DISS. ETH NO. 20516

Formalizing the Logic of Event-B

Partial Functions, Definitional Extensions, and Automated Theorem Proving

A dissertation submitted to
ETH ZURICH
for the degree of
Doctor of Sciences

presented by

Matthias Schmalz
Diplom der Informatik, Universität zu Lübeck
born on October 5, 1981
citizen of Germany

accepted on the recommendation of

Prof. Dr. David Basin
Prof. Dr. Cliff Jones
Prof. Dr. Peter Müller
Prof. Dr. Tobias Nipkow

2012

For Yang and Maximilian

Abstract

The basic motivation behind this thesis is to develop methods and tools for building highly reliable computerized systems. The work builds on the formal method *Event-B*, the corresponding development environment *Rodin*, and the theorem prover *Isabelle/HOL*. Event-B provides a methodology for developing models of complex systems. A major strength of Event-B is its carefully designed refinement calculus, which helps to break big problems down to manageable pieces. A major weakness of Event-B and Rodin is that the underlying logic is poorly understood; this has led to unsound proofs and impedes enhancements of the logic. Another problem is that it is difficult to improve the performance of Rodin's theorem prover on domains to which it has not been applied before. Isabelle/HOL is a theorem prover with strong theoretical foundations and powerful facilities to adapt its proof methods to new domains. Like Rodin, Isabelle/HOL can be used to develop models of systems, but it offers less user guidance and lacks several useful features of Rodin. The contribution of this thesis is to develop a comprehensive theoretical foundation of Event-B's logic and to bring Rodin together with Isabelle/HOL, creating a tool that has the advantages of both worlds.

The specification of Event-B's logic covers abstract syntax, semantics, proofs, and methods for definitional theory extensions. Since Event-B's logic closely resembles higher-order logic, I define its semantics by an embedding into higher-order logic with the benefit that several meta-results on higher-order logic can be straightforwardly transferred to Event-B. Event-B explicitly supports partial functions; as is often the case for logics of partial functions, some design decisions are difficult to make and to explain. I therefore carefully analyze the impact of the non-trivial design decisions underlying Event-B's logic; my analysis provides useful information for planning future changes of Event-B and for developing other logics of partial functions. By integrating Isabelle/HOL as a theorem prover into Rodin, Rodin obtains a proof tactic that improves over existing tactics in terms of soundness, adaptability to new domains, and sometimes even performance.

Although this research has been driven by the aim to improve Event-B and Rodin, it has led to results of a more general interest. One of them is *directed rewriting*, a term rewriting technique for logics of partial functions that has been implemented in several theorem provers, but lacks a widely known theoretical justification. I show under which conditions directed rewriting is safe and sound, and I demonstrate that directed rewriting significantly reduces the need for solving preconditions of rewrite rules during proofs.

Another result of general interest concerns the embedding of logics of partial functions into classical logics. Such embeddings typically suffer an exponential blowup, if connectives and quantifiers are interpreted in Kleene semantics. I propose an embedding that applies to a broader class of logics of partial functions than existing embeddings. With my parametric complexity analysis and empirical evaluation I refute the plausible and widely accepted belief that the exponential overhead of such embeddings is unacceptably high for practical applications.

Zusammenfassung

Gegenstand dieser Dissertation sind Methoden zur Entwicklung qualitativ hochwertiger computerbasierter Systeme. Ausgangspunkte sind die formale Methode *Event-B*, die zugehörige Entwicklungsumgebung *Rodin* und der generische Theorembeweiser *Isabelle* instanziiert mit Logik höherer Stufe (Isabelle/HOL). Event-B ermöglicht es, verschiedenste Arten von komplexen Systemen auf intuitive Art und Weise zu modellieren und deren Korrektheit zu beweisen. Der Erfolg von Event-B wird jedoch dadurch relativiert, dass sich mithilfe verschiedener Versionen von Rodin die Korrektheit fehlerhafter Systeme beweisen lässt. Grund dafür waren Programmierfehler in Rodins Theorembeweiser, die häufig auf ein mangelhaftes Verständnis der zugrunde liegenden Logik zurückzuführen sind. Dieses mangelhafte Verständnis der Logik behindert auch die Weiterentwicklung von Event-B. Eine weitere Schwäche von Rodins Theorembeweiser besteht darin, dass seine Suchstrategien sich nur mit sehr grossem Aufwand verändern lassen. Der Theorembeweiser Isabelle/HOL zeichnet sich dagegen dadurch aus, dass Fehler in Beweisen äusserst unwahrscheinlich sind und sich dessen Beweisstrategien mit geringem Aufwand für spezifische Anwendungsfälle optimieren lassen. Isabelle/HOL kann zwar grundsätzlich auch zum Modellieren von Systemen verwendet werden, ist in dieser Hinsicht jedoch deutlich weniger intuitiv zu bedienen als Rodin. Der erste Beitrag dieser Dissertation ist eine systematische Beschreibung der Logik von Event-B, die gängigen wissenschaftlichen Qualitätskriterien genügt. Der zweite Beitrag besteht in der Integration von Isabelle/HOL in Rodin, so dass eine Entwicklungsumgebung entsteht, die die Vorteile von Event-B und Isabelle/HOL vereint.

Die Beschreibung der Logik von Event-B beinhaltet deren Syntax, Semantik, Beweiskalküle sowie Methoden zur Erweiterung von Theorien mithilfe von Definitionen. Da die Logik von Event-B viele Ähnlichkeiten mit Logik höherer Stufe hat, definiere ich deren Semantik mithilfe einer Einbettung in Logik höherer Stufe. Dies hat den Vorteil, dass verschiedenste Erkenntnisse über Logik höherer Stufe sich einfach auf Event-B übertragen lassen. Die Logik von Event-B bietet verschiedene Mechanismen, die die Modellierung von partiellen Funktionen erleichtern, den Aufbau der Logik selbst jedoch verkomplizieren. Ich analysiere, wie diese Mechanismen zusammenwirken; dies erleichtert es, die Auswirkungen von zukünftigen Veränderungen der Logik vorherzusagen. Durch die Anbindung von Isabelle/HOL erhält Rodin eine Beweisstrategie, die bestehenden Beweisstrategien in Bezug auf Korrektheit, Anpassbarkeit und manchmal auch Performanz überlegen ist.

Auch wenn das primäre Ziel dieses Forschungsprojekts die Verbesserung von Event-B und Rodin ist, haben manche der behandelten Beweistechniken Anwendungen ausserhalb von Event-B. Eine dieser Techniken ist *Directed Rewriting*, eine Termersetzungsstrategie für Logiken mit partiellen Funktionen. Obwohl diese Strategie bereits in mehreren Theorembeweisern implementiert wurde, gab es bislang keine allgemein bekannte Erklärung für deren Korrektheit. Ich erkläre, unter welchen

Bedingungen directed Rewriting korrekt ist, und ich zeige empirisch, dass directed Rewriting Termersetzungsbeweise in praktisch relevanten Szenarien erheblich vereinfacht.

Ein weiteres Ergebnis mit Anwendungen ausserhalb von Event-B betrifft die Einbettung von Logiken mit partiellen Funktionen in klassische Logiken. Dieses Problem ist insbesondere dann schwierig, wenn boolesche Junktoren und Quantoren gemäss Kleene Semantik interpretiert werden. Ich definiere eine Einbettungen, die eine grössere Klasse von Logiken mit partiellen Funktionen abdeckt als bereits bekannte Einbettungen. Weiterhin untersuche ich die Effizienz meiner Einbettung sowohl mit einer parametrischen Komplexitätsanalyse als auch mithilfe von Experimenten; mit meiner Analyse widerlege ich die weitverbreitete Annahme, dass Einbettungen von Kleene Logiken in klassische Logiken für praktische Anwendungen zu ineffizient sind.

Acknowledgements

I would like to thank my wife Yang for her outstanding patience, loyalty, and support. If there is one thing that is harder than writing a PhD thesis, then it is to take care of a little child while the husband is writing a PhD thesis.

I am very grateful to my supervisor David Basin for continuously reminding me how outstanding research is done. I hope that I have understood at least half of his lessons. I would also like to thank him for granting the freedom to pursue my own ideas.

I am deeply indebted to numerous persons for their advice, inspiring discussions, moral support, patience, loyalty, and other kinds of help:

Jean-Raymond Abrial, Nicolas Beauger, the administrators of ETH's high-performance computing cluster Brutus, Michael Butler, Cas Cremers, Leo Freitas, Andreas Fürst, Barbara Geiser, Rainer Gmehlich, Katrin Grau, Gudmund Grov, Stefan Hallerstede, Matus Harvan, Thai Son Hoang, Cliff Jones, Felix Klaedtke, Michael Leuschel, Felix Lösch, Issam Maamria, Ognjen Maric, Srdjan Marinovic, Farhad Mehta, Simon Meier, Peter Müller, Thomas Muller, Tobias Nipkow, Carine Pascal, Lawrence Paulson, Jann Röder, Mark Saaltink, Patrick Schaller, Benedikt Schmidt, Christoph Sprenger, Laurent Voisin, Makarius Wenzel, Burkhart Wolff.

Thank you very much for your support!

This research has been funded by the European Union's FP7 project Deploy (`www.deploy-project.eu`) and by ETH Zurich's Institute of Information Security. I am very grateful for this financial support; it allowed my family and me to have a comfortable life during the last years time.

Contents

1. Introduction **15**
- 1.1. Event-B . . . 16
  - 1.1.1. Strengths . . . 17
  - 1.1.2. Limitations . . . 18
- 1.2. Other Modeling and Verification Environments . . . 20
- 1.3. Problem Statement . . . 21
- 1.4. Approach to a Solution . . . 21
- 1.5. Overview and Contributions . . . 23

2. Event-B's Logic **27**
- 2.1. Abstract Syntax . . . 27
  - 2.1.1. Types and Terms . . . 28
  - 2.1.2. An Example Signature . . . 30
- 2.2. Semantics . . . 31
  - 2.2.1. Isabelle/HOL . . . 31
  - 2.2.2. Preliminaries – the Theory EB_Prelims . . . 32
  - 2.2.3. Denotations of Types and Terms . . . 33
  - 2.2.4. Extending Structures . . . 34
  - 2.2.5. Semantic Notions . . . 34
  - 2.2.6. An Example Structure . . . 35
  - 2.2.7. Substitutions . . . 36
    - 2.2.7.1. Type Substitutions . . . 36
    - 2.2.7.2. Operator Substitutions . . . 38
    - 2.2.7.3. Ordinary Substitutions . . . 40
- 2.3. Proofs . . . 40
  - 2.3.1. Sequents . . . 41
  - 2.3.2. Rules of Proofs . . . 42
    - 2.3.2.1. Inference Rules . . . 42
    - 2.3.2.2. Symmetric Rewrite Rules . . . 44
    - 2.3.2.3. Directed Rewrite Rules . . . 46
    - 2.3.2.4. Expressiveness and Limitations . . . 48
  - 2.3.3. Theories . . . 50
- 2.4. Comparison to Rodin's Implementation . . . 51

3. Event-B's Theories **55**
3.1. The Theory Core . . . 57
3.1.1. The Theory Min . . . 57
3.1.2. The Theory Log_0 . . . 58
3.1.3. The Theory Log . . . 59
3.1.4. The Theory Set_0 . . . 62
3.1.5. The Theory Choice . . . 63
3.1.6. The Theory Int_0 . . . 64
3.1.7. The Theory Prod . . . 67
3.1.8. The Theory Core . . . 67
3.2. Definitional Extensions . . . 67
3.2.1. New Rules . . . 69
3.2.2. New Operators . . . 70
3.2.2.1. Operator Definitions . . . 70
3.2.2.2. Strict Operator Definitions . . . 72
3.2.3. New Binders . . . 74
3.2.4. Operator Variables in a Monotonic Setting . . . 78
3.3. Derived Theories . . . 82
3.3.1. The Theory Bool . . . 82
3.3.2. The Theory Set . . . 83
3.3.3. The Theory Rel . . . 84
3.3.4. The Theory Int . . . 86
3.3.5. The Theory EventB . . . 88
3.4. Comparison to Other Expositions on Event-B's Logic . . . 88

4. Impact of Design Decisions **91**
4.1. Directed Rewriting . . . 91
4.2. Dependencies Between Design Decisions . . . 94
4.3. Kleene versus McCarthy Semantics . . . 97
4.4. Alternative Approaches to Partial Functions . . . 99

5. Automated Theorem Proving **105**
5.1. Preliminaries . . . 106
5.1.1. Benchmarks . . . 106
5.1.2. Hardware and Software . . . 107
5.2. Unlifting . . . 108
5.2.1. Trivial Algorithm . . . 109
5.2.2. Efficient Algorithm . . . 111
5.2.3. Operator Variables . . . 112
5.2.4. Complexity Analysis . . . 113
5.2.5. Implementation . . . 118
5.2.6. Empirical Evaluation . . . 121

5.3. Theorem Proving in Isabelle/HOL 126
5.3.1. Abrial Benchmark 126
5.3.1.1. Using Predefined Proof Methods 126
5.3.1.2. Design of Axe 134
5.3.1.3. Performance of Axe 136
5.3.1.4. Impact of Design Decisions 139
5.3.1.5. Comparison to Rodin 144
5.3.2. BepiColombo Benchmark 145
5.3.2.1. Further Improvements of Axe 145
5.3.2.2. Comparison to Rodin 154
5.3.3. Limitations 155
5.3.4. Conclusions 155

6. Conclusions **157**
6.1. Summary of Contributions 157
6.2. Future Work 159
6.3. Personal Remarks 160

List of Figures **163**

Bibliography **165**

A. Statistics on Rules Implemented in Rodin **173**
A.1. Conditional Rewrite Rules 173
A.2. Truly Directed Rewrite Rules in Rodin 174
A.3. Conditional Rewrite Rules with Dispensable Conditions 176

B. Complexity of Unlifting **177**
B.1. Definitions 177
B.2. Automated Tool Setup 180
B.3. Main Results 182

C. Configurations of Rodin's Auto-Tactic **187**

1. Introduction

Computerized systems play an increasingly important role in everyday life. Some systems perform safety critical tasks, for example systems that control plains, trains, cars, space crafts, or nuclear power plants. The consequences of failure in such systems may be catastrophic. As a result, there is a strong need for methods that make catastrophic failures so unlikely that the remaining risk is acceptable to the society.

There are several ways of improving the quality of a system, ranging from appropriate business processes (enforcing, e.g., separation of duty) over guidelines for phrasing system requirements to methods for implementing and verifying systems. In the context of this thesis, it is assumed that a specification has been formulated in an unambiguous manner and it remains to develop a system that is correct with respect to this specification. This is a well-known problem in the field of *formal methods* [see Hinchey et al., 2010]. The postulate that specifications are "unambiguous" rules out requirements like "The car should drive smoothly" because the term "smoothly" is interpreted differently by different drivers.

Whether a system is correct with respect to its specification can in general not be computed, because the underlying problem is undecidable. A noteworthy exception is the case of systems that have finitely many states; in this case, correctness can sometimes be checked with a model checker [see Baier and Katoen, 2008, Clark et al., 2000]. Model checking has the pleasing advantage that it requires little user interaction; but it is inapplicable to a significant class of systems, because they have too many states or the system or specification is too hard to express in the languages supported by model checkers. For such systems, model checkers may be useful in showing the presence of bugs, but they are unable to show their absence.

Although the correctness of a system can in general not be computed, it can be proved within a mathematical formalism (provided the system is indeed correct). Methods for developing systems and reasoning about their correctness date back to the late sixties of the last century [Dijkstra, 1976, Hoare, 1969]. A considerable effort has been made to develop corresponding tools [e.g. Abrial et al., 2010, Arthan, 1996, Barnett et al., 2011, Böhme et al., 2010, ClearSy, Cok and Kiniry, 2004, Flanagan et al., 2002, Jones et al., 1991, Kaufmann et al., 2000, Nipkow et al., 2002, Ouimet and Lundqvist, 2007, Owre et al., 1992, Saaltink, 1997, Toyn and McDermid, 1995]. These tools allow users to describe systems and specifications in a formal language and to carry out the corresponding correctness proofs. The duty of the tool is to help the user structure the descriptions of system and specification, and to check whether the correctness proof is correct; most tools also support the user in finding a correctness proof, if the system is correct, or pointing out bugs, if the system is

incorrect.

In practice, many problems remain with this approach. The user may fail to faithfully translate the system or the intended specification into the language supported by the formal method. Or the user may misunderstand the "real" requirements on the system. Thus, even if the system is correct with respect to the given specification, it may fail to behave in the intended way.

The theorem prover or some other critical component of the tool may be incorrect. Or the language used to express system and specification may suffer fundamental problems, which may in the extreme case have the consequence that every system is considered correct. The user may fail to prove correctness because the effort of communicating a proof to the theorem prover is too high. Or limitations in the tool may make it impossible to prove correctness of certain correct systems.

Although the progress made during the last decade is impressive, development of correct systems remains a demanding task, in particular for systems at industrial scale. Developing a formal engineering method and corresponding tool does not only involve a serious amount of implementation work, but also the development of theoretical foundations and the solution of other research problems. Roughly speaking, the goal of this thesis is to make a substantial progress towards a solution of this challenge.

1.1. Event-B

The starting point of this research is the Event-B method [Abrial, 2010] with the Rodin platform [Abrial et al., 2010], the corresponding development environment. The choice of a starting point is of course somewhat arbitrary; to explain my choice, I summarize Event-B's main features, strengths, and limitations.

Event-B is a method for modeling discrete transition systems as a collection of *events*, which are a restricted version of guarded commands [Dijkstra, 1975]. In the simplest case, the user specifies one transition system, called *machine*, and an *invariant claim*, i.e., the claim that a given property, the *invariant*, holds in every reachable state. Rodin generates several *proof obligations*, i.e., logical statements entailing correctness of the system with respect to its specification. If the user proves all proof obligations, it is for example known that the invariant holds in every reachable state.

A typical Event-B model consists of *several* transition systems organized in a chain. The first machine of the chain is the *abstract* machine, and each subsequent machine is claimed to *refine* its predecessor. The abstract machine usually describes a small but crucial part of the specification. Subsequent machines introduce more details: either details on the specification or details on the system explaining how exactly the system is executed. So every machine in the chain constitutes a limited view on the system and specification, with increasing amount of detail. Abrial [2010] describes with several examples how this process works in practice.

Users have to prove several proof obligations entailing that the various refinement

claims are true. In a nutshell, a machine m_2 *refines* a machine m_1 if and only if every execution of m_2 is an execution of m_1. In practice, the notion of refinement is more complicated, but the details are irrelevant at this point.

