrow ((P \land P) \land P)$

\[\begin{array}{c}
\text{Transitive} \ (P \land P).
\end{array}\]

(1) $P \vdash P \leftrightarrow (P \land P)$

(2) $P \vdash (P \land P) \leftrightarrow ((P \land P) \land P)$

Transparent. (§#37)

**Type:** Special.

**Info:** Marks a function as expandable.

**Details:** Undoes the effect of Opaque.

**Example:**
$\vdash \text{zip } (\emptyset, \emptyset) = \emptyset$

\[\begin{array}{c}
\text{Opaque zip2; Transparent zip2; Reduce NF All.}
\end{array}\]

$\vdash \emptyset = \emptyset$

Trivial. (§#38)

**Type:** Instantaneous; logic.

**Info:** Proves the trivial proposition TRUE.

**Details:** Immediately proves any inner goal of the form TRUE. The tactic automatically looks beyond unintroduced quantors and implications that appear in front of the statement.

**Example:**
$\vdash \forall p. P \rightarrow \neg P \rightarrow \text{TRUE}$

\[\begin{array}{c}
\text{Trivial.}
\end{array}\]

Q.E.D.
**Appendix A: Tactic Library of SPARKLE**

**Uncurry.**

*Type:* Equivalence; backwards; programming.

*Info:* Uncurries all applications in the current goal.

*Details:* Forces all curried applications \((f \ x_1 \ldots \ x_i \ x_{i+1} \ldots x_n)\) in the current goal to be uncurried to \(f \ x_1 \ldots x_n\).

*Example:* \(\vdash [((+) 1) 1 : \text{map} ((+) 1) []] = [2] \)

\(<\text{Uncurry.}>\)

\(\vdash [1 + 1 : \text{map} ((+) 1) []] = [2] \)

*Notes:* This tactic can also be applied in a forwards manner on a hypothesis.

**Undo <num>**.

*Type:* Special.

*Info:* Undoes the last \(n\) steps of the proof.

*Details:* SPARKLE does not memorize the last actions of the user. Instead, \(n\) upwards steps in the proof tree are made.

*Example:* \(\vdash \forall x. xs ++ [] = [] \)

\(<\text{Induction} \; xs; \text{Reduce.} \; \text{Undo} \; 2.>\)

\(\vdash \forall x. xs ++ [] = [] \)

**Unshare <var>.**

*Type:* Equivalence; backwards; programming.

*Info:* Unshares a single let binding.

*Details:* Replaces an occurrence of a variable \(x\) with its shared expression \(E\), if it appears in the context of the let binding \(x = E\).

*Example:* \(\vdash \; \text{let} \; x = 7, y = 12 \; \text{in} \; x + y = 19 \)

\(<\text{Unshare} \; y.>\)

\(\vdash \; \text{let} \; x = 7, y = 12 \; \text{in} \; x + 12 = 19 \)

*Notes:* A single occurrence can be replaced, or all occurrences at once. Also, this tactic can be applied in a forwards manner on a hypothesis.

**Witness <Expr>/<Prop>.**

*Type:* Strengthening; backwards; logic.

*Info:* Chooses a witness for an existentially quantified goal.

*Details:* The current goal must be of the form \(\exists x. P\), and \(P[x \mapsto T]\) (where \(T\) is the term argument) must be well-typed. If this is the case, then the goal is replaced with \(P[x \mapsto T]\).

*Example:* \(\vdash \exists x. x * x = x \)

\(<\text{Witness} \; 1.>\)

\(\vdash 1 * 1 = 1 \)

*Notes:* This tactic can also be applied in a forwards manner on a hypothesis.
Summary

In our modern day life, we often make use of computers and computer software. We have learned a great deal about producing software of high quality, and we can therefore rely on computers for carrying out many important tasks in our society. Software is complex, however, and is still developed by humans, who tend to make mistakes. Errors in software can therefore never be avoided completely. Unfortunately, failure of software has become a well-known part of our lives as well.

Detecting software bugs as early as possible is an important activity with which problems can be detected before they can do damage. This thesis focuses on one particular method of finding software bugs, namely formal reasoning. From a theoretical perspective, this method shows great promise, as it is able to find all deviations from specified desired behavior. Unfortunately, this comes at a severe cost too.

The main idea of formal reasoning is to express the desired behavior of a program by means of formal properties, and to prove the correctness of these properties with formal logic. This entire process takes place on the textual level of the source code, and the program itself is never executed. The proof has to take all possible choices in the source code into account. The amount of choices made in software is usually gigantic, however, which causes formal reasoning to scale badly.

The used programming language has a great influence on the complexity of formal reasoning. The closer the language is to mathematics, the easier reasoning becomes. Particularly suited are functional programming languages, in which variables and assignments are not allowed at all. This makes it possible to determine the meaning of a piece of code independent of the context in which it occurs, a feature that makes formal reasoning much easier.

This thesis researches the applicability of formal reasoning in the context of the functional programming language Clean. For this research, the dedicated proof assistant Sparkle has been developed, a computer program that supports the process of building formal proofs. Sparkle is geared towards Clean programmers specifically, and has been written entirely in Clean itself. Currently, a Haskell front-end for Clean is in development; with it, Sparkle will become available for reasoning about Haskell programs as well.

The first three chapters of this thesis provide a description of Sparkle. Chapter 2 describes its prototype, CleanProverSystem. Chapter 3 focuses
Summary

on the dedicated nature of Sparkle, which is integrated into the IDE and offers reasoning steps that are tailored towards Clean. Chapter 4 provides an extensive tutorial of the use of Sparkle in practice.

An important feature of both Clean and Haskell is lazy evaluation, which entails that computations are only performed when their result is really needed. Lazy evaluation allows infinite structured to be used, but also makes it necessary for semantics to take the undefined case ‘⊥’ into consideration explicitly. This has a profound effect on reasoning, in which definedness conditions of the form \( \neg (E = ⊥) \) will appear frequently. Moreover, Clean, and to a lesser extent Haskell as well, allow strictness annotations to be placed in the program, which subtly influence the propagation of ⊥ values.

Dealing with definedness conditions is cumbersome, and is often omitted in informal reasoning. We, however, consider it an integral part of reasoning about lazy functional programs. We have therefore built extensive support in Sparkle for dealing with definedness, which is most noticeable in the specialized behavior of the reasoning steps Reduce, Induction, Cases and Definedness. This support makes dealing with definedness much easier.

In this thesis, definedness is treated in Chapters 5 and 6. Chapter 5 treats the practical consequences in the form of specialized reasoning support. Chapter 6 focuses on the theoretical consequences, by incorporating ⊥ in the standard semantics of Launchbury[La93].

The semantics of Sparkle are an extension of the standard graph-rewriting semantics of Clean. Evaluation is defined by means of a custom single-step term-graph reduction system, which is confluent and behaves equivalently to Launchbury’s system[La93]. On top of it, a standard first-order proposition logic is defined. Equality on expressions is added to this logic by means of equivalence of observable program output. This equality is referentially transparent, and is capable of dealing with infinite and undefined reductions.

In this thesis, semantics is treated in Chapters 7 and 8. Chapter 7 defines the custom reduction system, and Chapter 8 defines equality in terms of it.

The introduction of Sparkle has made reasoning about Clean-programs much easier: its dedicated reasoning steps are intuitive to apply, its integration in the IDE removes the threshold for starting with formal reasoning, and its tactic suggestions allow small proofs to be built effortlessly. Properties such as those from Bird's Introduction to Functional Programming[Bir98] can be proved in Sparkle with very little effort indeed. For large proofs, however, scaling issues still exist.