The overall correctness proof is structured into refinement and invariant claims. Refinement has the effect that invariants (and other kinds of properties) of one machine continue to hold in subsequent machines. Refinement therefore supports incremental development and separation of concerns.

1.1.1. Strengths

I view Event-B and Rodin as promising for developing correct systems for the following reasons. Refinement facilitates writing systems and specifications, because it relieves the user of the burden of coping with all details at once. By organizing the overall correctness proof into proof obligations, the proof task is automatically divided into manageable pieces that depend on small parts of the model. Since a system in Event-B is a collection of events, Event-B can be naturally applied to problems from various domains, such as the development of concurrent algorithms, network protocols, and digital circuits [Abrial, 2010]. Event-B is the immediate successor of classical B [Abrial, 2005], which has been successfully applied to large scale industrial projects [Badeau and Amelot, 2005, Behm et al., 1999]. This demonstrates the maturity of the underlying methodology.

As partial functions (such as array access, division, and file manipulation) are omnipresent in computer systems, Event-B takes partial functions seriously. Informally, if the user applies a function to an argument that does not belong to the function's domain, there will be a proof obligation that cannot be proved.

Several people have contributed useful features to Rodin. *UML-B* [Snook and Butler, 2008] allows users to define Event-B models with UML-like diagrams (having a formal semantics). It facilitates structuring large models, because UML diagrams are often easier to grasp than long lists of events.

The ProB tool [Leuschel et al., 2011] has been integrated into Rodin as an animator and model checker; it helps users in gaining trust that the formalization of the system and specification coincides with their intentions. ProB can discover simple errors with little effort: either by a manual or automated search of the state space (i.e., by animation or model checking), or by applying constraint solving techniques to proof obligations [Hallerstede and Leuschel, 2011].

The interaction with Rodin's theorem prover is intuitive and easy to learn. A user performs a proof step by clicking on a hyperlink on a symbol to which the proof step is related. Quantifiers are instantiated by entering the witness in a textfield and clicking on the quantifier.

If the model changes, some proof obligations change and the corresponding proofs need to be revised. Rodin implements several measures for minimizing the effort of revising proofs [Mehta, 2007]. For example, Rodin records the hypotheses of a proved proof obligation on which the corresponding proof depends. If the proof obligation changes (because of a change in the model), the proof is converted into a proof of

the changed proof obligation, provided the hypotheses on which the proof depends are still available.

This list of useful features is not complete. In summary, Rodin offers several features that support users in modeling large and complex systems.

1.1.2. Limitations

At the time of starting this research, there have been several problems that Event-B and Rodin did not solve.

Modeling. Unlike in classical B [Abrial, 2005], control structures like sequential compositions, conditionals, and loops need to be modeled by using auxiliary variables that simulate a program counter; this encoding of control flow complicates maintenance of models and proof activities. It is an ongoing challenge to express and reason about real-time, probabilistic, and liveness properties; progress on some of these topics is reported in [Hoang and Abrial, 2011, Yilmaz and Hoang, 2010]. Gmehlich et al. [2011] give a detailed account on the difficulties of applying Event-B and Rodin to an industrial cruise control system; in particular, it is demanding to synchronize an Event-B model with a large number of requirements.

Logical Foundations. I designate the language used to express the atomic elements of models (such as guards, invariants, and proof obligations) as Event-B's *logic*. Although Event-B is used to reason about the correctness of systems, Event-B's logic itself lacks a rigorous specification. It is therefore unclear what it actually means that Rodin's theorem prover is correct. There is of course an unwritten agreement on basic questions, e.g., $2+2=4$ should be provable and $2+2=5$ should not. The main confusion concerns Event-B's approach to partial functions (see Section 3.4).

The lack of a clear semantics has had several negative consequences. Firstly, it raises a credibility problem. Rodin users are told that the use of formal methods only makes sense with a clean specification; but Rodin does not have one.

A more technical problem is that several releases of Rodin are affected by consistency bugs [Schmalz, 2012]. With these Rodin releases, every system can be proved correct. Some of the consistency bugs have been introduced because of a poor understanding of the underlying semantics.

Another problem concerns the construction of proof tactics. A poor understanding of what exactly qualifies a proof step as sound leads to unnecessary restrictions in tactics; I give examples in Section A.3.

To improve a logic it is necessary to change it. Without a clear understanding of the logic, it is hard to foresee the consequences of changes. In practice, this has impeded or prevented useful changes of Event-B's logic – the risk of change was considered too high.

One of the impeded changes is the introduction of *theory extension methods*; these are methods for introducing new types, functions, or proof rules. A naive approach to theory extensions easily leads to inconsistency [see Gordon and Pitts, 1993, p. 221,

for one of the pitfalls]. Without clean theoretical foundations, it is neither possible to formulate the requirements on theory extension methods nor to prove that these requirements are fulfilled.

Theorem Proving. Rodin users complain about the performance of Rodin's automated theorem provers. The situation has been improved by implementing relevance filtering techniques [Röder, 2010], but there remains room for further improvements. Since automated theorem proving is an undecidable problem, it is trivial to observe that the performance can be improved. But it is a relevant problem because users have expressed that they view the current performance as unsatisfactory.

Using common mathematical notation, it is impossible to prove $3^3 = 27$ with Rodin's theorem prover, at least if bugs in the theorem prover are not exploited. The formula $2^2 = 4$ can be proved, but the known proof requires an invocation of the rule that rewrites $\mathsf{card}(\mathbb{P}(R))$ to $2^{\mathsf{card}(R)}$, for finite sets R. So there are "simple" theorems that cannot be proved with Rodin's theorem provers or whose proofs are surprisingly lengthy and complicated. This problem is unavoidable because a generally accepted definition of "simple" theorem with a corresponding efficient proof procedure does not exist.

Other proof assistants [such as those in Gordon and Melham, 1993, Nipkow et al., 2002] cope with this problem by having powerful facilities for configuring existing proof tactics and defining new ones. In contrast, Rodin supports two ways of defining new tactics. The first way is to implement (in Java) a patch of Rodin's theorem prover; the person who writes the patch is responsible that it does not introduce inconsistency. The second way is to introduce new proof rules with Rodin's theory plug-in [Butler and Maamria, 2010, Maamria and Butler, 2010]; but although the theory plug-in is a valuable effort, it still has several restrictions that competing proof assistants do not have: it can be used to introduce new inconsistencies into Rodin's proof calculus [Schmalz, 2012], there are restrictions on how the connectives and quantifiers of predicate calculus may occur in rules, and the before-mentioned problem with exponentiation cannot be solved. Overall, Rodin's facilities for improving its theorem prover are significantly weaker than those of competing proof assistants.

Engineering. Rodin and its plug-ins have suffered from a number of engineering problems. The installation of plug-ins is not always intuitive and sometimes unreasonably slow. Users have experienced that the platform occasionally crashes. In some cases, the quality of the architecture, source code, documentation, or user interaction is not as high as it perhaps should be. For example, some kinds of failures are silently ignored or reported with too little emphasis; the logfile typically contains numerous events, and it is hard to distinguish minor problems from serious ones. Working with large models has demanded a lot of patience from the user; but the situation has recently improved. Not all problems originate from Rodin's source code; some of them are inherited by Eclipse, which is of course not an excuse.

It is important to view these engineering limitations in the right context. Building an integrated development environment is a non-trivial and resource draining task, and the resources for "mere" tool development have always been very limited. With this in mind, Rodin still is a remarkable piece of software.

1.2. Other Modeling and Verification Environments

An alternative tool for modeling and verifying systems is the theorem prover Isabelle/HOL [Nipkow et al., 2002], the instantiation of the generic theorem prover Isabelle [Paulson, 1989] to higher-order logic (HOL). Isabelle/HOL has been applied in prestigious verification projects such as the refinement based development of an operating system kernel [Klein et al., 2010]. Sprenger and Basin [2010] have implemented an Event-B style refinement methodology within Isabelle/HOL and applied it to develop security protocols. By using Isabelle/HOL, most of the limitations of Rodin's theorem prover are avoided. Isabelle/HOL gives users much flexibility on the used methodology: in refinement based approaches, users may freely choose and change the underlying refinement calculus. In contrast, it is much more difficult to change Event-B's refinement calculus because it is hard-coded in Rodin's source code.

On the downside, Isabelle/HOL does not provide features for validating models such as Rodin's UML representation, animation, and model-checking; to validate models, users have to inspect the theory source code. The price of flexibility is that the decomposition of refinement and invariant claims into proof obligations has to be guided by the user. Such a manual decomposition into proof obligations is feasible for small models or with strongly experienced users, but it bears the risk that users are overwhelmed by repetitive trivial tasks. Unlike in Rodin, dependencies of proofs are not cached; whenever a declaration within a theory is changed, all subsequent proofs need to be re-checked.

It is of course possible to overcome these limitations of Isabelle/HOL by building appropriate front-ends or extensions. Examples include the front-ends for Z and UML/OCL by Brucker et al. [2003].

Spec# is a well-known approach for reasoning about software systems [Barnett et al., 2011]. Spec# is an extension of the C# programming language with annotations such as assertions, pre- and postconditions for methods, and invariants for classes. There is also a similar extension of C, supported by the verifying C compiler [Cohen et al., 2009]. The annotations are transformed into proof obligations that are passed to an automated theorem prover, such as Z3 [de Moura and Bjørner, 2008]. Proof obligations of the verifying C compiler can also be interactively proved in Isabelle/HOL [Böhme et al., 2010]. The main advantage and disadvantage of this approach is that the system is expressed in a programming language. It is an advantage because the used programming languages are widely known, and sophisticated translations between system models and programming languages are avoided. It is a disadvantage because it limits the scope of the approach to software and it

inhibits the use of refinement: all details of the system have to be specified at once. It also seems difficult to use Spec# for reasoning about concurrent systems or about systems that are executed within a sophisticated environment.

Overall, there are many methods and tools for modeling and verifying systems. The decision to take Event-B as a starting point of my thesis is of course somewhat arbitrary. The disadvantage of focusing on one method is that results may not generalize to other methods; as a great advantage of this approach, it is easier to see which problems really matter in practice and to develop results that make a real difference in verification projects.

1.3. Problem Statement

The first goal of this research is to improve Rodin's automated theorem proving capabilities by developing a new proof tactic. The new proof tactic should be sound with a high degree of certainty according to common standards in the field of theorem proving. The performance of the new tactic should be at least as good as the performance of Rodin's auto-tactic, ideally even better. In comparison to Rodin's auto-tactic, it should be easier to improve the performance of the new tactic on problems from new domains; this capability is called *adaptability*. Good performance and adaptability should be demonstrated with experiments based on a challenging industrial case study.

In order to show (or even claim) that a proof tactic is sound, there has to be a rigorous definition of soundness. The notion of soundness depends on a semantics, and the semantics of a logic can only be defined once its syntax is clear. The second goal of this research is therefore to develop a comprehensive specification of Event-B's logic covering at least its syntax and semantics. In general, the logic specification should describe the logic implemented by Rodin and not a simplified version [such as Abrial, 2010]; exceptions to this rule need to be carefully justified. To support the further evolution of Event-B's logic, the motivations and consequences of non-trivial design decisions should be explained. It should be demonstrated with examples to which extent the specification of syntax and semantics is suitable for developing and reasoning about non-trivial features such as proof calculi, theory extension methods, or embedding techniques.

Although this problem statement is tailored to Event-B, the solution should apply to a more general setting whenever possible.

1.4. Approach to a Solution

Foundation of Event-B's Logic. It has been claimed (without further explanation) that Event-B uses a "first-order logic and set theory" [Abrial et al., 2010] or "set theory built on first order predicate calculus" [Mehta, 2007, 2008]; but Event-B's logic actually closely resembles higher-order logic [as in Gordon and Pitts, 1993]:

similar to higher-order logic, terms of Event-B are typed according to a Hindley-Milner style type discipline, the logic admits quantification over sets, and binders like set comprehension are part of the object logic. It is therefore natural to define the semantics of Event-B's logic by an embedding into some version of higher-order logic. The only major difference between Event-B's logic and higher-order logic is the way how Event-B treats partial functions; this is the main challenge that needs to be solved when defining a semantics.

Another challenge is to make sure that the semantics reflects the unwritten agreement in the Event-B community on which statements are supposed to be true. Since the existing literature makes contradictory statements (cf. Section 3.4), this is more a social process than a technical problem. It is crucial to clearly communicate what the semantics is so that an agreement is eventually achieved. In the Event-B literature [Abrial, 2010, Mehta, 2008] and in Rodin's documentation, it is common to specify the underlying semantics in terms of proof rules that are supposed to be sound. To help readers who are familiar with Event-B but unfamiliar with higher-order logic understand the semantics given in this thesis, I complement the definition of semantics with a proof calculus that highlights important semantic properties.

The meta-properties of higher-order logic have been extensively studied [see e.g. Andrews, 2002, Gordon and Pitts, 1993], which has lead to well-known results on soundness, completeness, incompleteness, and theory extension methods. By defining the semantics of Event-B's logic based on higher-order logic, it becomes easier to understand the meta-properties of Event-B's logic.

Theorem Proving. To improve Rodin's automated theorem proving capabilities, I have decided to follow Paulson's advice [Paulson, 1992]:

> Don't write a theorem prover. Try to use someone else's.

It is difficult to determine a priori which theorem prover is most suitable for a particular class of problems. I have decided to integrate Isabelle/HOL [Nipkow et al., 2002] into Rodin because Isabelle/HOL's strengths match Rodin's limitations very well.

Isabelle/HOL follows well-known principles to achieve soundness with a high degree of certainty, namely the LCF approach [Gordon et al., 1979] for the internal representation of theorems and definitional extension methods [Gordon and Pitts, 1993, Wenzel, 1997] for theory construction. Although the subset of Isabelle/HOL's source code that needs to be trusted (its *trusted kernel*) is bigger than in other highly trustworthy theorem provers [see Slind, 2010], very few soundness or conservativity bugs have been discovered. The latest bug in Isabelle's trusted kernel that is known to me has been fixed until 2007[1]. Isabelle/HOL is therefore widely accepted as a highly trustworthy theorem prover.

[1] Traces of this bug can be found in the NEWS file of `Isabelle2007`; it is related to circular definitions.

As a generic theorem prover, Isabelle provides powerful generic proof tactics that have matured during several years or decades and that can be adapted to new domains with little effort. Isabelle/HOL also integrates competition winning automated theorem provers [Böhme and Weber, 2010, Meng et al., 2006]; such an integration of theorem provers does not compromise soundness because proofs of external theorem provers are verified by Isabelle's trusted kernel.

Although an integration of Isabelle/HOL into Rodin seems promising, it is not obvious that it will be successful. One challenge stems from the different approaches to partial functions, i.e., functions in Event-B are by default partial, and functions in HOL are by default total. Moreover, Event-B provides several operators related to relations that have no counterparts in HOL. It is a priori unclear how difficult it is to configure Isabelle's automated methods to reason about these operators. And as in every engineering project, unexpected challenges may arise at any time.

1.5. Overview and Contributions

This thesis has two main contributions. The first contribution is a theoretical foundation of Event-B's logic; this covers topics like syntax, semantics, proofs, definitional extensions, and an analysis of the impact of various design decisions in a logic of partial functions. The second contribution is an improvement of Rodin's automated theorem proving capabilities in terms of soundness, adaptability, and sometimes even performance. Although the research has been driven by Event-B, some of the results are of a more general nature, as indicated below.

Chapter 2. The contribution of Chapter 2 is a comprehensive specification of abstract syntax, semantics, and proofs for Event-B's logic. Unlike other expositions on Event-B's logic [Abrial, 2010, Mehta, 2008, Metayer and Voisin, 2009], my presentation treats the sets of available symbols (type operators, operators, and binders) and proof rules as unknown parameters. This degree of generality is motivated by the observation that the symbols and proof rules available in Rodin change over time.

Alternative expositions on Event-B's logic still are valuable sources of information, but they are inadequate as specification documents for the following reasons: Abrial [2010] introduces Event-B's logic as a version of naive set theory (i.e., without types), which is known to be inconsistent; note that Rodin does not implement naive set theory. Mehta [2008] addresses only a fragment of Event-B's logic. Metayer and Voisin [2009] disregard semantics and proofs.

Chapter 3. Chapter 3 introduces Event-B's *main theory*, i.e., the symbols (type operator, operators, and binders) that are by default available in Rodin and a restricted set of proof rules. The semantics of symbols is either explicitly given, specified by appropriate proof rules, or often both. I show that the proof calculus given by Event-B's main theory is sound; to increase the degree of certainty that the soundness proof

is correct, I have carried out a substantial part of it in Isabelle/HOL. The first contribution of Chapter 3 is pragmatic: the chapter informs users and developers about the semantics of the symbols available in Rodin, and it shows that it is practically feasible to apply Isabelle/HOL to formally prove soundness of rules.

Chapter 3 also introduces *definitional theory extension methods*, i.e., methods for introducing new operators, binders, and proof rules without performing soundness proofs in Isabelle/HOL. Roughly speaking, the term *definitional* means that introducing new symbols or rules does not compromise soundness. Such methods have a long history in logics [see e.g. Shoenfield 1967, p. 41 or van Dalen 2004, p. 104], and the methods given in Chapter 3 closely resemble those of Isabelle and higher-order logic [Gordon and Pitts, 1993, Paulson, 1989, Wenzel, 1997]. The development of definitional theory extension methods is however more than a trivial transfer because Event-B's explicit support for partial functions poses novel challenges, in particular when defining binders and introducing rules. As a proof of concept, I have applied my theory extension methods to develop a substantial part of Event-B's main theory.

So the second contribution of Chapter 3 is the development of definitional theory extension methods: it shows that the foundations laid in Chapter 2 can be used to develop a non-trivial enhancement of the logic with reasonable effort. Moreover, it provides the first theoretical justification of the basic functionality of Rodin's theory plug-in [Butler and Maamria, 2010]. Overall, the process of writing Chapters 2 and 3 has led to the discovery of several consistency bugs in Rodin's theorem prover.

Chapter 4. Defining a logic of partial functions is easy; the challenge is to make it practically useful. While developing such a logic, one has to make numerous difficult design decisions; the consequences of these decisions are often unclear. The contribution of Chapter 4 is to review the main decisions underlying Event-B's logic, analyze their impact and relations between each other, and compare them to other approaches for modeling partial functions. Chapter 4 helps to foresee the consequences of design decisions related to partial functions.

The best discovery of Chapter 4 is in my view the observation how the need for solving preconditions during term rewriting is significantly reduced due to *directed rewriting*. This is a non-standard term rewriting technique for logics of partial functions. Although it has been implemented in Rodin and PVS [Owre et al., 1992], there is to my best knowledge no written account on the soundness or practical relevance of this technique[2]. I have developed a soundness argument in Section 2.3.2.3 and show in Section 4.1 that directed rewriting makes a significant number of conditional rewrite rules unconditional; directed rewriting therefore reduces the overhead of solving conditions during term rewriting.