Since its introduction, Sparkle has been used in various research projects. With my assistance, it has been extended with reasoning support for type classes (Chapter 9 of this thesis) and generalized induction ([Lv04]), and it has been used for reasoning about an arrow based semantic model of the GEC and iData toolkits (Chapter 10 of this thesis). Sparkle has also been used by others. In Budapest, a customized version of Sparkle named Sparkle-T ([THK05, THK06]) has been built which supports temporal reasoning. Finally, Sparkle has been used for proving properties of I/O-models both in Dublin ([DBv04, DBv05]) and in Budapest ([TKH08]).
Samenvatting

In ons moderne leven maken we geregeld gebruik van computers en software. We hebben veel geleerd over hoe we software van hoge kwaliteit kunnen produceren, en we kunnen daarom veel belangrijke taken in onze samenleving door computers laten uitvoeren. Echter, software is vaak complex, en wordt nog steeds geschreven door zeker niet onfeilbare mensen. Fouten in software zijn daarom nooit volledig te voorkomen. Helaas kijken we er vaak niet eens meer van op als een stuk software niet goed functioneert.

Het zo vroeg mogelijk sporen van fouten in software is een belangrijke activiteit waarmee problemen kunnen worden opgespoord voor ze schade aanrichten. Dit proefschrift richt zich op een specifieke methode om fouten op te sporen, namelijk *formeel redeneren*. Deze methode is vanuit een theoretisch oogpunt veelbelovend, want *alle* afwijkingen van het gewenste gedrag kunnen ermee gevonden worden. Helaas brengt dit ook grote kosten met zich mee.

De achterliggende gedachte van formeel redeneren is om het gewenste gedrag van software uit te drukken in formele eigenschappen, en om dan de correctheid van deze eigenschappen te bewijzen met behulp van formele logica. Dit proces vindt volledig plaats op het tekstuele niveau van de broncode, en het programma hoeft nooit uitgevoerd te worden. Het bewijs moet rekening houden met alle mogelijke keuzes die in de broncode worden gemaakt. Omdat in grotere programma’s dit aantal snel gigantisch groot wordt, schaalt formeel redeneren niet al te best.

De gebruikte programmeertaal heeft een grote invloed op de complexiteit van formeel redeneren. Hoe wiskundiger de programmeertaal, hoe makkelijker redeneren wordt. In het bijzonder geschikt zijn functionele programmeertalen, waarin het gebruik van variabelen niet is toegestaan. Dit maakt het mogelijk om de betekenis van een stuk code te bepalen onafhankelijk van de context waarin het geplaatst is, een eigenschap die formeel redeneren veel makkelijker maakt.

Dit proefschrift onderzoekt de toepasbaarheid van formeel redeneren in de context van de functionele programmeertaal *Clean*. Voor dit onderzoek is de bewijs assistent *Sparkle* ontwikkeld, een computerprogramma dat ondersteuning biedt voor het bouwen van formele bewijzen. *Sparkle* is speciaal bedoeld om door *Clean* programmeurs te worden gebruikt, en is zelf ook volledig in *Clean* geschreven. Er is op dit moment een *Haskell* front-end voor *Clean* in ontwikkeling; hiermee zal *Sparkle* ook kunnen worden gebruikt om te redeneren over *Haskell* programma’s.
De eerste drie hoofdstukken van dit proefschrift geven een algemene beschrijving van Sparkle. Hoofdstuk 2 gaat over CleanProverSystem, het eerste prototype en de voorganger van Sparkle. Hoofdstuk 3 richt zich voornamelijk op de gespecialiseerde kant van Sparkle, dat in de IDE is geïntegreerd en redeneerstappen aanbiedt die toegespitst zijn op Clean. Hoofdstuk 4 geeft een uitgebreide handleiding voor het gebruik van Sparkle in de praktijk.

Een belangrijke eigenschap van zowel Clean als Haskell is luie evaluatie, wat betekent dat berekeningen pas worden uitgevoerd als hun resultaat echt nodig is. Luie evaluatie maakt het gebruik van oneindige structuren mogelijk, maar zorgt er ook voor dat de semantic expliciet rekening moet houden met de ongedefinieerde waarde ‘⊥’. Dit is goed merkbaar tijdens het redeneren, waarin voorwaarden van de vorm ¬(E = ⊥) vaak zullen opduiken. Daarnaast is het in Clean, en in mindere mate ook in Haskell, mogelijk om striktheids annotaties in programma’s op te nemen, wat het gedrag van ‘⊥’ waarden verder beïnvloedt.