[2]To be precise, I have published a paper on directed rewriting in the course of doing this research [Schmalz, 2011]. Dawson [1998] develops a version of directed rewriting for a fragment of LPF [Barringer et al., 1984], and Maamria and Butler [2010] for a limited version of Event-B (although with a flaw in their soundness proof). They do not analyze practical relevance, and it is not trivial to carry their soundness proofs over to (full) Event-B or PVS.

Chapter 5. In Chapter 5, I report on the lessons learned during the integration of Isabelle/HOL as an automated theorem prover into Rodin. The starting point of a proof attempt is a naive encoding of the Event-B proof obligation in HOL; unfortunately, Isabelle/HOL's predefined proof methods perform poorly on this encoding. I therefore develop a preprocessing phase that rewrites the naive encoding of a proof obligation to a form that is more suitable for automated theorem proving (cf. Section 5.2).

This preprocessing, which I call *unlifting*, is a problem of general interest: it is the problem of translating from a logic that explicitly supports partial functions to a classical logic, which supports only total functions. The problem of unlifting arises whenever a theorem prover for classical logic is applied to a logic of partial functions; several algorithms have been proposed in the literature [Abrial and Mussat, 2002, Berezin et al., 2005, Darvas et al., 2008, Owre and Shankar, 1999, Woodcock et al., 2009]. The unlifting algorithm in Section 5.2 is more general than all of these algorithms because it is able to cope with partial functions whose domains are unknown. Other contributions are a detailed complexity analysis (Section 5.2.4), an empirical evaluation showing that my implementation is efficient on problems from various (including industrial) domains (Section 5.2.6), and an implementation based on Isabelle's simplifier [Nipkow, 1989] (Section 5.2.5). The implementation based on Isabelle has the advantage that the unlifting algorithm is sound by construction and can be optimized easily without compromising soundness.

Another contribution is a novel proof method for Isabelle/HOL, called `axe`, which is invoked on the output of the unlifting algorithm. I have evaluated the performance of `axe` on conjectures generated from academic and industrial developments whose sizes range up to 18000. (The size of a formula is the number of leaves of its abstract syntax tree.) On the academic benchmark, the performance of `axe` is similar to the performance of Rodin's auto-tactic. On the industrial benchmark, `axe` discharges significantly more proof obligations automatically at the price of an increased runtime. Since Rodin's auto-tactic has been optimized for years and the `axe` method only for months, I expect that it is practically feasible to further improve the performance of `axe`.

In my empirical evaluation, I make a clear distinction between a *training benchmark* used to drive the optimization process and a *validation benchmark* that does not influence the optimization process; this approach allows me to measure generality of the various performance improvements. In a few cases, an improvement on the training benchmark caused minor performance regressions on the validation benchmark; but in all other cases, the generalization was good or even excellent. This is a clear sign that improving `axe` (or other proof methods) often pays off in the long run.

The `axe` method is a non-trivial combination of Isabelle's predefined proof methods (such as `auto`, `blast`, `force`, `metis`, and `smt`). Given that `axe` is a combination of predefined methods, it is not surprising that `axe` outperforms the methods it is built from. A major challenge is to decide when to invoke which of Isabelle's predefined proof methods, with which parameters, and how long to wait for termination. In

typical Isabelle proofs, the user makes these decisions; `axe` improves over Isabelle's predefined proof methods by making these decisions for the user.

Another reason why `axe` outperforms Isabelle's predefined proof methods on the considered benchmarks is that it carefully avoids or delays brute force strategies such as naive case splitting. This improves the performance of `axe` on large problems.

Overall, the main contribution of Chapter 5 is a novel automated tactic for Rodin, which is sound by construction. On one of the two considered benchmarks, this tactic discharges roughly as many proof obligations as Rodin's auto-tactic; on the other one (with larger proof obligations), it discharges significantly more proof obligations. The tactic inherits Isabelle's great support for adapting tactics to new domains. So Chapter 5 shows that an integration of Isabelle/HOL into Rodin is practically feasible and useful.

Chapter 6. In Chapter 6, I review the main results, address future work, and conclude with personal remarks on this research.

2. Event-B's Logic

2.1. Abstract Syntax

In the following I define the abstract syntax of Event-B's logic. The attribute "abstract" means that the given syntax is restricted to features that are essential for developing a semantics and reasoning about Event-B's logic. In particular, I will neither give a rigorous account of infix notation nor specify which kinds of strings are admissible as identifiers; these topics are important when developing a parser but dispensable when reasoning about the logic. A slightly outdated specification of the *concrete* syntax of Event-B's logic, i.e., the input language of Rodin's parser, can be found in [Metayer and Voisin, 2009].

To simplify the presentation and the development of not yet implemented improvements, I have sometimes taken the liberty to generalize over the version of Event-B's logic implemented by Rodin. Minor deviations from Rodin's implementation are addressed on the fly; major deviations are justified in Section 2.4.

Sequences. Sequences are not part of Event-B's syntax, but they are essential for describing it; I therefore introduce several notational conventions. The sequence of length n with t_i at position i for $1 \leq i \leq n$ is denoted by $t_1, \ldots, t_n$ or alternatively $t_1 \ \ldots \ t_n$. Variables denoting sequences are written in bold and underlined. If the sequence $\underline{\boldsymbol{t}}$ has at least i elements, t_i stands for the element at position i in $\underline{\boldsymbol{t}}$. Moreover, $|\underline{\boldsymbol{t}}|$ stands for $\underline{\boldsymbol{t}}$'s length and $\underline{\boldsymbol{t}}\,\underline{\boldsymbol{t}}'$ for the concatenation of $\underline{\boldsymbol{t}}$ and $\underline{\boldsymbol{t}}'$. I will introduce more conventions when needed.

Symbols. The syntactic entities of Event-B's logic are built from *symbols* arranged in the following pairwise disjoint categories:

1. *delimiters*: $(,)$, $\cdot$, $|$, \$, $\mathbb{8}$, $\vdash$, $\sqsubseteq$, $\equiv$,
2. *type variables*, usually denoted by α, β,
3. *variable names*, usually denoted by x, y, z,
4. *non-logical symbols*, such as $\Rightarrow, =, 1, \forall, \mathcal{Z}$.

I assume that each of the categories 2, 3, and 4 comprises infinitely many symbols. Symbols in the categories 1, 2, and 3 are also referred to as *logical* symbols.

2.1.1. Types and Terms

Informally, the symbols that may be used to build syntactic entities are given by a *signature*. More precisely, a *signature* Σ comprises three pairwise disjoint sets of non-logical symbols: a set of *type operators*, a set of *(ordinary) operators*, and a set of *binders*. In the following, I will explain how the symbols of a signature are used to form types and terms. Readers familiar with higher-order logic [see e.g. Gordon and Melham, 1993] or the programming language ML will recognize that Event-B has a Hindley-Milner style type system.

Types. A signature Σ assigns a non-negative integer, called *arity*, to each type operator. Every signature includes the *boolean* type operator $\mathcal{B}$ of arity zero. A type operator of arity zero is also called *basic type*. The set of *types (over Σ)* is the smallest set such that every type variable is a type, and $\tau(\underline{\boldsymbol{\nu}})$ is a type if $\underline{\boldsymbol{\nu}}$ is a sequence of types and τ a type operator of arity $|\underline{\boldsymbol{\nu}}|$. If ν is a basic type, I usually write ν instead of $\nu()$.

Example 2.1. Examples of type operators in Rodin include the basic type $\mathcal{Z}$ (*integer type*) and the *set* type operator $\mathcal{P}$ of arity 1. Informally, the type $\mathcal{Z}$ denotes the set of all integers and $\mathcal{P}(\mathcal{Z})$ the set of all sets of integers; α stands for an unspecified nonempty set and $\mathcal{P}(\alpha)$ for the set of all subsets of α. Type variables are closest to Rodin's *given types*, which are also known as *carrier sets* or *type parameters*. □

A *(type) substitution σ (over Σ)* consists of a sequence $\underline{\boldsymbol{\alpha}}$ of pairwise distinct type variables and a sequence $\underline{\boldsymbol{\mu}}$ of types, where $|\underline{\boldsymbol{\alpha}}| = |\underline{\boldsymbol{\mu}}|$, and is written $[\underline{\boldsymbol{\alpha}} := \underline{\boldsymbol{\mu}}]$. The substitution σ maps the type ν to the type $\nu\sigma$ obtained by simultaneously replacing every occurrence of α_i by μ_i for $1 \leq i \leq |\underline{\boldsymbol{\alpha}}|$ in ν. The elements of $\underline{\boldsymbol{\mu}}$ are called the *right-hand sides* of σ. The type sequence $\underline{\boldsymbol{\nu'}}$ is an *instance* of $\underline{\boldsymbol{\nu}}$ iff $|\underline{\boldsymbol{\nu'}}| = |\underline{\boldsymbol{\nu}}|$ and there is a substitution σ such that $\nu'_i = \nu_i\sigma$, for $1 \leq i \leq |\underline{\boldsymbol{\nu}}|$.

Terms. The signature Σ associates with each ordinary operator f a sequence of types $\underline{\boldsymbol{\nu}}$ (the *argument type*) and a type μ (the *result type*), written as $f \mathbin{:\!\!\sim} \underline{\boldsymbol{\nu}} \rightharpoonup \mu$. A *constant* $c \mathbin{:\!\!\sim} \mu$ is an operator with empty argument type and result type μ. With each binder Q the signature Σ associates a non-empty sequence $\underline{\boldsymbol{\nu}}$ (the *bound variable type*), a non-empty sequence $\underline{\boldsymbol{\mu}}$ (the *argument type*), and a type ξ (the *result type*), written as $Q \mathbin{:\!\!\sim} (\underline{\boldsymbol{\nu}} \rightharpoonup \underline{\boldsymbol{\mu}}) \rightharpoonup \xi$.

Example 2.2. Examples of operators in Rodin include $0 \mathbin{:\!\!\sim} \mathcal{Z}$, *conjunction* $\wedge \mathbin{:\!\!\sim} (\mathcal{B}, \mathcal{B}) \rightharpoonup \mathcal{B}$, and *membership* $\in \mathbin{:\!\!\sim} (\alpha, \mathcal{P}(\alpha)) \rightharpoonup \mathcal{B}$. An example of a binder is the *universal quantifier* $\forall \mathbin{:\!\!\sim} (\alpha \rightharpoonup \mathcal{B}) \rightharpoonup \mathcal{B}$, which informally takes a function mapping elements of α to booleans and yields a boolean. □

An *(ordinary) variable* $x \mathbin{:\!\!\sim} \nu$ consists of a variable name x and a type ν. An *operator variable* $\$f \mathbin{:\!\!\sim} \underline{\boldsymbol{\nu}} \rightharpoonup \mu$ consists of a variable name f, *argument types* $\underline{\boldsymbol{\nu}}$, and a *result type* μ; if $\underline{\boldsymbol{\nu}}$ is empty, I usually write $\$f \mathbin{:\!\!\sim} \mu$ instead of $\$f \mathbin{:\!\!\sim\!\!\rightharpoonup} \mu$.

Letter	Usage
α, β	type variable
τ	type operator
ν, μ, ξ	type
x, y, z	ordinary variable or variable name
f, g	operator
c	constant
t, u	term
φ, ψ, χ	formula
$\$f, \$g$	operator variable
$\$x, \$y, \$z$	operator variable with empty argument type
$\$\varphi, \$\psi, \$\chi$	operator variable of result type $\mathcal{B}$

These conventions also apply to letters decorated with subscripts and primes.

Figure 2.1.: Naming conventions

Conditions T1-T4 below inductively define *terms (over Σ)* and their types. The statement "$\underline{t}$ *has type* $\underline{\nu}$", abbreviated as $\underline{t} \mathbin{⦂} \underline{\nu}$, indicates that $|\underline{t}| = |\underline{\nu}|$ and t_i is of type ν_i, $1 \leq i \leq |\underline{t}|$.

T1: Every ordinary variable of type ν is a term of type ν.

T2: If $f \mathbin{⦂} \underline{\nu} \rightharpoonup \mu$ is an operator and $\underline{t} \mathbin{⦂} \underline{\nu}'$ a sequence of terms, then $f(\underline{t} \mathbin{⦂} \underline{\nu}') \mathbin{⦂} \mu'$ is a term of type μ', provided $\underline{\nu}'\, \mu'$ is an instance of $\underline{\nu}\, \mu$.

T3: If $\$f \mathbin{⦂} \underline{\nu} \rightharpoonup \mu$ is an operator variable and $\underline{t} \mathbin{⦂} \underline{\nu}$ a sequence of terms, then $\$f(\underline{t} \mathbin{⦂} \underline{\nu}) \mathbin{⦂} \mu$ is a term of type μ.

T4: If $Q \mathbin{⦂} (\underline{\nu} \rightharpoonup \underline{\mu}) \rightharpoonup \xi$ is a binder, $\underline{x} \mathbin{⦂} \underline{\nu}'$ a sequence of pairwise distinct variables, and $\underline{t} \mathbin{⦂} \underline{\mu}'$ a sequence of terms, then $(Q\, \underline{x} \mathbin{⦂} \underline{\nu}' \cdot \underline{t} \mathbin{⦂} \underline{\mu}') \mathbin{⦂} \xi'$ is a term of type ξ', provided $\underline{\nu}'\, \underline{\mu}'\, \xi'$ is an instance of $\underline{\nu}\, \underline{\mu}\, \xi$, $|\underline{\nu}'| = |\underline{\nu}|$, and $\left|\underline{\mu}'\right| = \left|\underline{\mu}\right|$.

Given a constant c, I usually write $c \mathbin{⦂} \mu$ for the term $c() \mathbin{⦂} \mu$. A term of type $\mathcal{B}$ is called a *formula*. Figure 2.1 summarizes which letter is used for which syntactic entity.

Rodin imposes further restrictions on terms: Rodin's parser may reject a term if it contains variables with the same names and different types. Rodin's type checker stipulates that type variables are declared before their first usage. I ignore these restrictions because they have no logical significance and would merely complicate the presentation without adding clarity.

I adopt the usual definitions of *bound* and *free* (ordinary) variables. For uniformity, I also view operator and type variables as *free*. Unless mentioned otherwise, I consider *alpha-congruent* terms, i.e., terms that informally speaking differ only in the names

of bound variables, as identical; Hindley and Seldin [2008] give precise definitions of these notions.

Extending Signatures. Extending a signature informally means introducing additional symbols. Formally, a signature Σ' *extends* a signature Σ iff the type operators, operators, and binders of Σ are type operators, operators, and binders of Σ', respectively, and Σ' assigns arities, bound variable, argument, and result types to symbols of Σ in the same way as Σ.

2.1.2. An Example Signature

As a running example, I define the signature Σ_0 introducing the following symbols:

Type operators:

- the basic types $\mathcal{B}$ (of *booleans*) and $\mathcal{Z}$ (of *integers*),
- the *set* type operator $\mathcal{P}$ of arity 1.

Operators:

- $\mathsf{D} \mathbin{:} \alpha \rightharpoonup \mathcal{B}$ (*well-definedness*),
- $= \mathbin{:} (\alpha, \alpha) \rightharpoonup \mathcal{B}$ (*equality*),
- $\top \mathbin{:} \mathcal{B}$ (*truth*), $\bot \mathbin{:} \mathcal{B}$ (*falsity*), and $\bullet \mathbin{:} \alpha$ (*ill-definedness*),
- $\neg \mathbin{:} \mathcal{B} \rightharpoonup \mathcal{B}$ (*negation*) and $\wedge \mathbin{:} (\mathcal{B}, \mathcal{B}) \rightharpoonup \mathcal{B}$ (*conjunction*),
- $\vee \mathbin{:} (\mathcal{B}, \mathcal{B}) \rightharpoonup \mathcal{B}$ (*disjunction*) and $\Rightarrow \mathbin{:} (\mathcal{B}, \mathcal{B}) \rightharpoonup \mathcal{B}$ (*implication*),
- $\in \mathbin{:} (\alpha, \mathcal{P}(\alpha)) \rightharpoonup \mathcal{B}$ (*membership*) and $\mathbb{P} \mathbin{:} \mathcal{P}(\alpha) \rightharpoonup \mathcal{P}(\mathcal{P}(\alpha))$ (*powerset*),
- $\varnothing \mathbin{:} \mathcal{P}(\alpha)$ (*empty set*) and $\cap \mathbin{:} (\mathcal{P}(\alpha), \mathcal{P}(\alpha)) \rightharpoonup \mathcal{P}(\alpha)$ (*intersection*),
- $0 \mathbin{:} \mathcal{Z}$, $1 \mathbin{:} \mathcal{Z}$, $\mathbb{Z} \mathbin{:} \mathcal{P}(\mathcal{Z})$, and $\mathsf{mod} \mathbin{:} (\mathcal{Z}, \mathcal{Z}) \rightharpoonup \mathcal{Z}$ (*modulo*).

Binders:

- $\forall \mathbin{:} (\alpha \rightharpoonup \mathcal{B}) \rightharpoonup \mathcal{B}$ (*universal quantifier*),
- $\mathsf{collect} \mathbin{:} (\alpha \rightharpoonup \mathcal{B}) \rightharpoonup \mathcal{P}(\alpha)$ (*simple set comprehension*).

I usually write $\{x \mid \varphi\}$ for $\mathsf{collect}\, x \cdot \varphi$ and $\forall \underline{\boldsymbol{x}} \cdot \varphi$ for $\forall x_1 \cdot \ldots \forall x_{|\underline{\boldsymbol{x}}|} \cdot \varphi$. The names of the symbols (e.g., negation for $\neg$) should be viewed as hints to their standard semantics, which I will make precise in Section 2.2.6.

Conventions. To improve readability, type constraints "$\mathbin{⦂} \nu$" are omitted when there is no danger of confusion. I use infix notation for most operators taking two arguments and leave out parentheses when the precedence is clear or irrelevant. By default, operators taking one argument have higher precedence than operators in infix notation; e.g., $\neg\varphi \wedge \psi$ stands for $(\neg\varphi) \wedge \psi$. The scope of a binder extends as far to the right as possible, e.g., $\forall x \cdot \varphi \wedge \psi$ is to be read as $\forall x \cdot (\varphi \wedge \psi)$. By default, a term is of the most general type, and I assign the same types to different occurrences of a variable name. Hence, $\forall x \cdot x \in R \wedge x = x$ stands for

$$\forall x \mathbin{⦂} \alpha \cdot (x \mathbin{⦂} \alpha) \in (R \mathbin{⦂} \mathcal{P}(\alpha)) \wedge (x \mathbin{⦂} \alpha) = (x \mathbin{⦂} \alpha).$$

2.2. Semantics

For the reasons explained in the introduction (cf. Section 1.4), I have decided to define the semantics of Event-B's logic by an embedding into some version of higher-order logic. As I plan to integrate Isabelle/HOL as a theorem prover into Rodin, it makes sense to define the semantics in terms of Isabelle's version of higher-order logic. I will address the question whether I have defined the "right" semantics at the beginning of Chapter 3.

2.2.1. Isabelle/HOL

The term *Isabelle* refers to the generic theorem prover designed by Paulson [1989] that supports various logics. The abbreviation HOL stands for higher-order logic as implemented by Isabelle. The instantiation of Isabelle by HOL is denoted by Isabelle/HOL [Nipkow et al., 2002].

For readers that are not familiar with Isabelle/HOL, I give an informal summary of the features that are relevant for my thesis. HOL's type system has much in common with ML's. The details are elaborated in the texts by Gordon and Pitts [1993], and Wenzel [1997]. The notation $t :: \nu$ indicates that the term t has type ν. A term of type $\nu \Rightarrow \mu$ denotes a function taking one argument of type ν and yielding a result of type μ.

Functions taking n arguments, $n > 0$, are represented by terms of type $\nu_1 \Rightarrow \cdots \Rightarrow \nu_{n+1}$. The type operator $\Rightarrow$ associates to the right. The application of the function f to the arguments $x_1, \ldots, x_{|\underline{\boldsymbol{x}}|}$ is written $f\ x_1\ \ldots\ x_{|\underline{\boldsymbol{x}}|}$ or $f\ \underline{\boldsymbol{x}}$ and should be read as $(\ldots(f\ x_1)\ \ldots)\ x_{|\underline{\boldsymbol{x}}|}$.

HOL supports infix notations for some functions. For example, $\mathsf{plus}\ x\ y$ can be written as $x + y$. Function applications have higher precedence than infix operators: $f\ x + 1$ is to be read as $(f\ x) + 1$.

Terms of type $\nu \Rightarrow \mu$ represent total functions. It is also possible to encode partial functions in HOL, but some partial functions are simply approximated by total functions: HOL's integer division div is a total function of type $\mathsf{int} \Rightarrow \mathsf{int} \Rightarrow \mathsf{int}$. The developers of HOL have decided that $x\ \ \mathsf{div}\ \ (0 :: \mathsf{int})$ equals 0. The way how

partial function are approximated by total functions varies from case to case: in particular, the "least integer" Least $\{x :: \mathsf{int}.\ \mathsf{True}\}$ is left unspecified.

In a nutshell, a HOL theory comprises various kinds of declarations, in particular definitions of constants, theorems with proofs, and postulates of axioms. Within this document, I say that a HOL theory THY′ *extends* another HOL theory THY iff THY′ includes all declarations of THY. Informally, THY′ *definitionally* extends THY iff THY′ extends THY and THY′ only includes postulates of axioms contained in THY. This definition of "definitional" is somewhat imprecise because some kinds of declarations postulate new axioms in some versions of HOL and are definitional in others; Wenzel [1997] carefully explains why type definitions (and therefore also datatype declaration) constitute examples of such declarations.

To dispel any doubt, I say that THY′ *definitionally* extends THY iff THY′ extends THY and every formula φ built from symbols declared in THY is valid[1] in THY′ iff φ is valid in THY. For the purposes of this document, it suffices to know that HOL's constant definitions and specifications are definitional in this sense [as shown by Gordon and Pitts, 1993, Wenzel, 1997].

2.2.2. Preliminaries – the Theory EB_Prelims

The theory EB_Prelims introduces auxiliary notation, which I use later in the definition of the actual semantics. The main theory of HOL (called Main) introduces *option types*, defined by

$$\textbf{datatype}\ \alpha\ \mathsf{option} = \mathsf{Some}\,\alpha \mid \mathsf{None}.$$

Intuitively, the type α option contains "copies" of all elements x of the type α, written as $\mathsf{Some}\,x$. Additionally, the type α option contains a constant None that is distinct from every "copy" $\mathsf{Some}\,x$. For brevity, the theory EB_Prelims introduces the notation $\alpha\uparrow$ for α option, $x\uparrow$ for $\mathsf{Some}\,x$, and $\bullet$ for None.

Moreover, EB_Prelims defines the functions WD, T, F, and WT by

$$\mathsf{WD}\,x = (x \neq \bullet),$$
$$\mathsf{T}\,\varphi = (\varphi = \mathsf{True}\uparrow), \qquad \mathsf{F}\,\varphi = (\varphi = \mathsf{False}\uparrow), \qquad \mathsf{WT}\,\varphi = (\varphi \neq \mathsf{False}\uparrow).$$

For a term t of type $\nu\uparrow$, the formula $\mathsf{WD}\,t$ indicates whether t is *well-defined*, i.e., distinct from $\bullet$. The opposite of well-defined is *ill-defined*. If φ is a term of type bool↑, then $\mathsf{T}\,\varphi$ indicates whether φ is *true*, i.e., equal to True↑, and $\mathsf{F}\,\varphi$ whether φ is *false*, i.e., equal to False↑. The formula $\mathsf{WT}\,\varphi$ indicates whether φ is *weakly true*, i.e., true or ill-defined.

[1] The literature distinguishes several notions of validity for higher-order logic [see Andrews, 2002]. Within this document, *validity* always refers to the standard semantics of higher-order logic [as in Andrews, 2002, Gordon and Pitts, 1993]; in particular, the type $\alpha \Rightarrow \beta$ denotes the set of *all* functions that map the set denoted by α into the set denoted by β.

2.2.3. Denotations of Types and Terms

Given a signature Σ, a *structure (over Σ)* specifies the *denotations* of Event-B's types and terms over Σ. Technically, a structure $(\mathsf{M}, [\![\cdot]\!])$ consists of a HOL theory M extending $\mathsf{EB\_Prelims}$ and a *denotation function* $[\![\cdot]\!]$ mapping Event-B's type operators, types, operators, binders, and terms to HOL's type operators, types, constants, and terms.

I adopt the convention that $[\![\underline{\boldsymbol{t}}]\!]$ abbreviates $[\![t_1]\!]\ \ldots\ [\![t_{|\underline{\boldsymbol{t}}|}]\!]$, for a sequence $\underline{\boldsymbol{t}}$ of types or terms. If the interpretation function $[\![\cdot]\!]$ is clear from the context, I designate the structure $(\mathsf{M}, [\![\cdot]\!])$ by M.

The denotation $[\![\alpha]\!]$ of an Event-B type variable α is defined as α; here I assume, without loss of generality, that every Event-B type variable is a HOL type variable. For every Event-B type operator τ of arity n, $[\![\tau]\!]$ is a HOL type operator taking n arguments[2]. The boolean type $\mathcal{B}$ denotes bool, and a type $\tau(\underline{\boldsymbol{\nu}})$ denotes the type $([\![\underline{\boldsymbol{\nu}}]\!])\ [\![\tau]\!]$, i.e., the result of applying the type operator $[\![\tau]\!]$ to the types $[\![\underline{\boldsymbol{\nu}}]\!]$.

While the denotation of a type is obtained by renaming type operators, the situation is more involved for terms. The denotation function $[\![\cdot]\!]$ maps operators $f \wr \underline{\boldsymbol{\nu}} \rightharpoonup \mu$ to HOL constants of type $[\![\nu_1]\!]\uparrow \Rightarrow \ldots \Rightarrow [\![\nu_{|\underline{\boldsymbol{\nu}}|}]\!]\uparrow \Rightarrow [\![\mu]\!]\uparrow$ and binders $Q \wr (\underline{\boldsymbol{\nu}} \rightharpoonup \underline{\boldsymbol{\mu}}) \rightharpoonup \xi$ to HOL constants of type

$$([\![\nu_1]\!] \Rightarrow \ldots \Rightarrow [\![\nu_{|\underline{\boldsymbol{\nu}}|}]\!] \Rightarrow [\![\mu_1]\!]\uparrow) \Rightarrow \ldots \Rightarrow ([\![\nu_1]\!] \Rightarrow \ldots \Rightarrow [\![\nu_{|\underline{\boldsymbol{\nu}}|}]\!] \Rightarrow [\![\mu_{|\underline{\boldsymbol{\mu}}|}]\!]\uparrow) \Rightarrow [\![\xi]\!]\uparrow.$$

An Event-B term of type ν denotes a HOL term of type $[\![\nu]\!]\uparrow$ as follows:

1. $[\![x \wr \nu]\!] = (x :: [\![\nu]\!])\uparrow$.
2. $[\![f(\underline{\boldsymbol{t}}) \wr \mu']\!] = (([\![f]\!]\ [\![\underline{\boldsymbol{t}}]\!]) :: [\![\mu']\!]\uparrow)$.
3. $[\![\$f(\underline{\boldsymbol{t}}) \wr \mu]\!] = ((\$f\ [\![\underline{\boldsymbol{t}}]\!]) :: [\![\mu]\!]\uparrow)$.
4. $[\![(Q\underline{\boldsymbol{x}} \wr \underline{\boldsymbol{\nu}} \cdot \underline{\boldsymbol{t}}) \wr \xi']\!] = (([\![Q]\!]\ (\lambda\underline{\boldsymbol{x}} :: [\![\underline{\boldsymbol{\nu}}]\!].\ [\![t_1]\!])\ \ldots\ (\lambda\underline{\boldsymbol{x}} :: [\![\underline{\boldsymbol{\nu}}]\!].\ [\![t_{|\underline{\boldsymbol{t}}|}]\!])) :: [\![\xi']\!]\uparrow)$.

For convenience, it is assumed that, for each Event-B variable name x, both x and $\$x$ are available as variable names in M. In 4, the notation $\underline{\boldsymbol{x}} :: \underline{\boldsymbol{\nu}}$ abbreviates $(x_1 :: \nu_1)\ \ldots\ (x_{|\underline{\boldsymbol{x}}|} :: \nu_{|\underline{\boldsymbol{\nu}}|})$.

Remark 2.3. A particular property of Event-B's semantics is that types are *flat*. Suppose that the underlying signature has a function type operator $\rightarrow$, which is interpreted as $\Rightarrow$. Then, Event-B terms of type $\alpha \rightarrow \beta$ denote HOL terms of type $(\alpha \Rightarrow \beta)\uparrow$. Consequently, a function is either well-defined or ill-defined as a whole. In a non-flat interpretation of function types, Event-B terms of type $\alpha \rightarrow \beta$ denote HOL terms of type $\alpha\uparrow \Rightarrow \beta\uparrow$ or perhaps $\alpha \Rightarrow \beta\uparrow$. Flatness of all types distinguishes Event-B from LCF [see Paulson, 1987].

Recall that HOL provides only one operator, namely function application, and one binder, namely lambda-abstraction, and views the remaining operators and binders

[2] In this regard, I also view Isabelle's *type synonyms* as type operators.

as constants of suitable function types. Since Event-B types are flat, it is impossible to structure Event-B's logic in a similar way: Event-B constants denote terms of type $\nu\uparrow$ and are therefore in general unsuitable to represent operators and binders. □

2.2.4. Extending Structures

Informally, there are two ways of extending a structure: extend the underlying signature or extend the underlying HOL theory. Formally, a structure $(\mathsf{M}', [\![\cdot]\!]')$ over Σ' *extends* a structure $(\mathsf{M}, [\![\cdot]\!])$ over Σ iff Σ' extends Σ, M' extends M, and the restriction of $[\![\cdot]\!]'$ to symbols, types, and terms of Σ coincides with $[\![\cdot]\!]$. Moreover, $(\mathsf{M}', [\![\cdot]\!]')$ *definitionally* extends $(\mathsf{M}, [\![\cdot]\!])$ iff $(\mathsf{M}', [\![\cdot]\!]')$ extends $(\mathsf{M}, [\![\cdot]\!])$ and M' definitionally extends M.

2.2.5. Semantic Notions

An Event-B term is *well-defined* (*ill-defined*) iff its denotation is well-defined (ill-defined). The Event-B terms t and u are *equivalent* iff $[\![t]\!] = [\![u]\!]$ is valid. An Event-B formula φ is *valid* iff $\mathsf{T}\ [\![\varphi]\!]$ is valid.

For the following definitions, I suppose that Q is a HOL function of type

$$([\![\nu_1]\!] \Rightarrow \ldots \Rightarrow [\![\nu_n]\!] \Rightarrow [\![\mu_1]\!]\uparrow) \Rightarrow \ldots \Rightarrow ([\![\nu_1]\!] \Rightarrow \ldots \Rightarrow [\![\nu_n]\!] \Rightarrow [\![\mu_m]\!]\uparrow) \Rightarrow [\![\xi]\!]\uparrow,$$

where $n, m \geq 0$. If $n > 0$ and $m > 0$, Q may serve as the denotation of a binder. If $n = 0$, one should think of Q as the denotation of an operator.

Domain. The *domain* of Q is the set $\{(\underline{\boldsymbol{f}}).\ \mathsf{WD}\ Q\ \underline{\boldsymbol{f}}\}$. The *domain* of an Event-B operator or binder is the domain of its denotation.

Definiteness. The function Q is *definite* iff

$$\forall \underline{\boldsymbol{x}}.\ \mathsf{WD}\ (g_1\ \underline{\boldsymbol{x}}) \wedge \cdots \wedge \mathsf{WD}\ (g_m\ \underline{\boldsymbol{x}}) \qquad \text{implies} \qquad \mathsf{WD}\ (Q\ g_1\ \ldots\ g_m).$$

An Event-B operator or binder is *definite* iff its denotation is.

Smashedness and Strictness. The function Q is *smashed in its ith argument* iff

$$\mathsf{WD}\ (Q\ g_1\ \ldots\ g_m) \qquad \text{implies} \qquad \forall \underline{\boldsymbol{x}}.\ \mathsf{WD}\ (g_i\ \underline{\boldsymbol{x}}),$$

where $1 \leq i \leq m$. Moreover, Q is *smashed* iff Q is smashed in all arguments. The function Q is the *smashed extension* of a function Q' of type

$$([\![\nu_1]\!] \Rightarrow \ldots \Rightarrow [\![\nu_n]\!] \Rightarrow [\![\mu_1]\!]) \Rightarrow \ldots \Rightarrow ([\![\nu_1]\!] \Rightarrow \ldots \Rightarrow [\![\nu_n]\!] \Rightarrow [\![\mu_m]\!]) \Rightarrow [\![\xi]\!]$$

iff the following formula is valid:

$$Q\ g_1\ \ldots\ g_m = \begin{cases} (Q'\ g'_1\ \ldots\ g'_m)\uparrow & (\forall \underline{\boldsymbol{x}}.\ g_i\ \underline{\boldsymbol{x}} = (g'_i\ \underline{\boldsymbol{x}})\uparrow \text{ for } 1 \leq i \leq m) \\ \bullet & (\exists \underline{\boldsymbol{x}}.\ \neg\mathsf{WD}\ (g_i\ \underline{\boldsymbol{x}}) \text{ for some } i \text{ with } 1 \leq i \leq m). \end{cases}$$

In the case of $n = 0$, it is common to replace the term smashed by the term *strict*. The opposite of strict is *lazy*. The notions of strictness and smashedness are carried over to operators and binders of Event-B in the obvious way; e.g., an operator is strict in its first argument iff its denotation is.

Monotonicity. The *flat domain order* $\sqsubseteq$ [see e.g. Paulson, 1987] is defined by

$$\forall x\ y.\ x \sqsubseteq y \longleftrightarrow (\mathsf{WD}\ x \longrightarrow x = y).$$

The order $\sqsubseteq$ is lifted to functions in the usual point-wise fashion:

$$f \sqsubseteq g \longleftrightarrow (\forall x.\ f\ x\ \sqsubseteq g\ x).$$

The function Q is *monotonic* iff

$$\forall \underline{\boldsymbol{f}}\ \underline{\boldsymbol{g}}.\ f_1 \sqsubseteq g_1 \wedge \cdots \wedge f_{|\underline{\boldsymbol{f}}|} \sqsubseteq g_{|\underline{\boldsymbol{g}}|} \qquad \text{implies} \qquad Q\ \underline{\boldsymbol{f}} \sqsubseteq Q\ \underline{\boldsymbol{g}}.$$

An Event-B operator or binder is *monotonic* iff its denotation is.

2.2.6. An Example Structure

The structure $(\mathsf{EB}_0, [\![\cdot]\!])$ defines the denotations of the types and terms of Σ_0, the signature defined in Section 2.1.2. I start with the denotations of the given type operators, the well-definedness operator D, and ill-definedness $\bullet$:

$$[\![\mathcal{B}]\!] = \mathsf{bool}, \qquad [\![\mathcal{Z}]\!] = \mathsf{int}, \qquad [\![\mathcal{P}]\!] = \mathsf{set},$$
$$[\![\mathsf{D}]\!]\ x = (\mathsf{WD}\ x)\uparrow, \qquad [\![\bullet]\!] = \bullet.$$

The operator D is definite but not monotonic (and therefore not strict).

Most operators f of Σ_0 denote the strict extensions of their counterparts f^{HOL} in HOL according to the following table:

f	$=$	$\top$	$\bot$	$\neg$	$\in$	$\mathbb{P}$	$\varnothing$	$\cap$	0	1	$\mathbb{Z}$
f^{HOL}	$=$	True	False	$\neg$	$\in$	Pow	$\{\}$	$\cap$	$0 :: \mathsf{int}$	$1 :: \mathsf{int}$	UNIV :: int

If x is an ordinary variable, then $x = x$ is valid, because x is well-defined. If however $\$x$ is an operator variable, then $\$x = \$x$ is not valid, because $\$x$ could be ill-defined.

The operator mod is strict but not definite, because it yields an ill-defined result if an argument is negative:

$$[\![\mathsf{mod}]\!]\ x\uparrow\ y\uparrow = \begin{cases} (x\ \mathsf{mod}\ y)\uparrow & (x \geq 0 \wedge y > 0) \\ \bullet & (\text{otherwise}). \end{cases}$$

The denotation of conjunction is given by

$$[\![\wedge]\!]\ \varphi\ \psi = \begin{cases} \mathsf{True}\uparrow & (\mathsf{T}\ \varphi \wedge \mathsf{T}\ \psi) \\ \mathsf{False}\uparrow & (\mathsf{F}\ \varphi \vee \mathsf{F}\ \psi) \\ \bullet & (\text{otherwise}). \end{cases}$$

The denotations of disjunction and implication are defined such that $\$\varphi \vee \$\psi$ is equivalent to $\neg(\neg\$\varphi \wedge \neg\$\psi)$ and $\$\varphi \Rightarrow \$\psi$ is equivalent to $\neg\$\varphi \vee \$\psi$. Note that conjunction, disjunction, and implication are not strict in either argument. In particular, both $\bot \wedge \$\varphi$ and $\$\varphi \wedge \bot$ are equivalent to $\bot$. Contrast this to intersection: neither $\varnothing \cap \$R$ nor $\$R \cap \varnothing$ is equivalent to $\varnothing$, because $\$R$ is not necessarily well-defined and intersection is strict.