Rekening houden met gedefinieerdheid is een extra belasting die in informele bewijzen vaak wordt weggelaten. Wij vinden het echter een belangrijk onderdeel van redeneren over functionele programma’s. Er is in Sparkle daarom uitgebreide ondersteuning opgenomen voor het omgaan met gedefinieerdheid; dit is bijvoorbeeld goed te zien aan het gedrag van de redeneerstappen Reduce, Induction, Cases and Definedness. Door deze ondersteuning wordt het redeneren met gedefinieerdheid een stuk aangenaamer.

In dit proefschrift wordt gedefinieerdheid behandeld in de hoofdstukken 5 en 6. Hoofdstuk 5 behandelt de praktische gevolgen in de vorm van de speciale redeneerondersteuning in Sparkle. Hoofdstuk 6 behandelt de theoretische gevolgen door ‘⊥’ expliciet op te nemen in de standaard semantic van Launchbury[Lau93].

De semantic achter Sparkle is een uitbreiding van de standaard semantic van Clean. Voor evaluatie is een eigen één-staps term-graph reductie systeem gedefinieerd, dat confluent is en zich equivalent aan Launchbury’s systeem[Lau93] gedraagt. Bovenop dit reductie systeem is een eersteorde propositie logica gedefinieerd, waarin gelijkheid op expressies wordt bepaald door te kijken naar observeerbaar programmagedrag. Deze gelijkheid is referentieel transparant en kan omgaan met ongedefinieerde en oneindige reducties.

In dit proefschrift wordt semantic behandeld in de hoofdstukken 7 en 8. In hoofdstuk 7 wordt het reductie systeem beschreven, en in hoofdstuk 8 de gelijkheid op expressies.

Sparkle heeft het redeneren over Clean programma’s veel makkelijker gemaakt: het biedt speciale redeneerstappen die intuitief zijn toe te passen, het is in de IDE ingebouwd zodat het makkelijk is om met redeneren te beginnen, en het heeft een hint mechanisme dat het maken van kleine bewijzen kan automatiseren. Eigenschappen zoals die uit ‘Introduction to Functional Programming’[Bir98] kunnen in Sparkle moeiteloos worden bewezen. Voor grote bewijzen bestaan er echter nog steeds schaalbaarheidsproblemen.

Sparkle is al in verschillende onderzoeksprojecten gebruikt. Er zijn twee lokale uitbreidingen van Sparkle gebouwd: voor de ondersteuning van type classes (zie hoofdstuk 9 van dit proefschrift) en voor de ondersteuning van generieke inductie (zie [Lv04]). Ook is Sparkle ingezet om de correctheid te
bewijzen van een semantisch model van de GEC and iData bibliotheken (zie hoofdstuk 10 van dit proefschrift). Aan deze projecten heb ik zelf meegewerkt.

Daarnaast is Sparkle ook door anderen gebruikt. In Boedapest is er een lokale versie met de naam Sparkle-T ([THK05, THK06]) gebouwd waarmee het mogelijk is om temporele redeneringen te maken. Ook is Sparkle toegepast voor het bewijzen van I/O-programma’s, zowel in Dublin ([DBv04, DBv05]) als in Boedapest ([TKH08]).
Curriculum Vitae

1976   Born on May 24, Velp, the Netherlands;

1988-1994  Christelijk Lyceum Arnhem (Highschool),
           level: VWO, Gymnasium;

1994-1998  Radboud University Nijmegen,
           master’s computer science,
           graduated Cum Laude;

1998-2003  Junior researcher, Radboud University Nijmegen;

2003-2005  Main researcher on project ‘Quality Analysis of Analysis Models’,
           issued by Force Vision (Royal Netherlands Navy);

2005-2008  Thesis completion;
Bibliography


Titles in the IPA Dissertation Series since 2002

M.C. van Wezel. *Neural Networks for Intelligent Data Analysis: theoretical and experimental aspects*. Faculty of Mathematics and Natural Sciences, UL. 2002-01

V. Bos and J.J.T. Kleijn. *Formal Specification and Analysis of Industrial Systems*. Faculty of Mathematics and Computer Science and Faculty of Mechanical Engineering, TU/e. 2002-02