The denotation of collect is the smashed extension of HOL's Collect, and the denotation of $\forall$ is defined such that

$$[\![\forall x \cdot \varphi]\!] = \begin{cases} \mathsf{True}\uparrow & (\forall x.\ \mathsf{T}\ [\![\varphi]\!]) \\ \mathsf{False}\uparrow & (\exists x.\ \mathsf{F}\ [\![\varphi]\!]) \\ \bullet & (\text{otherwise}). \end{cases}$$

The universal quantifier may be viewed as generalized conjunction: for instance, $\forall x \circledcirc \mathcal{B} \cdot \$\varphi(x)$ is equivalent to $\$\varphi(\top) \wedge \$\varphi(\bot)$. The variables bound by a binder range over well-defined values: $\forall x \cdot \mathsf{D}(x)$ is valid, and $\{x \mid \neg\,\mathsf{D}(x)\}$ is equivalent to the empty set $\varnothing$.

2.2.7. Substitutions

Event-B has three kinds of variables: type variables, ordinary variables, and operator variables. Correspondingly, I will define three kinds of substitutions. Although it may seem completely obvious how to apply ordinary and type substitutions to terms, the process involves some pitfalls; some of these pitfalls are hard to extract from the literature and others are specific to Event-B. To avoid any ambiguities, I give formal definitions of all three kinds of substitutions, referring to the literature when possible. To validate my definitions, I will analyze the impact of a substitution on the denotation of the term it is applied to; I will point out a close relationship between Event-B's substitutions and their HOL counterparts. The lemmas of this section constitute an important prerequisite for several proofs in Sections 2.3 and 3.2.

2.2.7.1. Type Substitutions

Intuitively, a type substitution $[\underline{\boldsymbol{\alpha}} := \underline{\boldsymbol{\nu}}]$ is applied to a term by simultaneously replacing α_i by ν_i, for $1 \leq i \leq |\underline{\boldsymbol{\alpha}}|$. One subtlety needs to be observed: applying a type substitution may identify variables having the same name but distinct types. That gives rise to the question: what is the result of applying $[\alpha := \mathcal{B}]$ to $\forall x \circledcirc \alpha \cdot x \circledcirc \mathcal{B}$? After all that has been said it could be either $\forall y \circledcirc \mathcal{B} \cdot x \circledcirc \mathcal{B}$ or $\forall x \circledcirc \mathcal{B} \cdot x \circledcirc \mathcal{B}$. The following formal definition dispels any doubts:

Definition 2.4. Let σ be a type substitution. The result of applying σ to a term is defined by the following recursive equations:

1. $(x \circledcirc \nu)\sigma = x \circledcirc (\nu\sigma)$,
2. $(f(\underline{\boldsymbol{t}}) \circledcirc \mu)\sigma = f(\underline{\boldsymbol{t}}\sigma) \circledcirc (\mu\sigma)$,

3. $(\$f(\underline{\boldsymbol{t}}) \wr \mu)\sigma = \$f(\underline{\boldsymbol{t}}\sigma) \wr (\mu\sigma)$,

4. $((Q\underline{\boldsymbol{x}} \cdot \underline{\boldsymbol{t}}) \wr \xi)\sigma = (Q\underline{\boldsymbol{x}}\sigma \cdot \underline{\boldsymbol{t}}\sigma) \wr (\xi\sigma)$.

Here, $\underline{\boldsymbol{t}}\sigma$ abbreviates $t_1\sigma, \ldots, t_{|\underline{\boldsymbol{t}}|}\sigma$. In 4, the names of the bound variables $\underline{\boldsymbol{x}}$ are chosen such that (i) different variables in $\underline{\boldsymbol{x}}$ have different names, and (ii) the names of free variables of $(Q\underline{\boldsymbol{x}} \cdot \underline{\boldsymbol{t}})$ differ from the variable names in $\underline{\boldsymbol{x}}$. □

Recall that $\forall x \wr \alpha \cdot x \wr \mathcal{B}$ and $\forall y \wr \alpha \cdot x \wr \mathcal{B}$ are identical, because they are alpha-congruent. Then, Proviso (ii) of Definition 2.4 has the effect that $(\forall x \wr \alpha \cdot x \wr \mathcal{B})[\alpha := \mathcal{B}]$ equals $(\forall y \wr \mathcal{B} \cdot x \wr \mathcal{B})$, similarly as in HOL [Gordon and Pitts, 1993]. Proviso (i) serves a similar purpose.

To gain a better understanding of type substitutions, I investigate their impact on the denotations of terms. The *denotation* $[\![[\underline{\boldsymbol{\alpha}} := \underline{\boldsymbol{\nu}}]]\!]$ of a type substitution is defined by $[\underline{\boldsymbol{\alpha}} := [\![\underline{\boldsymbol{\nu}}]\!]]$[3]. Type substitutions are closely related with their denotations:

Lemma 2.5. *If σ is a type substitution, t a term, and ν a type, then $[\![\nu\sigma]\!] = [\![\nu]\!][\![\sigma]\!]$ and $[\![t\sigma]\!] = [\![t]\!][\![\sigma]\!]$ are valid.*

Proof. It is straightforward to show validity of $[\![\nu\sigma]\!] = [\![\nu]\!][\![\sigma]\!]$.

It remains to prove $[\![u\sigma]\!] = [\![u]\!][\![\sigma]\!]$ by structural induction over u. If u is an ordinary variable $x \wr \nu$, the proof proceeds as follows:

$$\begin{aligned}[\![(x \wr \nu)\sigma]\!] &= [\![x \wr (\nu\sigma)]\!] = (x :: [\![\nu\sigma]\!])\uparrow = (x :: ([\![\nu]\!][\![\sigma]\!]))\uparrow \\ &= (x :: [\![\nu]\!])\uparrow[\![\sigma]\!] = [\![x \wr \nu]\!][\![\sigma]\!].\end{aligned}$$

The cases of operators and operator variables proceed similarly and are therefore omitted. If u is of the form $(Q\underline{\boldsymbol{x}} \cdot \underline{\boldsymbol{t}}) \wr \xi$, the proof proceeds as follows:

$$\begin{aligned}&[\![((Q\underline{\boldsymbol{x}} \wr \underline{\boldsymbol{\nu}} \cdot \underline{\boldsymbol{t}}) \wr \xi)\sigma]\!] \\ &= [\![((Q\underline{\boldsymbol{x}} \wr \underline{\boldsymbol{\nu}}\sigma \cdot \underline{\boldsymbol{t}}\sigma) \wr (\xi\sigma)]\!] \\ &= ([\![Q]\!]\ (\lambda\underline{\boldsymbol{x}} :: [\![\underline{\boldsymbol{\nu}}\sigma]\!].\ [\![t_1\sigma]\!])\ \ldots\ (\lambda\underline{\boldsymbol{x}} :: [\![\underline{\boldsymbol{\nu}}\sigma]\!].\ [\![t_{|\underline{\boldsymbol{t}}|}\sigma]\!])) :: [\![\xi\sigma]\!]\uparrow \\ &= ([\![Q]\!]\ (\lambda\underline{\boldsymbol{x}} :: ([\![\underline{\boldsymbol{\nu}}]\!][\![\sigma]\!]).\ [\![t_1]\!][\![\sigma]\!])\ \ldots\ (\lambda\underline{\boldsymbol{x}} :: ([\![\underline{\boldsymbol{\nu}}]\!][\![\sigma]\!]).\ [\![t_{|\underline{\boldsymbol{t}}|}]\!][\![\sigma]\!])) :: ([\![\xi]\!][\![\sigma]\!])\uparrow \\ &= (([\![Q]\!]\ (\lambda\underline{\boldsymbol{x}} :: [\![\underline{\boldsymbol{\nu}}]\!]).\ [\![t_1]\!])\ \ldots\ (\lambda\underline{\boldsymbol{x}} :: [\![\underline{\boldsymbol{\nu}}]\!].\ [\![t_{|\underline{\boldsymbol{t}}|}]\!])) :: ([\![\xi]\!])\uparrow)[\![\sigma]\!] \\ &= [\![(Q\underline{\boldsymbol{x}} \wr \underline{\boldsymbol{\nu}} \cdot \underline{\boldsymbol{t}}) \wr \xi]\!][\![\sigma]\!].\end{aligned}$$

The equality between the fourth and fifth line is justified by the definition of type substitutions in higher-order logic [Gordon and Pitts, 1993]. In higher-order logic, a similar proviso on the names of bound variables $\underline{\boldsymbol{x}}$ as Event-B is stipulated. The proviso in higher-order logic follows from the proviso in Event-B. □

Type variables may be seen as place-holders for types. By defining the effect of type substitutions on terms, I make precise how exactly type variables can be instantiated. Lemma 2.5 provides a semantic characterization of type substitutions σ; in particular, if φ is valid, then so is $\varphi\sigma$.

[3] I deviate from the usual notation of substitutions in higher-order logic; the substitution $[\underline{\boldsymbol{\alpha}} := \underline{\boldsymbol{\nu}}]$ is usually written as $[\nu_1/\alpha_1, \ldots, \nu_{|\underline{\boldsymbol{\nu}}|}/\alpha_{|\underline{\boldsymbol{\alpha}}|}]$ [see e.g. Gordon and Pitts, 1993].

2.2.7.2. Operator Substitutions

Definition 2.6. An *operator substitution* σ is written

$$[\$f_1(\underline{\$\boldsymbol{x}^1}) := u_1, \ldots, \$f_n(\underline{\$\boldsymbol{x}^n}) := u_n],$$

where $\$f_1, \ldots, \$f_n$ are pairwise distinct, the elements of $\underline{\$\boldsymbol{x}^i}$ are pairwise distinct, the type of $\underline{\$\boldsymbol{x}^i}$ is the argument type of $\$f_i$, and the type of u_i the result type of $\$f_i$, for $1 \leq i \leq n$. The terms $u_1, \ldots, u_n$ are the *right-hand sides* of σ. □

Before presenting a formal definition of the effect of operator substitutions on terms, let us consider several examples.

Example 2.7. First, consider the operator substitution $[\$\varphi := x]$. This substitution merely replaces $\$\varphi$ by x:

$$(\$\varphi \wedge \neg\$\varphi)[\$\varphi := x] \qquad \text{equals} \qquad (x \wedge \neg x).$$

Bound variables are renamed so that the occurrences of x introduced by $[\$\varphi := x]$ are free:

$$(\forall x \cdot \$\varphi)[\$\varphi := x] \qquad \text{equals} \qquad (\forall y \cdot x).$$

Now consider the substitution $[\$\varphi(\$y) := (\$y = 0)]$. Intuitively, it maps the operator variable $\$\varphi$ to the function mapping $\$y$ to $\$y = 0$; for example,

$$\$\varphi(1)[\$\varphi(\$y) := (\$y = 0)] \qquad \text{equals} \qquad 1 = 0.$$

The argument $\$y$ of $\$\varphi$ may also match a bound variable:

$$(\forall x \cdot \$\varphi(x))[\$\varphi(\$y) := (\$y = 0)] \qquad \text{equals} \qquad (\forall x \cdot x = 0).$$

Again, bound variables are renamed such that occurrences of ordinary variables introduced by operator substitutions become free:

$$(\forall x \cdot \$\varphi(x))[\$\varphi(\$y) := (\$y = x)] \qquad \text{equals} \qquad (\forall y \cdot y = x).$$

□

The following definition specifies the result of applying operator substitutions to terms:

Definition 2.8. Let σ be the operator substitution

$$[\$f_1(\underline{\$\boldsymbol{x}^1}) := u_1, \ldots, \$f_n(\underline{\$\boldsymbol{x}^n}) := u_n].$$

Then the result of applying σ to a term is defined recursively as follows:

1. $x\sigma = x$,
2. $f(\underline{\boldsymbol{t}})\sigma = f(\underline{\boldsymbol{t}\sigma})$,

3. $\$g(\underline{\boldsymbol{t}})\sigma = \$g(\underline{\boldsymbol{t}}\sigma)$, provided $\$g$ differs from $\$f_i$, for $1 \leq i \leq n$,

4. $\$f_i(\underline{\boldsymbol{t}})\sigma = u_i[\$x_1^i := t_1\sigma, \ldots, \$x^i{}_{|\underline{\$\boldsymbol{x}^{\boldsymbol{i}}}|} := t_{|\underline{\boldsymbol{t}}|}\sigma]$, for $1 \leq i \leq n$,

5. $(Q\underline{\boldsymbol{y}} \cdot \underline{\boldsymbol{t}})\sigma = Q\underline{\boldsymbol{y}} \cdot \underline{\boldsymbol{t}}\sigma$, provided no element of $\underline{\boldsymbol{y}}$ occurs free in $u_1, \ldots, u_n$.

Here, $\underline{\boldsymbol{t}}\sigma$ abbreviates $t_1\sigma, \ldots, t_{|\underline{\boldsymbol{t}}|}\sigma$. The proviso of 5 is achieved by suitably renaming bound variables. □

The *denotation* $[\![\sigma]\!]$ of the operator substitution σ in Definition 2.6 is

$$[\$f_1 := \lambda\underline{\$\boldsymbol{x}^{\boldsymbol{1}}}.\ [\![u_1]\!], \ldots, \$f_n := \lambda\underline{\$\boldsymbol{x}^{\boldsymbol{n}}}.\ [\![u_n]\!]].$$

With this definition of denotation, operator substitutions fulfill a duality property analog to Lemma 2.5:

Lemma 2.9. *If t is a term and σ an operator substitution, then $[\![t\sigma]\!] = [\![t]\!][\![\sigma]\!]$ is valid.*

Proof. First, note that it is straightforward to prove the assertion of the lemma for operator substitutions of the form $[\$x_1 := u_1, \ldots, \$x_n := u_n]$.

For the general case, suppose the operator substitution σ is of the form

$$[\$f_1(\underline{\$\boldsymbol{x}^{\boldsymbol{1}}}) := u_1, \ldots, \$f_n(\underline{\$\boldsymbol{x}^{\boldsymbol{n}}}) := u_n].$$

It remains to show $[\![t'\sigma]\!] = [\![t']\!][\![\sigma]\!]$ by structural induction over t'.

If t' is an ordinary variable, or takes the form $f(\underline{\boldsymbol{t}})$ or $\$g(\underline{\boldsymbol{t}})$, where $\$g$ differs from $\$f_i$, for $1 \leq i \leq n$, the proof is straightforward and therefore omitted. If t' is of the form $\$f_i(\underline{\boldsymbol{t}})$, with $1 \leq i \leq n$, the proof proceeds as follows:

$$\begin{aligned}
[\![\$f_i(\underline{\boldsymbol{t}})\sigma]\!] &= [\![u_i[\$x_1^i := t_1\sigma, \ldots, \$x^i{}_{|\underline{\$\boldsymbol{x}^{\boldsymbol{i}}}|} := t_{|\underline{\boldsymbol{t}}|}\sigma]]\!] \\
&= [\![u_i]\!][\$x_1^i := [\![t_1\sigma]\!], \ldots, \$x^i{}_{|\underline{\$\boldsymbol{x}^{\boldsymbol{i}}}|} := [\![t_{|\underline{\boldsymbol{t}}|}\sigma]\!]] \\
&= [\![u_i]\!][\$x_1^i := [\![t_1]\!][\![\sigma]\!], \ldots, \$x^i{}_{|\underline{\$\boldsymbol{x}^{\boldsymbol{i}}}|} := [\![t_{|\underline{\boldsymbol{t}}|}]\!][\![\sigma]\!]] \\
&= (\lambda\underline{\$\boldsymbol{x}^{\boldsymbol{i}}}.\ [\![u_i]\!])\ ([\![\underline{\boldsymbol{t}}]\!][\![\sigma]\!]) \\
&= (\$f_i\ [\![\underline{\boldsymbol{t}}]\!])[\![\sigma]\!] \\
&= [\![\$f_i(\underline{\boldsymbol{t}})]\!][\![\sigma]\!].
\end{aligned}$$

The step from the first to the second line is justified by the independent proof of the case that the operator variables involved in the substitution have empty argument types.

If t' is of the form $Q\underline{\boldsymbol{y}} \cdot \underline{\boldsymbol{t}}$, the proof proceeds as follows:

$$\begin{aligned}
[\![(Q\underline{\boldsymbol{y}} \cdot \underline{\boldsymbol{t}})\sigma]\!] &= [\![Q\underline{\boldsymbol{y}} \cdot \underline{\boldsymbol{t}}\sigma]\!] \\
&= [\![Q]\!]\ (\lambda\underline{\boldsymbol{y}}.\ [\![t_1\sigma]\!])\ \ldots\ (\lambda\underline{\boldsymbol{y}}.\ [\![t_{|\underline{\boldsymbol{t}}|}\sigma]\!]) \\
&= [\![Q]\!]\ (\lambda\underline{\boldsymbol{y}}.\ [\![t_1]\!][\![\sigma]\!])\ \ldots\ (\lambda\underline{\boldsymbol{y}}.\ [\![t_{|\underline{\boldsymbol{t}}|}]\!][\![\sigma]\!]) \\
&= ([\![Q]\!]\ (\lambda\underline{\boldsymbol{y}}.\ [\![t_1]\!])\ \ldots\ (\lambda\underline{\boldsymbol{y}}.\ [\![t_{|\underline{\boldsymbol{t}}|}]\!]))[\![\sigma]\!] \\
&= [\![Q\underline{\boldsymbol{y}} \cdot \underline{\boldsymbol{t}}]\!][\![\sigma]\!].
\end{aligned}$$

The equality of the third and fourth line rests on the assumption in the definition of operator substitutions that no element of $\underline{\boldsymbol{y}}$ occurs free in $u_1, \ldots, u_n$. □

2.2.7.3. Ordinary Substitutions

It is hard to define *ordinary substitutions*, i.e., substitutions replacing ordinary variables by terms, in such a way that a duality property analog to Lemma 2.5 or 2.9 holds. The problem is most easily demonstrated by an example. Suppose the ordinary substitution σ is defined such that $x\sigma$ is $\bullet$. Then $[\![x\sigma]\!]$ is $\bullet$, but there is no way of defining $[\![\sigma]\!]$ such that $[\![x]\!][\![\sigma]\!]$ equals $\bullet$, because $[\![x]\!]$ is $x{\uparrow}$. There are two ways out: (1) exclude substitutions that substitute ill-defined terms for ordinary variables, or (2) show some weaker version of Lemma 2.5 or 2.9. As the only application of ordinary substitutions in this document is to change the names of ordinary variables, I take option (1) by restricting the right-hand sides of ordinary substitutions to ordinary variables.

An *ordinary substitution* $[x := y]$ consists of two ordinary variables x, y of the same type. The result of *applying* $[x := y]$ *to a term* is obtained in two phases. First, consistently rename bound variables such that they differ from y. Second, replace all free occurrences of x by y. A detailed definition can be found, e.g., in Hindley and Seldin [2008]. After defining $[\![[x \mathbin{:8} \nu := y \mathbin{:8} \nu]]\!]$ as $[x :: [\![\nu]\!] := y :: [\![\nu]\!]]$, it is straightforward to show:

Lemma 2.10. *If t is a term and σ an ordinary substitution, then $[\![t\sigma]\!] = [\![t]\!][\![\sigma]\!]$ is valid.*

2.3. Proofs

After defining syntax and semantics, it remains to introduce a notion of proof for Event-B's logic. Since Rodin's signature and proof calculus are constantly changing, defining one fixed proof calculus for one fixed signature lacks the required flexibility. Instead, I go a step further and develop a formal language for expressing a wide variety of proof calculi. These are the main requirements:

- Flexibility: the language should cover a considerable fragment of the rules available in Rodin. Challenges include how to express non-freeness conditions in rules involving binders and how to cope with term rewriting.
- Convenience: Rodin users should be able to use it to specify proof rules conveniently. This requirement rules out programming languages like Java or ML.

In this section, types and terms are all drawn from the same signature, designated as Σ. The denotations of types and terms are specified by an arbitrary but fixed structure M over Σ. Occasionally, I make use of symbols from Σ_0 (introduced in Section 2.1.2) assuming that their denotations are given by EB_0 (introduced in Section 2.2.6).

semantics	unsound rule
WW	$\dfrac{\underline{\boldsymbol{\chi}} \vdash \psi}{\underline{\boldsymbol{\chi}}, \neg\psi \vdash \bot}\ \mathsf{not_E}$
SS	$\dfrac{\underline{\boldsymbol{\chi}}, \varphi \vdash \bot}{\underline{\boldsymbol{\chi}} \vdash \neg\varphi}\ \mathsf{not_I}$

semantics	unsound rule
WS	$\dfrac{}{\underline{\boldsymbol{\chi}}, \varphi \vdash \varphi}\ \mathsf{hyp}$
SW	$\dfrac{\underline{\boldsymbol{\chi}} \vdash \psi \qquad \underline{\boldsymbol{\chi}}, \psi \vdash \varphi}{\underline{\boldsymbol{\chi}} \vdash \varphi}\ \mathsf{cut}$

Figure 2.2.: Unsound inference rules for the various sequent semantics

2.3.1. Sequents

Following Event-B's tradition [Abrial, 2010], I organize proofs in terms of hypothetical statements, called *sequents*, which of course go back to Gentzen [1935] and Prawitz [2006]. A *sequent* $\underline{\boldsymbol{\psi}} \vdash \varphi$ consists of a finite set $\{\underline{\boldsymbol{\psi}}\}$ of formulae, called *hypotheses*, and a single formula φ, called *goal*. A *proof obligation* is a sequent that is used to express a desired property of an Event-B model. I write $(\underline{\boldsymbol{\psi}} \vdash \varphi)\sigma$ for $\underline{\boldsymbol{\psi}}\sigma \vdash \varphi\sigma$.

I consider several ways of defining denotations of sequents, all of the form

$$[\![\psi_1, \ldots, \psi_n \ \vdash\ \varphi]\!] = (\mathcal{H}\ [\![\psi_1]\!] \wedge \cdots \wedge \mathcal{H}\ [\![\psi_n]\!] \longrightarrow \mathcal{G}\ [\![\varphi]\!]),$$

where the functions $\mathcal{H}$ and $\mathcal{G}$ range over T and WT. I distinguish *WW-, WS-, SW-, and SS-semantics*, where the first letter indicates the choice of $\mathcal{H}$ and the second the choice of $\mathcal{G}$; the letter S ("strong") represents the choice T, and the letter W ("weak") represents the choice WT. A sequent is *WW-, WS-, SW-, or SS-valid* iff its denotation is valid according to WW-, WS-, SW-, or SS-semantics, respectively.

The choice of sequent semantics is not straightforward. The various sequent semantics are usually evaluated [Mehta, 2008, Owe, 1993] by constructing proof calculi for them and arguing that one of them is most "natural". Of course, the perception of "what is natural" varies from person to person.

Figure 2.2 illustrates this point. It gives for each of the four semantics an inference rule that is unsound with respect to exactly that semantics. As I will specify below, an inference rule is *sound* iff validity of the *antecedents* (the sequents above the horizontal line) implies validity of the *consequent* (the sequent below the horizontal line). Soundness of the rules in Figure 2.2 can be recovered by adding the antecedent $\underline{\boldsymbol{\chi}} \vdash \mathsf{D}(\varphi)$ or $\underline{\boldsymbol{\chi}} \vdash \mathsf{D}(\psi)$, but it may be unpleasing to discharge these additional antecedents during proofs.

It is therefore not surprising that different logics are based on different semantics: Owe [1993] carries out an extensive comparison of proof calculi; he favors WS-semantics. Mehta [2008] constructs a proof calculus for SW-semantics and convincingly argues that this proof calculus is natural. To my best knowledge, Mehta's work constitutes the main reason why Event-B is based on SW-semantics. PVS

[Owre and Shankar, 1999] is also based on SW-semantics. LPF [Barringer et al., 1984] is based on SS-semantics.

I believe that the "best" semantics cannot be determined once and for all, as the underlying quality measure depends too much on individual perception. As a novel argument in favor of SW-semantics, I will point out that directed rewriting (to be introduced in Section 2.3.2.3) requires SW-semantics, and that term rewriting in Rodin greatly benefits from directed rewriting (Section 4.1).

In this document, I adopt the convention that sequents are interpreted in SW-semantics unless indicated otherwise. In particular, the statement Γ is *valid* means that Γ is SW-valid.

2.3.2. Rules of Proofs

The subject of this section is to introduce the various kinds of rules that constitute a proof calculus.

2.3.2.1. Inference Rules

An *inference rule* is written

$$\frac{\underline{\mathbf{\Gamma}}}{\Gamma_0}\ \mathsf{r} \quad (\underline{\boldsymbol{x}} \text{ fresh}) \tag{2.1}$$

and consists of an optional *name* r, a sequence $\underline{\mathbf{\Gamma}}$ of sequents, called the *antecedents*, a single sequent Γ_0, called the *consequent*, and an optional *freshness* condition $\underline{\boldsymbol{x}}$. The variables in $\underline{\boldsymbol{x}}$ are pairwise distinct.

The above inference rule r *denotes*

$$(\forall \underline{\boldsymbol{x}} :: [\![\underline{\boldsymbol{\nu}}]\!].[\![\Gamma_1]\!] \wedge \cdots \wedge [\![\Gamma_{|\underline{\mathbf{\Gamma}}|}]\!]) \longrightarrow [\![\Gamma_0]\!],$$

where $\underline{\boldsymbol{\nu}}$ is the type of $\underline{\boldsymbol{x}}$. A rule is *sound* iff its denotation is valid. The rule r informally states: "If the elements of $\underline{\mathbf{\Gamma}}$ are true under the assumption that the elements of $\underline{\boldsymbol{x}}$ are arbitrary but fixed, then Γ_0 is true."

Roughly speaking, in a *backwards proof* a sequent Δ is proved by first choosing a rule with Δ as consequent and then proving the antecedents of the rule. This is repeated until no sequents remain to be proved. This attempt of a definition of proof bears a problem: if the only available rule is

$$\frac{}{\vdash \$x = \$x}\ ,$$

one may prove $\vdash \$x = \$x$, but one may not prove $\vdash 1 = 1$ or $\top \vdash \$x = \$x$.

To resolve this pathological situation, I develop a more sophisticated notion of proof, which has to some extent been inspired by logical frameworks [Basin and Matthews, 2002] and the foundations of Isabelle [Paulson, 1989]. I distinguish the set rules of rules that the user specifies from the set rules' of *derived rules* that may actually be used in proofs. Formally, rules' is the smallest superset of rules that

is closed under the *derivation methods* introduced in the remainder of the current section, namely *lifting*, *instantiation*, *conversion*, and *(weak) congruence*. The set of sequents *provable from* rules is the smallest set such that, if all antecedents of a rule in rules' are provable, then so is the consequent.

For inference rules, I define the derivation methods of *lifting* and *instantiation*. Note that the freshness condition $\underline{\boldsymbol{x}}$ has the effect that a derivation method does not introduce additional free occurrences of $\underline{\boldsymbol{x}}$.

Definition 2.11 (Lifting)**.** Let r be the rule

$$\frac{\underline{\boldsymbol{\psi}}_1 \vdash \varphi_1 \quad \dots \quad \underline{\boldsymbol{\psi}}_n \vdash \varphi_n}{\underline{\boldsymbol{\psi}}_0 \vdash \varphi_0}\,\mathsf{r} \quad (\underline{\boldsymbol{x}} \text{ fresh})$$

and $\underline{\boldsymbol{\chi}}$ a sequence of formulae such that the variables in $\underline{\boldsymbol{x}}$ do not occur free in any element of $\underline{\boldsymbol{\chi}}$. Then the result of *lifting* r *over* $\underline{\boldsymbol{\chi}}$ is

$$\frac{\underline{\boldsymbol{\chi}}, \underline{\boldsymbol{\psi}}_1 \vdash \varphi_1 \quad \dots \quad \underline{\boldsymbol{\chi}}, \underline{\boldsymbol{\psi}}_n \vdash \varphi_n}{\underline{\boldsymbol{\chi}}, \underline{\boldsymbol{\psi}}_0 \vdash \varphi_0} \quad (\underline{\boldsymbol{x}} \text{ fresh}).$$

□

Definition 2.12 (Instantiation)**.** Let r be the rule

$$\frac{\underline{\boldsymbol{\Gamma}}}{\Gamma_0}\,\mathsf{r} \quad (\underline{\boldsymbol{x}} \text{ fresh})$$

and σ a substitution. If σ is a type or ordinary substitution, suppose that

1. the elements of $\underline{\boldsymbol{x}}\sigma$ are pairwise distinct and
2. if z occurs free in $\underline{\boldsymbol{\Gamma}}$ and $z\sigma$ occurs in $\underline{\boldsymbol{x}}\sigma$, then z occurs in $\underline{\boldsymbol{x}}$.

If σ is an operator substitution, suppose that no variable in $\underline{\boldsymbol{x}}$ occurs free in a right-hand side of σ. Then, the result of *instantiating* r *with* σ is

$$\frac{\underline{\boldsymbol{\Gamma}}\sigma}{\Gamma_0\sigma} \quad (\underline{\boldsymbol{x}}\sigma \text{ fresh}).$$

□

To justify the above derivation methods, I show that they preserve soundness.

Lemma 2.13. *The result of lifting or instantiating a sound inference rule is sound.*

Proof. For lifting, suppose that the inference rule

$$\frac{\underline{\boldsymbol{\psi}}_1 \vdash \varphi_1 \quad \dots \quad \underline{\boldsymbol{\psi}}_n \vdash \varphi_n}{\underline{\boldsymbol{\psi}}_0 \vdash \varphi_0}\,\mathsf{r} \quad (\underline{\boldsymbol{x}} \text{ fresh}) \tag{2.2}$$

is sound. I show, without loss of generality, that the result of lifting r over one formula is sound. Let χ be a formula such that the variables in $\underline{\boldsymbol{x}}$ do not occur free in χ. Recall that the denotation of r is of the form

$$(\forall \underline{\boldsymbol{x}}.\ \Gamma_1 \wedge \cdots \wedge \Gamma_n) \longrightarrow \Gamma_0. \tag{2.3}$$

As r is sound, its denotation is valid, and hence

$$(\mathsf{T}\ [\![\chi]\!] \longrightarrow (\forall \underline{\boldsymbol{x}}.\ \Gamma_1 \wedge \cdots \wedge \Gamma_n)) \longrightarrow \mathsf{T}\ [\![\chi]\!] \longrightarrow \Gamma_0$$

is valid as well. As no variable in $\underline{\boldsymbol{x}}$ is free in χ, no variable in $\underline{\boldsymbol{x}}$ is free in $\mathsf{T}\ [\![\chi]\!]$, and therefore

$$(\forall \underline{\boldsymbol{x}}.\ (\mathsf{T}\ [\![\chi]\!] \longrightarrow \Gamma_1) \wedge \cdots \wedge (\mathsf{T}\ [\![\chi]\!] \longrightarrow \Gamma_n)) \longrightarrow \mathsf{T}\ [\![\chi]\!] \longrightarrow \Gamma_0 \tag{2.4}$$

is valid. Now observe that (2.4) is equivalent to the denotation of the result of lifting r over χ, which is therefore sound.

For instantiation, again suppose that r in (2.2) is sound. The denotation of r is of the form

$$(\forall \underline{\boldsymbol{x}}.\ \Gamma) \longrightarrow \Gamma_0.$$

Given a substitution σ meeting the side-conditions of instantiation,

$$(\forall \underline{\boldsymbol{x}}.\ \Gamma)[\![\sigma]\!] \longrightarrow (\Gamma_0[\![\sigma]\!]) \tag{2.5}$$

is valid, because validity is closed under substitution. Moreover, by Lemmas 2.5, 2.9, and 2.10, the result of instantiating r with σ denotes

$$(\forall \underline{\boldsymbol{x}}[\![\sigma]\!].\ \Gamma[\![\sigma]\!]) \longrightarrow (\Gamma_0[\![\sigma]\!]). \tag{2.6}$$

Using the side-conditions of instantiation, it can be shown by structural induction over Γ that $(\forall \underline{\boldsymbol{x}}.\ \Gamma)[\![\sigma]\!]$ is equivalent to $\forall \underline{\boldsymbol{x}}[\![\sigma]\!].\ \Gamma[\![\sigma]\!]$. Therefore, validity of (2.5) implies validity of (2.6). □

2.3.2.2. Symmetric Rewrite Rules

A *symmetric rewrite rule* consists of an optional *name* r, a *condition* $\varphi \wr \mathcal{B}$, a *left-hand side* $t \wr \nu$, and a *right-hand side* $u \wr \nu$, and is written

$$\frac{\varphi \wr \mathcal{B}}{t \wr \nu \ \equiv\ u \wr \nu}\ \mathsf{r}. \tag{2.7}$$

An *unconditional* symmetric rewrite rule has the condition $\top$ and is written $t \wr \nu \equiv u \wr \nu$[4]. The rule in (2.7) is *sound* iff $\mathsf{WT}\ [\![\varphi]\!] \longrightarrow [\![t]\!] = [\![u]\!]$ is valid. The choice to

[4] If the constant $\top$ is not available or has a non-standard denotation, this definition does not make sense. To cope with such situations, one could treat unconditional symmetric rewrite rules as a separate kind of rule with its own derivation methods. Since the details are lengthy and uninteresting, I do not take this approach.

interpret φ as $\mathsf{WT}\ [\![\varphi]\!]$ instead of $\mathsf{T}\ [\![\varphi]\!]$ is somewhat arbitrary and mostly influenced by the rules I have seen. Note that $\mathsf{T}\ [\![\varphi]\!]$ can always be encoded as $\mathsf{WT}\ [\![\mathsf{D}(\varphi) \wedge \varphi]\!]$.

Intuitively, the symmetric rewrite rule in (2.7) may be used to replace $t\sigma$ by $u\sigma$ during a proof, provided $\varphi\sigma$ is weakly true. This is formalized by the following derivation methods:

Definition 2.14 (Conversion)**.** *Converting*

$$\frac{\chi}{\psi_1 \ \equiv \ \psi_2}$$

into inference rules yields

$$\frac{\vdash \chi \qquad \vdash \psi_2}{\vdash \psi_1} \qquad \text{and} \qquad \frac{\vdash \chi \qquad \psi_2 \vdash \varphi}{\psi_1 \vdash \varphi}\ .$$

□

Definition 2.15 (Instantiation)**.** Let

$$\frac{\varphi}{t \ \equiv \ u}\ \mathsf{r}$$

be a symmetric rewrite rule and σ a substitution. The result of *instantiating* r *with* σ is

$$\frac{\varphi\sigma}{t\sigma \ \equiv \ u\sigma}\ .$$

□

Definition 2.16 (Congruence)**.** Let

$$\frac{\varphi}{t_1 \ \equiv \ t_2}$$

be a symmetric rewrite rule. Applying the *congruence method* yields

$$\frac{\varphi}{f(\underline{\boldsymbol{u}}, t_1, \underline{\boldsymbol{u}}') \ \equiv \ f(\underline{\boldsymbol{u}}, t_2, \underline{\boldsymbol{u}}')}\ , \qquad \frac{\varphi}{\$f(\underline{\boldsymbol{u}}, t_1, \underline{\boldsymbol{u}}') \ \equiv \ \$f(\underline{\boldsymbol{u}}, t_2, \underline{\boldsymbol{u}}')}\ ,$$

$$\frac{\forall \underline{\boldsymbol{x}} \cdot \varphi}{Q\underline{\boldsymbol{x}} \cdot \underline{\boldsymbol{u}}, t_1, \underline{\boldsymbol{u}}' \ \equiv \ Q\underline{\boldsymbol{x}} \cdot \underline{\boldsymbol{u}}, t_2, \underline{\boldsymbol{u}}'}\ .$$

□

The congruence method only makes sense if the universal quantifier is available and has suitable semantics. It is possible to describe symmetric rewriting for signatures without the universal quantifier, but at the cost of a more complicated presentation.

Again, the above derivation methods preserve soundness.

Lemma 2.17. *If a symmetric rewrite rule is sound, then so are the results of conversion, instantiation, and congruence.*

Proof. The case of conversion is a consequence of the various definitions of soundness. The case of instantiation immediately follows from the fact that validity in higher-order logic is closed under substitutions and from Lemmas 2.5, 2.9, and 2.10.