S.P. Luttik. *Choice Quantification in Process Algebra*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2002-04


N. van Vugt. *Models of Molecular Computing*. Faculty of Mathematics and Natural Sciences, UL. 2002-07

A. Fehnker. *Citius, Vilius, Melius: Guiding and Cost-Optimality in Model Checking of Timed and Hybrid Systems*. Faculty of Science, Mathematics and Computer Science, KUN. 2002-08

R. van Stee. *On-line Scheduling and Bin Packing*. Faculty of Mathematics and Natural Sciences, UL. 2002-09


J.I. van Hemert. *Applying Evolutionary Computation to Constraint Satisfaction and Data Mining*. Faculty of Mathematics and Natural Sciences, UL. 2002-14


Y.S. Usenko. *Linearization in μCRL*. Faculty of Mathematics and Computer Science, TU/e. 2002-16

J.J.D. Aerts. *Random Redundant Storage for Video on Demand*. Faculty of Mathematics and Computer Science, TU/e. 2003-01

M. de Jonge. *To Reuse or To Be Reused: Techniques for component composition and construction*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2003-02

J.M.W. Visser. *Generic Traversal over Typed Source Code Representations*. Faculty of Natural Sciences,
S.M. Bohte. *Spiking Neural Networks.* Faculty of Mathematics and Natural Sciences, UL. 2003-04


S.V. Nedea. *Analysis and Simulations of Catalytic Reactions.* Faculty of Mathematics and Computer Science, TU/e. 2003-06


H.P. Benz. *Casual Multimedia Process Annotation – CoMPAs.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2003-08


M.H. ter Beek. *Team Automata – A Formal Approach to the Modeling of Collaboration Between System Components.* Faculty of Mathematics and Natural Sciences, UL. 2003-10


G.I. Jojgov. *Incomplete Proofs and Terms and Their Use in Interactive Theorem Proving.* Faculty of Mathematics and Computer Science, TU/e. 2004-02

P. Frisco. *Theory of Molecular Computing – Splicing and Membrane systems.* Faculty of Mathematics and Natural Sciences, UL. 2004-03

S. Maneth. *Models of Tree Translation.* Faculty of Mathematics and Natural Sciences, UL. 2004-04

Y. Qian. *Data Synchronization and Browsing for Home Environments.* Faculty of Mathematics and Computer Science and Faculty of Industrial Design, TU/e. 2004-05


L. Cruz-Filipe. *Constructive Real Analysis: a Type-Theoretical Formalization and Applications.* Faculty of Science, Mathematics and Computer Science, KUN. 2004-07

E.H. Gerding. *Autonomous Agents in Bargaining Games: An Evolutionary Investigation of Fundamentals, Strategies, and Business Applications.* Faculty of Technology Management, TU/e. 2004-08

N. Goga. *Control and Selection Techniques for the Automated Testing of Reactive Systems.* Faculty of Mathematics and Computer Science, TU/e. 2004-09


J. Pang. Formal Verification of Distributed Systems. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2004-14

F. Alkemade. Evolutionary Agent-Based Economics. Faculty of Technology Management, TU/e. 2004-15

E.O. Dijk. Indoor Ultrasonic Position Estimation Using a Single Base Station. Faculty of Mathematics and Computer Science, TU/e. 2004-16

S.M. Orzan. On Distributed Verification and Verified Distribution. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2004-17


P.J.L. Cuijpers. Hybrid Process Algebra. Faculty of Mathematics and Computer Science, TU/e. 2004-20

N.J.M. van den Nieuwelaar. Supervisory Machine Control by Predictive-Reactive Scheduling. Faculty of Mechanical Engineering, TU/e. 2004-21

E. Ábrahám. An Assertional Proof System for Multithreaded Java - Theory and Tool Support-. Faculty of Mathematics and Natural Sciences, UL. 2005-01

R. Ruimerman. Modeling and Remodeling in Bone Tissue. Faculty of Biomedical Engineering, TU/e. 2005-02

C.N. Chong. Experiments in Rights Control - Expression and Enforcement. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2005-03