For the case of congruence, assume that

$$\frac{\varphi}{t \;\equiv\; u} \tag{2.8}$$

is sound. I only prove validity of

$$\frac{\forall \underline{\boldsymbol{x}} \cdot \varphi}{Q\underline{\boldsymbol{x}} \cdot u, t_1, u' \;\equiv\; Q\underline{\boldsymbol{x}} \cdot u, t_2, u'}\,. \tag{2.9}$$

The other validity proofs are similar and simpler. From the soundness of (2.8), I conclude that

$$\forall \underline{\boldsymbol{x}}.\ \mathsf{WT}\ [\![\varphi]\!] \longrightarrow [\![t_1]\!] = [\![t_2]\!] \tag{2.10}$$

is true. To prove validity of (2.9), we assume that $\forall \underline{\boldsymbol{x}}.\ \mathsf{WT}\ [\![\varphi]\!]$ is true. With (2.10) that leads to the conclusion

$$\forall \underline{\boldsymbol{x}}.\ [\![t_1]\!] = [\![t_2]\!],$$

which implies

$$[\![Q\underline{\boldsymbol{x}} \cdot u, t_1, u']\!] = [\![Q\underline{\boldsymbol{x}} \cdot u, t_2, u']\!].$$

□

2.3.2.3. Directed Rewrite Rules

Surprisingly, symmetric rewriting does not explain soundness of term rewriting in Rodin, because in Rodin it is possible to rewrite terms to terms that are not equivalent. So Rodin implements several rewrite rules that are unsound if they are viewed as symmetric rewrite rules. Examples include

$$\$x = \$x \equiv \top, \tag{2.11}$$

$$\$x \in \$R \cap \$S \equiv \$x \in \$R \wedge \$x \in \$S, \tag{2.12}$$

$$\$x \in \varnothing \equiv \bot, \tag{2.13}$$

$$0 \in \{x \mid \$\varphi(x)\} \equiv \$\varphi(0), \tag{2.14}$$

$$\$x \bmod \$x \equiv 0. \tag{2.15}$$

The unsoundness of (2.11–2.14) stems from the special status of ill-definedness: (2.11) is unsound because equality is strict and therefore not reflexive, (2.12) because intersection is strict and conjunction is lazy, (2.13) because membership is strict, and (2.14) because set comprehension is smashed. The last rule (2.15) is unsound because it lacks a check that $\$x$ is greater than zero.

The fact that the above rules (2.11–2.15) are unsound does not imply that term rewriting in Rodin is unsound. It merely indicates that symmetric rewriting is an inappropriate explanation. To explain why term rewriting in Rodin is sound, I introduce the notion of *directed rewriting*.

A *directed rewrite rule* consists of an optional *name* r, a *condition* $\varphi \mathbin{8} \mathcal{B}$, a *left-hand side* $t \mathbin{8} \nu$, and a *right-hand side* $u \mathbin{8} \nu$, and is written

$$\frac{\varphi \mathbin{8} \mathcal{B}}{t \mathbin{8} \nu \;\sqsubseteq\; u \mathbin{8} \nu}\ \mathsf{r}. \tag{2.16}$$

An *unconditional* directed rewrite rule has the condition $\top$ and is written $t \mathbin{8} \nu \sqsubseteq u \mathbin{8} \nu$. The rule in (2.16) is *sound* iff $\mathsf{WT}\ [\![\varphi]\!] \longrightarrow [\![t]\!] \sqsubseteq [\![u]\!]$ is valid.

Conversion and instantiation for directed rewriting are similar as for symmetric rewriting:

Definition 2.18 (Conversion)**.** *Converting*

$$\frac{\chi}{\psi_1 \;\sqsubseteq\; \psi_2}$$

to inference rules yields

$$\frac{\vdash \chi \qquad \vdash \psi_2}{\vdash \psi_1} \qquad \text{and} \qquad \frac{\vdash \chi \qquad \psi_2 \vdash \varphi}{\psi_1 \vdash \varphi}\ .$$

□

Definition 2.19 (Instantiation)**.** Let

$$\frac{\varphi}{t \;\sqsubseteq\; u}\ \mathsf{r}$$

be a symmetric rewrite rule and σ a substitution. The result of *instantiating* r *with* σ is

$$\frac{\varphi\sigma}{t\sigma \;\sqsubseteq\; u\sigma}\ .$$

□

The main difference lies in the congruence method:

Definition 2.20 (Weak Congruence)**.** Let

$$\frac{\varphi}{t_1 \;\sqsubseteq\; t_2}$$

be a symmetric rewrite rule and assume that f and Q are monotonic. Applying the *congruence method* yields

$$\frac{\varphi}{f(\underline{\boldsymbol{u}}, t_1, \underline{\boldsymbol{u}}') \sqsubseteq f(\underline{\boldsymbol{u}}, t_2, \underline{\boldsymbol{u}}')}\,,$$
$$\frac{\forall \underline{\boldsymbol{x}} \cdot \varphi}{Q\underline{\boldsymbol{x}} \cdot \underline{\boldsymbol{u}}, t_1, \underline{\boldsymbol{u}}' \sqsubseteq Q\underline{\boldsymbol{x}} \cdot \underline{\boldsymbol{u}}, t_2, \underline{\boldsymbol{u}}'}\,.$$

□

The main restriction of directed rewriting is that only the arguments of monotonic operators and binders may be rewritten. In the context of Rodin this is no restriction at all, because all operators and binders available in Rodin are monotonic, and operator variables are not supported.

The derivation methods for directed rewrite rules preserve soundness. The proof is similar to the proof for symmetric rewriting. The case of conversion relies on the fact that sequents have SW-semantics; in fact, SW-semantics is the only one of the four considered semantics under which conversion preserves soundness.

Lemma 2.21. *If a directed rewrite rule is sound, then so are the results of conversion, instantiation, and congruence.*

There is more to be said about directed rewriting. In Section 4.1, I analyze the practical impact of directed rewriting as a proof strategy. In Section 4.2, I describe the effect of various design decisions, including monotonicity and SW-semantics, on Event-B's logic.

2.3.2.4. Expressiveness and Limitations

It is possible to conveniently formalize a wide variety of rules as inference rules, and symmetric and directed rewrite rules. In Chapter 3, I will give numerous examples. At this point, I comment on the kinds of rules that are available in Rodin but cannot be formalized or are difficult to formalize with the machinery developed in the preceding sections.

Associative-Commutative Operators. For simplicity, the congruence and instantiation methods do not take associativity or commutativity into account. It is thus impossible to formalize rewrite rules like

$$\$\varphi_1 \land \cdots \land \$\varphi_n \Rightarrow \$\varphi_i \sqsubseteq \top \qquad (1 \leq i \leq n).$$

It would however not be difficult to extend my formalism to cover rules involving associative-commutative operators.

Binders Binding an Arbitrary Number of Variables. There is currently no way of expressing

$$\frac{\mathsf{D}(\$y)}{\forall x_1 \cdot \ldots \forall x_n \cdot x_i = \$y \Rightarrow \$\varphi(x_i) \;\sqsubseteq\; \$\varphi(\$y)} \qquad (1 \leq i \leq n)$$

because of the arbitrary number of variables bound by the universal quantifier. Such rules still need to be encoded in the underlying programming language.

Type Expressions. The informal descriptions of some rules implemented by Rodin have constraints like "$\$R$ is a type expression". As an example, consider

$$\$x \in \$R \sqsubseteq \top \qquad (\$R \text{ is a type expression}). \tag{2.17}$$

This side-condition has the effect that $\$R$ is matched against terms denoting the lifted *universe* $\mathsf{UNIV}\uparrow$, where UNIV is defined by $\mathsf{UNIV} = \{x.\ \mathsf{True}\}$. Of course, the question whether a term denotes the universe is undecidable; therefore $\$R$ is matched against so-called "type expressions", i.e., terms that are built from a restricted set of constants and are therefore known to denote the lifted universe. Type expressions include terms built from $\mathbb{Z}$ and $\mathbb{P}$.

Even though the notion of type expression is not available, a fairly similar effect can be achieved with the machinery developed so far, provided the constant $\mathsf{UNIV} \mathbin{⦂} \mathcal{P}(\alpha)$ (denoting $\mathsf{UNIV}\uparrow$) is available. For example, the rule (2.17) is represented as

$$\$x \in \mathsf{UNIV} \sqsubseteq \top.$$

This rule already matches terms of the form $t \in \mathsf{UNIV}$, but it does not match terms of the form $t \in \mathbb{Z}$ or $t \in \mathbb{P}(\mathsf{UNIV})$. To cope with such terms, introduce the rules

$$\mathbb{Z} \equiv \mathsf{UNIV},$$
$$\mathbb{P}(\mathsf{UNIV}) \equiv \mathsf{UNIV}.$$

Enumerated Sets and Numerals. It is difficult to formalize rules involving enumerated sets like the following:

$$\$x \in \{\$y_1, \ldots, \$y_n\} \sqsubseteq \$x = \$y_1 \vee \cdots \vee \$x = \$y_n. \tag{2.18}$$

The difficulty stems from the fact that the enumerated set operator takes an arbitrary number of arguments.

In HOL, this problem is avoided by a clever representation of enumerated sets. HOL provides an operator $\mathsf{insert} :: \alpha \Rightarrow \alpha\ \mathsf{set} \Rightarrow \alpha\ \mathsf{set}$ such that

$$\mathsf{insert}\ x\ R = \{y.\ y = x \vee y \in R\}.$$

The enumerated set $\{t_1, \ldots, t_n\}$ is internally represented by

$$\mathsf{insert}\ t_1\ (\ldots\ (\mathsf{insert}\ t_n\ \varnothing)\ldots).$$

A rule analog to (2.18) is represented by

$$(x \in \mathsf{insert}\ y\ R) = (x = y \vee x \in R).$$

In Event-B, it is also difficult to formalize rules involving *numerals*, such as 0, 1, 2, because numerals are implemented as primitive constants. Consider for example the rule

$$i + j \equiv \text{“the numeral equal to } i + j\text{”} \qquad (i, j \text{ are numerals}).$$

Similarly as for enumerated sets, a rule corresponding to the above can be expressed in HOL (without reverting to ML tactics) thanks to a clever representation of numerals. The details can be found in [Cohen and Watson, 1991] and in the HOL theory sources, in particular the sources of Int.

My conclusion is that there are limitations when expressing rules involving enumerated sets and numerals. To overcome these limitations, I would choose more appropriate representations.

2.3.3. Theories

A *theory* $\mathsf{THY} = (\Sigma, \mathsf{rules})$ consists of a signature Σ and a set of rules rules over Σ. A sequent over Σ is *provable* in THY iff it is provable from rules. The theory THY is *consistent* iff some sequent over Σ is unprovable in THY. A structure M over Σ is a *model of* THY, written as $\mathsf{M} \models \mathsf{THY}$, iff all rules in rules are sound with respect to M.

Proposition 2.22 (Soundness)**.** *If* M *is a model of the theory* THY*, then every sequent provable in* THY *is valid with respect to* M*.*

Proof. Suppose M is a model of the theory $\mathsf{THY} = (\Sigma, \mathsf{rules})$. By the definition of model and Lemmas 2.13, 2.17, and 2.21, all rules in rules' are sound with respect to M.

I prove by induction that every sequent provable in THY is valid with respect to M. Let

$$\frac{\underline{\boldsymbol{\Gamma}}}{\Gamma_0}\ \mathsf{r} \quad (\underline{\boldsymbol{x}} \text{ fresh})$$

be an inference rule from rules'. By the induction hypothesis, all antecedents $\underline{\boldsymbol{\Gamma}}$ of r are valid. Hence, $\forall \underline{\boldsymbol{x}}.\ [\![\Gamma_1]\!] \wedge \cdots \wedge [\![\Gamma_{|\underline{\boldsymbol{\Gamma}}|}]\!]$ is valid, too. It remains to show that the consequent Γ_0 of r is valid, which follows with the fact that r is sound. □

Extending Theories. *Extending* a theory essentially means to introduce new symbols and rules. Formally, a theory $(\Sigma', \mathsf{rules}')$ *extends* a theory (Σ, rules) iff Σ' extends Σ and rules is a subset of rules'.

Merging Theories. *Merging* two theories THY_1 and THY_2 informally means to build a theory that contains the symbols and rules of THY_1 and THY_2. Formally, two signatures Σ_1 and Σ_2 are *compatible* iff symbols available in both Σ_1 and Σ_2 have the same kind (i.e., type operator, operator, or binder) and are assigned the same arities, bound variable, argument, and result types. *Merging* two compatible signatures Σ_1 and Σ_2 results in the signature $\Sigma_1 \cup \Sigma_2$ that contains exactly the symbols in Σ_1 and Σ_2 and is compatible with Σ_1 and Σ_2. Two theories are *compatible* iff the underlying signatures are. The result of *merging* two theories THY_1 and THY_2 is written $\mathsf{THY}_1 \cup \mathsf{THY}_2$ and obtained by merging the underlying signatures and taking the union of the underlying rule sets.

2.4. Comparison to Rodin's Implementation

My preceding elaboration of Event-B's logic extends in several ways the version of Event-B supported by Rodin. Rodin gives the booleans the status of a separate syntactic category; I revoke this special status of the booleans and treat them in the same way as any other type. I introduce operator variables and define symmetric rewriting, which do not have counterparts in Rodin. In the following, I shall explain my motivations for these extensions.

Boolean Type. In more traditional texts such as by Abrial [2010], and Metayer and Voisin [2009], the logic of Event-B is presented like a first-order logic with separate syntactic categories of "expressions" and "predicates". Intuitively, expressions stand for terms and predicates for formulae. Some operators (including membership $\in$) take only expressions as arguments; others (including conjunction $\wedge$) take only predicates. In particular, Rodin considers sets of predicates such as $\{\top\}$ as syntactically ill-formed.

At some point, it has been recognized that this way of treating predicates is too restrictive for some applications. Therefore, expressions of boolean type[5] were introduced, including the expressions TRUE and FALSE. The singleton set containing the boolean value "true" could henceforth be expressed as $\{\mathsf{TRUE}\}$. To enable the conversion from predicates to boolean expressions, the operator bool was introduced, which takes a predicate and yields the corresponding boolean expression: $\mathsf{bool}(\top)$ is equivalent to TRUE and $\mathsf{bool}(\bot)$ equivalent to FALSE. A conversion from boolean expressions to predicates is also available: a boolean expression t corresponds to the predicate $t = \mathsf{TRUE}$. Thanks to these conversions, Rodin's distinction between expressions and predicates has become meaningless.

Operator Variables. Earlier Rodin versions only supported a fixed set of rules, which were (and still are) hard-coded in Java. There was no need for operator variables, because rules were described in Java and their soundness was proved on paper or in the head of the developer. More recent Rodin versions allow the user to

[5] I thank Laurent Voisin for explaining the history of boolean expressions to me.

specify new rules and reason about their soundness [Maamria and Butler, 2010]; yet, Rodin does not support operator variables, which leads to several problems.

Obviously, rules with operator variables of non-empty argument type such as

$$\frac{\mathsf{D}(\$y)}{\forall x \cdot x = \$y \Rightarrow \$\varphi(x) \;\sqsubseteq\; \$\varphi(\$y)}$$

are hard to express without operator variables. Consequently, Rodin does not permit the user to specify such rules.

But even when the operator variables in a rule have empty argument types, the missing support for operator variables causes problems. I will illustrate this point with the example of directed rewriting; similar problems arise with specifying and reasoning about inference rules.

In Rodin, operators with empty argument types are usually replaced by ordinary variables. For example, the (sound) rule

$$\$x = \$x \;\sqsubseteq\; \top \tag{2.19}$$

is represented by

$$x = x \;\sqsubseteq\; \top. \tag{2.20}$$

Rodin generates the soundness proof obligation $\mathsf{D}(x = x) \vdash \mathsf{D}(\top) \wedge (x = x) = \top$, which happens to be valid. The general proceeding is: (1) replace operator variables by ordinary variables, (2) generate the soundness proof obligation by noting that $t \sqsubseteq u$ is sound iff $\mathsf{D}(t) \vdash \mathsf{D}(u) \wedge t = u$ is valid.

The problem with this proceeding is that validity of the soundness proof obligation does in general not imply soundness of the original rule. As an example, consider the unsound rule

$$\bot \wedge \$x = \$x \;\sqsubseteq\; \neg(\$x = \$x) \tag{2.21}$$

with the valid soundness proof obligation

$$\mathsf{D}(\bot \wedge x = x) \;\vdash\; \mathsf{D}(\neg(x = x)) \wedge (\bot \wedge x = x) = (\neg(x = x)).$$

When the first versions of Rodin's extensible term rewriter were developed, this problem was unknown, and users could therefore inadvertently introduce unsound rules like (2.21). In later versions, the problem was solved by imposing several syntactic restrictions on rules [Maamria and Butler, 2010]; in particular, directed rewrite rules with lazy operators or non-smashed binders (such as $\wedge, \vee, \Rightarrow, \forall, \exists$) on the left-hand side are no longer supported.

On the one hand, Rodin's missing support for operator variables has led to a complicated design (operator variables are "simulated" by ordinary variables), soundness bugs, and undesirable restrictions (restricted support for lazy operators and non-smashed binders). On the other hand, I admit that it would be a major undertaking to implement support for reasoning about operator variables in Rodin. I therefore pursue the following approach: Within this document, I assume that operator variables are available, because it leads to a simpler presentation. To make my results

applicable to Rodin (without a major reengineering), I point out how to translate proof obligations with operator variables to equivalent proof obligations without (cf. Section 3.2.4).

Symmetric Rewriting. Rodin does not support symmetric rewrite rules: a symmetric rewrite rule is viewed as the corresponding directed rewrite rule. Symmetric and directed rewriting behave differently if and only if non-monotonic symbols (operators, binders, or operator variables) are available. As long as Rodin only supports monotonic symbols, there is no need to implement symmetric rewriting.

3. Event-B's Theories

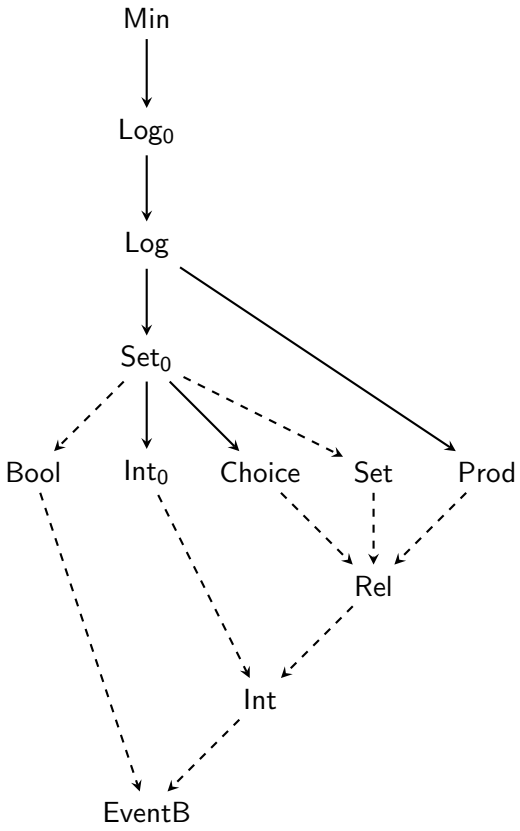
The main purpose of this chapter is to provide a comprehensive overview of the symbols (i.e., type operators, operators, and binders) available in Rodin and give them a semantics. This involves defining a signature and a structure, which I refer to as the *standard* signature and *standard* structure for the moment. A challenge is to provide convincing arguments that the standard structure is "right", in the sense that it should be used to validate Rodin's theorem provers. As there is no written agreement about requirements on Event-B's logic, the reader has to decide himself whether this is the case. To help the reader form his opinion, I provide the following information:

- I develop a theory of which the standard structure is a model. The rules of this theory illustrate important semantic properties to readers not familiar with HOL.
- I compare my semantics with the semantics underlying other texts about Event-B's logic, namely the texts by Mehta [2008], Metayer and Voisin [2009], and Abrial [2010]. See Section 3.4.