H. Gao. Design and Verification of Lock-free Parallel Algorithms. Faculty of Mathematics and Computing Sciences, RUG. 2005-04

H.M.A. van Beek. Specification and Analysis of Internet Applications. Faculty of Mathematics and Computer Science, TU/e. 2005-05


I. Kurtev. Adaptability of Model Transformations. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2005-08
T. Wolle. Computational Aspects of Treewidth - Lower Bounds and Network Reliability. Faculty of Science, UU. 2005-09

O. Tveretina. Decision Procedures for Equality Logic with Uninterpreted Functions. Faculty of Mathematics and Computer Science, TU/e. 2005-10

A.M.L. Liekens. Evolution of Finite Populations in Dynamic Environments. Faculty of Biomedical Engineering, TU/e. 2005-11

J. Eggermont. Data Mining using Genetic Programming: Classification and Symbolic Regression. Faculty of Mathematics and Natural Sciences, UL. 2005-12

B.J. Heeren. Top Quality Type Error Messages. Faculty of Science, UU. 2005-13

G.F. Frehse. Compositional Verification of Hybrid Systems using Simulation Relations. Faculty of Science, Mathematics and Computer Science, RU. 2005-14


T. Gelsema. Effective Models for the Structure of pi-Calculus Processes with Replication. Faculty of Mathematics and Natural Sciences, UL. 2005-17

P. Zoeteweij. Composing Constraint Solvers. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2005-18


M. Valero Espada. Modal Abstraction and Replication of Processes with Data. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2005-20

A. Dijkstra. Stepping through Haskell. Faculty of Science, UU. 2005-21


E. Dolstra. The Purely Functional Software Deployment Model. Faculty of Science, UU. 2006-01


P.R.A. Verbaan. The Computational Complexity of Evolving Systems. Faculty of Science, UU. 2006-03


M. Kyas. Verifying OCL Specifications of UML Models: Tool Support and Compositionality. Faculty of Mathematics and Natural Sciences, UL. 2006-05

M. Hendriks. Model Checking Timed Automata - Techniques and
Applications. Faculty of Science, Mathematics and Computer Science, RU. 2006-06

J. Ketema. Böhm-Like Trees for Rewriting. Faculty of Sciences, VUA. 2006-07

C.-B. Breunesse. On JML: topics in tool-assisted verification of JML programs. Faculty of Science, Mathematics and Computer Science, RU. 2006-08

B. Markvoort. Towards Hybrid Molecular Simulations. Faculty of Biomedical Engineering, TU/e. 2006-09

S.G.R. Nijssen. Mining Structured Data. Faculty of Mathematics and Natural Sciences, UL. 2006-10

G. Russello. Separation and Adaptation of Concerns in a Shared Data Space. Faculty of Mathematics and Computer Science, TU/e. 2006-11

L. Cheung. Reconciling Non-deterministic and Probabilistic Choices. Faculty of Science, Mathematics and Computer Science, RU. 2006-12

B. Badban. Verification techniques for Extensions of Equality Logic. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2006-13

A.J. Mooij. Constructive formal methods and protocol standardization. Faculty of Mathematics and Computer Science, TU/e. 2006-14


M.E. Warnier. Language Based Security for Java and JML. Faculty of Science, Mathematics and Computer Science, RU. 2006-16

V. Sundramoorthy. At Home In Service Discovery. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2006-17

B. Gebremichael. Expressivity of Timed Automata Models. Faculty of Science, Mathematics and Computer Science, RU. 2006-18

L.C.M. van Gool. Formalising Interface Specifications. Faculty of Mathematics and Computer Science, TU/e. 2006-19


J.V. Guillem Scholten. Mobile Channels for Exogenous Coordination of Distributed Systems: Semantics, Implementation and Composition. Faculty of Mathematics and Natural Sciences, UL. 2006-21

H.A. de Jong. Flexible Heterogeneous Software Systems. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2007-01

N.K. Kavaldjiev. A run-time reconfigurable Network-on-Chip for streaming DSP applications. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2007-02

M. van Veelen. Considerations on Modeling for Early Detection of Abnormalities in Locally Autonomous Distributed Systems. Faculty of Mathematics and Computing Sciences, RUG. 2007-03

Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2007-04


M.W.A. Streppel. Multifunctional Geometric Data Structures. Faculty of Mathematics and Computer Science, TU/e. 2007-07

N. Trčka. Silent Steps in Transition Systems and Markov Chains. Faculty of Mathematics and Computer Science, TU/e. 2007-08

R. Brinkman. Searching in encrypted data. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2007-09

A. van Weelden. Putting types to good use. Faculty of Science, Mathematics and Computer Science, RU. 2007-10


R. Boumen. Integration and Test plans for Complex Manufacturing Systems. Faculty of Mechanical Engineering, TU/e. 2007-12


C.F.J. Lange. Assessing and Improving the Quality of Modeling: A Series of Empirical Studies about the UML. Faculty of Mathematics and Computer Science, TU/e. 2007-14

T. van der Storm. Component-based Configuration, Integration and Delivery. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2007-15

B.S. Graaf. Model-Driven Evolution of Software Architectures. Faculty of Electrical Engineering, Mathematics, and Computer Science, TU/e. 2007-16

A.H.J. Mathijssen. Logical Calculi for Reasoning with Binding. Faculty of Mathematics and Computer Science, TU/e. 2007-17


W. Pieters. La Volonté Machinale: Understanding the Electronic Voting Controversy. Faculty of Science, Mathematics and Computer Science, RU. 2008-01

A.L. de Groot. Practical Automation Proofs in PVS. Faculty of Science, Mathematics and Computer Science, RU. 2008-02

M. Bruntink. Renovation of Idiomatic Crosscutting Concerns in Embedded Systems. Faculty of Electrical Engineering, Mathematics, and Computer Science, TU/e. 2008-03

A.M. Marin. An Integrated System to Manage Crosscutting Concerns in Source Code. Faculty of Electrical
Engineering, Mathematics, and Computer Science, TUD. 2008-04

N.C.W.M. Braspenning. Model-based Integration and Testing of High-tech Multi-disciplinary Systems. Faculty of Mechanical Engineering, TU/e. 2008-05

M. Bravenboer. Exercises in Free Syntax: Syntax Definition, Parsing, and Assimilation of Language Conglomerates. Faculty of Science, UU. 2008-06


I.S.M. de Jong. Integration and Test Strategies for Complex Manufacturing Machines. Faculty of Mechanical Engineering, TU/e. 2008-08

I. Hasuo. Tracing Anonymity with Coalgebras. Faculty of Science, Mathematics and Computer Science, RU. 2008-09

L.G.W.A. Cleophas. Tree Algorithms: Two Taxonomies and a Toolkit. Faculty of Mathematics and Computer Science, TU/e. 2008-10


M. Farshi. A Theoretical and Experimental Study of Geometric Networks. Faculty of Mathematics and Computer Science, TU/e. 2008-12


E.M. Bortnik. Formal Methods in Support of SMC Design. Faculty of Mechanical Engineering, TU/e. 2008-16


M. van der Horst. Scalable Block Processing Algorithms. Faculty of Mathematics and Computer Science, TU/e. 2008-18


E. Mumford. Drawing Graphs for Cartographic Applications. Faculty of Mathematics and Computer Science, TU/e. 2008-21

E.H. de Graaf. Mining Semi-structured Data, Theoretical and Ex-
perimental Aspects of Pattern Evaluation. Faculty of Mathematics and Natural Sciences, UL. 2008-22

R. Brijder. Models of Natural Computation: Gene Assembly and Membrane Systems. Faculty of Mathematics and Natural Sciences, UL. 2008-23

A. Koprowski. Termination of Rewriting and Its Certification. Faculty of Mathematics and Computer Science, TU/e. 2008-24


H. Kastenberg. Graph-Based Software Specification and Verification. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-27


M. de Mol. Reasoning about Functional Programs: Sparkle, a proof assistant for Clean. Faculty of Science, Mathematics and Computer Science, RU. 2009-02