In a first step, I define a theory Core with a corresponding standard model covering a fraction of the symbols available in Rodin (Section 3.1). Next, I develop methods for extending theories with new operators, binders, and rules (Section 3.2); these methods are *definitional*, which informally means that the extended theory does not assert new axioms. In Section 3.2.4, I point out how to implement these theory extension methods in a setting without operator variables, which is helpful for their implementation in Rodin. Finally (Section 3.3), I construct Event-B's main theory EventB as a definitional extension of Core; EventB covers all symbols available in Rodin and possesses a model that definitionally extends the standard model of Core. Figure 3.1 gives an outline of Event-B's theories.

Some readers may consider it as excessive to first define definitional theory extension methods and then use them to construct EventB. Instead, one could have defined a model for EventB directly in HOL. The work on definitional extensions has been motivated by the recent implementation of theory extension methods in Rodin [Butler and Maamria, 2010]. Although these extension methods are intended to be definitional, at time of writing this goal has not been achieved yet [Schmalz, 2012]; an important reason for this is a poor understanding of the underlying logic. With my work, I point out how the semantics of Section 2.2 can be used to prove that a representative set of theory extension methods are definitional.

To improve the presentation, I take the liberty to introduce operators and binders that are not available in Rodin. The first occurrences of such symbols are marked



The theory Core – missing in the diagram to simplify the layout – results from merging Int_0, Choice, and Prod. Arrows indicate theory extensions. Dashed arrows indicate definitional extensions; theories with several incoming dashed arrows, such as Rel, definitionally extend the result of merging the theories from which the incoming arrows originate, e.g., Rel definitionally extends $\mathsf{Choice} \cup \mathsf{Set} \cup \mathsf{Prod}$.

Figure 3.1.: Event-B's theories

with the superscript †. The models of the theories developed in this chapter are available at [Schmalz].

3.1. The Theory Core

During the development of the theory Core, it quickly became clear that there is a tradeoff between the number of provided symbols and the simplicity of the corresponding rules. The main purpose of the following proof calculus is to communicate the main properties of the underlying structure to readers not familiar with HOL. I have therefore sacrificed minimality for the benefit of clarity. In particular, I have decided against deriving the integers from an abstract axiom of infinity [as in Andrews, 2002], because I doubt that such an approach would help to clarify the semantics of Event-B's integers. As a result of these considerations, Core covers first-order logic, sets, the axiom of choice, Cartesian products, and the integers.

3.1.1. The Theory Min

The theory Min introduces rules that illustrate essential properties of sequents and well-definedness. Its signature contains the boolean type $\mathcal{B}$, the *ill-definedness* constant $\bullet^\dagger \wr \alpha$, and the *well-definedness* operator $\mathsf{D}^\dagger \wr \alpha \rightharpoonup \mathcal{B}$. The rules of Min are presented in two blocks; the first block is as follows:

$$\frac{}{\$\varphi \vdash \$\varphi}\ \mathsf{hyp} \qquad \frac{\vdash \$\varphi}{\$\psi \vdash \$\varphi}\ \mathsf{mon}$$

$$\frac{\vdash \mathsf{D}(\$\chi) \qquad \vdash \$\chi \qquad \$\chi \vdash \$\varphi}{\vdash \$\varphi}\ \mathsf{cut}$$

$$\frac{\$\chi, \mathsf{D}(\$\chi) \vdash \$\varphi}{\$\chi \vdash \$\varphi}\ \mathsf{D_L} \qquad \frac{\mathsf{D}(\$\varphi) \vdash \$\varphi}{\vdash \$\varphi}\ \mathsf{D_R}$$

Note that the cut rule differs from the version of Abrial [2010] by the antecedent $\vdash \mathsf{D}(\$\chi)$. Removing this antecedent from cut would actually make the rule unsound with respect to the semantics developed below. This indicates that Abrial had different semantics in mind than the Rodin developers.

The rules in the second block indicate that *well-definedness conditions* $\mathsf{D}(t)$ and ordinary variables are well-defined, and $\bullet$ is ill-defined.

$$\frac{}{\vdash \mathsf{D}(\mathsf{D}(\$x))}\ \mathsf{DD}$$

$$\frac{}{\vdash \mathsf{D}(x)}\ \mathsf{D\_var} \qquad \frac{}{\mathsf{D}(\bullet) \vdash \$\varphi}\ \mathsf{D\_ild}$$

The Standard Model. The *standard* structure $(\mathsf{EB\_Min}, [\![\cdot]\!])$ over the signature of Min defines the denotation of $\bullet$ as $\bullet$ and the denotation of D by $[\![\mathsf{D}]\!](x) = (\mathsf{WD}\, x)\uparrow$. The theory EB_Min is a definitional extension of EB_Prelims; I omit further details on EB_Min because they are irrelevant for understanding which sequents are valid with respect to $(\mathsf{EB\_Min}, [\![\cdot]\!])$. Using Isabelle, I have proved that the rules of Min are sound with respect to the underlying standard structure.

Proposition 3.1. *The structure* $(\mathsf{EB\_Min}, [\![\cdot]\!])$ *constitutes a model of* Min.

3.1.2. The Theory $\mathsf{Log_0}$

The theory $\mathsf{Log_0}$ extends Min by introducing basic concepts of first-order logic with equality. The additional operators and binders of $\mathsf{Log_0}$ are

- $= \wr (\alpha, \alpha) \rightharpoonup \mathcal{B}$ (*equality*),
- $\wedge \wr (\mathcal{B}, \mathcal{B}) \rightharpoonup \mathcal{B}$ (*conjunction*),
- $\mathsf{cond}^\dagger \wr (\mathcal{B}, \alpha, \alpha) \rightharpoonup \alpha$ (*conditional*),
- $\forall \wr (\alpha \rightharpoonup \mathcal{B}) \rightharpoonup \mathcal{B}$ (*universal quantifier*).

The conditional is not accessible through Rodin's concrete syntax, but it has an internal representation. I usually write $\mathsf{if}\ \varphi\ \mathsf{then}\ t\ \mathsf{else}\ u$ for conditional terms $\mathsf{cond}(\varphi, t, u)$. The formula $\forall \underline{\boldsymbol{x}} \cdot \varphi$ abbreviates $\forall x_1 \cdot \ldots \forall x_{|\underline{\boldsymbol{x}}|} \cdot \varphi$.

The theory $\mathsf{Log_0}$ adds the following rules to Min:

$$\mathsf{D}(\$x = \$y) \;\equiv\; \mathsf{D}(\$x) \wedge \mathsf{D}(\$y) \quad \mathsf{eq_D}$$

$$\frac{}{\vdash\ \$x = \$x}\ \mathsf{refl} \qquad \frac{\$x = \$y\ \vdash\ \$\varphi(\$y)}{\$x = \$y\ \vdash\ \$\varphi(\$x)}\ \mathsf{subst}$$

$$\frac{\mathsf{D}(\$x) = \mathsf{D}(\bullet)}{\$x\ \equiv\ \bullet}\ \mathsf{ild\_unique} \qquad \frac{\$\varphi\ \vdash\ \$\psi \qquad \$\psi\ \vdash\ \$\varphi}{\vdash\ \$\varphi = \$\psi}\ \mathsf{bool\_extn}$$

$$\frac{\$\psi_1, \$\psi_2\ \vdash\ \$\varphi}{\$\psi_1 \wedge \$\psi_2\ \vdash\ \$\varphi}\ \mathsf{conj_L} \qquad \frac{\vdash\ \$\varphi_1 \qquad \vdash\ \$\varphi_2}{\vdash\ \$\varphi_1 \wedge \$\varphi_2}\ \mathsf{conj_R}$$

$$\mathsf{D}(\mathsf{if}\ \$\varphi\ \mathsf{then}\ \$x\ \mathsf{else}\ \$y) \;\equiv\; \mathsf{D}(\$\varphi) \wedge (\mathsf{if}\ \$\varphi\ \mathsf{then}\ \mathsf{D}(\$x)\ \mathsf{else}\ \mathsf{D}(\$y)) \quad \mathsf{cond_D}$$

$$\frac{\vdash\ \mathsf{D}(\$y) \qquad \forall x \cdot \$\psi(x), \$\psi(\$y)\ \vdash\ \$\varphi}{\forall x \cdot \$\psi(x)\ \vdash\ \$\varphi}\ \mathsf{all_L} \qquad \frac{\vdash\ \$\varphi(x)}{\vdash\ \forall x \cdot \$\varphi(x)}\ \mathsf{all_R} \quad (x \text{ fresh})$$

The above rules neither allow one to prove interesting sequents involving conditionals nor do they reveal the domains of conjunction and the universal quantifier. I introduce suitable rules with the theory Log below, because these rules can be stated more conveniently after introducing additional operators.

The Standard Model. The *standard* structure $(\mathsf{EB\_Log0}, [\![\cdot]\!])$ over the signature of $\mathsf{Log_0}$ definitionally extends the standard model of Min, and defines the denotations of additional operators and binders as follows. Equality denotes the strict extension of HOL's equality. The denotations of conjunction and conditional are defined by

$$[\![\wedge]\!]\ \varphi\ \psi = \begin{cases} \mathsf{True}\uparrow & (\mathsf{T}\ \varphi \wedge \mathsf{T}\ \psi) \\ \mathsf{False}\uparrow & (\mathsf{F}\ \varphi \vee \mathsf{F}\ \psi) \\ \bullet & (\text{otherwise}), \end{cases} \qquad [\![\mathsf{cond}]\!]\ \varphi\ x\ y = \begin{cases} x & (\mathsf{T}\ \varphi) \\ y & (\mathsf{F}\ \varphi) \\ \bullet & (\neg(\mathsf{WD}\ \varphi)). \end{cases}$$

The universal quantifier denotes

$$[\![\forall x \cdot \varphi]\!] = \begin{cases} \mathsf{True}\uparrow & (\forall x.\ \mathsf{T}\ [\![\varphi]\!]) \\ \mathsf{False}\uparrow & (\exists x.\ \mathsf{F}\ [\![\varphi]\!]) \\ \bullet & (\text{otherwise}). \end{cases}$$

I have used Isabelle to prove that the rules of $\mathsf{Log_0}$ are sound with respect to the underlying standard structure.

Proposition 3.2. *The structure* $(\mathsf{EB\_Log0}, [\![\cdot]\!])$ *constitutes a model of* $\mathsf{Log_0}$.

3.1.3. The Theory Log

The theory Log extends $\mathsf{Log_0}$ with more first-order logic symbols; these symbols are of a derived nature. I include them in the theory Core because they allow for more readable formulations of some rules, and they are useful for developing definitional extension methods in Section 3.2.

The theory Log provides the following symbols:

- $\top \wr \mathcal{B}$ (*truth*), $\bot \wr \mathcal{B}$ (*falsity*),
- $\neg \wr \mathcal{B} \rightharpoonup \mathcal{B}$ (*negation*), $\neq \wr (\alpha, \alpha) \rightharpoonup \mathcal{B}$ (*inequality*),
- $\vee \wr (\mathcal{B}, \mathcal{B}) \rightharpoonup \mathcal{B}$ (*disjunction*), $\Rightarrow \wr (\mathcal{B}, \mathcal{B}) \rightharpoonup \mathcal{B}$ (*implication*),
- $\Leftrightarrow \wr (\mathcal{B}, \mathcal{B}) \rightharpoonup \mathcal{B}$ (*equivalence*),
- $\exists \wr (\alpha \rightharpoonup \mathcal{B}) \rightharpoonup \mathcal{B}$ (*existential quantifier*),
- $\equiv_{\mathsf{eb}}{}^{\dagger} \wr (\alpha, \alpha) \rightharpoonup \mathcal{B}$ (*strong equality*), $\sqsubseteq_{\mathsf{eb}}{}^{\dagger} \wr (\alpha, \alpha) \rightharpoonup \mathcal{B}$ (*flat order*),
- $\mathsf{T}_{\mathsf{eb}}{}^{\dagger} \wr \mathcal{B} \rightharpoonup \mathcal{B}$ (*truth operator*), $\mathsf{WT}_{\mathsf{eb}}{}^{\dagger} \wr \mathcal{B} \rightharpoonup \mathcal{B}$ (*weak truth operator*).

The formula $\exists \underline{\boldsymbol{x}} \cdot \varphi$ abbreviates $\exists x_1 \cdot \ldots \exists x_{|\underline{\boldsymbol{x}}|} \cdot \varphi$.

The theory Log adds several rules to Log_0, which are presented in six blocks. The rules in the first block emphasize the "derived" status of the above symbols:

$$\begin{array}{lcl}
\mathsf{D}(\mathsf{D}(\bullet)) & \equiv & \top, \\
\mathsf{D}(\bullet) & \equiv & \bot, \\
\neg \$\varphi & \equiv & \$\varphi = \bot, \\
\$x \neq \$y & \equiv & \neg(\$x = \$y), \\
\$\varphi \vee \$\psi & \equiv & \neg(\neg\$\varphi \wedge \neg\$\psi), \\
\$\varphi \Rightarrow \$\psi & \equiv & \neg\$\varphi \vee \$\psi, \\
\$\varphi \Leftrightarrow \$\psi & \equiv & \$\varphi = \$\psi, \\
\exists x \cdot \$\varphi(x) & \equiv & \neg\forall x \cdot \neg\$\varphi(x), \\
(\$x \equiv_{\mathsf{eb}} \$y) & \equiv & (\mathsf{D}(\$x) \Leftrightarrow \mathsf{D}(\$y)) \wedge (\mathsf{D}(\$x) \Rightarrow \$x = \$y), \\
(\$x \sqsubseteq_{\mathsf{eb}} \$y) & \equiv & \mathsf{D}(\$x) \Rightarrow (\mathsf{D}(\$y) \wedge \$x = \$y), \\
\mathsf{T}_{\mathsf{eb}}(\$\varphi) & \equiv & \mathsf{D}(\$\varphi) \wedge \$\varphi, \\
\mathsf{WT}_{\mathsf{eb}}(\$\varphi) & \equiv & \mathsf{D}(\$\varphi) \Rightarrow \$\varphi.
\end{array}$$

The second block contains the following rules for reasoning about conditionals:

$$\begin{array}{lcl}
\mathsf{if}\ \top\ \mathsf{then}\ \$x\ \mathsf{else}\ \$y & \equiv & \$x, \\
\mathsf{if}\ \bot\ \mathsf{then}\ \$x\ \mathsf{else}\ \$y & \equiv & \$y.
\end{array}$$

The rules in the third block determine the domains of conjunction and the universal quantifier:

$$\begin{array}{lcl}
\mathsf{D}(\$\varphi \wedge \$\psi) & \equiv & (\mathsf{D}(\$\varphi) \wedge \mathsf{D}(\$\psi)) \vee \mathsf{T}_{\mathsf{eb}}(\neg\$\varphi) \vee \mathsf{T}_{\mathsf{eb}}(\neg\$\psi), \\
\mathsf{D}(\forall x \cdot \$\varphi(x)) & \equiv & \Big(\forall x \cdot \mathsf{D}(\$\varphi(x))\Big) \vee \Big(\exists x \cdot \mathsf{T}_{\mathsf{eb}}(\neg\$\varphi(x))\Big).
\end{array}$$

The above rules have the effect that conjunction is lazy in both arguments and the universal quantifier behaves like a "generalized conjunction". This kind of semantics is sometimes attributed to Kleene [1952, §64][1]. In fact, Metayer and Voisin [2009] give the impression that Event-B's conjunction and universal quantifier have *McCarthy semantics* [McCarthy, 1963], i.e., conjunction is strict in its first and lazy in its second argument, and the universal quantifier is smashed. The purpose of [Metayer and Voisin, 2009] is however not to define the domains of operators and binders but instead to describe an approximation technique (cf. Sections 3.4 and 4.3).

The rules in the fourth block define the domains of the remaining operators and

[1] Cheng and Jones [1990] meticulously describe the history of Kleene semantics.

binders:

$$
\begin{array}{lcl}
\mathsf{D}(\top) & \equiv & \top, \\
\mathsf{D}(\bot) & \equiv & \top, \\
\mathsf{D}(\neg \$\varphi) & \equiv & \mathsf{D}(\$\varphi), \\
\mathsf{D}(\$x \neq \$y) & \equiv & \mathsf{D}(\$x) \land \mathsf{D}(\$y), \\
\mathsf{D}(\$\varphi \lor \$\psi) & \equiv & \Big(\mathsf{D}(\$\varphi) \land \mathsf{D}(\$\psi)\Big) \lor \mathsf{T}_{\mathsf{eb}}(\$\varphi) \lor \mathsf{T}_{\mathsf{eb}}(\$\psi), \\
\mathsf{D}(\$\varphi \Rightarrow \$\psi) & \equiv & \Big(\mathsf{D}(\$\varphi) \land \mathsf{D}(\$\psi)\Big) \lor \mathsf{T}_{\mathsf{eb}}(\neg \$\varphi) \lor \mathsf{T}_{\mathsf{eb}}(\$\psi), \\
\mathsf{D}(\$\varphi \Leftrightarrow \$\psi) & \equiv & \mathsf{D}(\$\varphi) \land \mathsf{D}(\$\psi), \\
\mathsf{D}(\exists x \cdot \$\varphi(x)) & \equiv & \Big(\forall x \cdot \mathsf{D}(\$\varphi(x))\Big) \lor \Big(\exists x \cdot \mathsf{T}_{\mathsf{eb}}(\$\varphi(x))\Big).
\end{array}
$$

The rules in the fifth block are taken from [Mehta, 2008]; they emphasize similarities between Event-B's first-order fragment and classical first-order logic:

$$\frac{}{\vdash \top}\ \mathsf{truth_R} \qquad \frac{}{\bot \vdash \$\varphi}\ \mathsf{falsity_L}$$

$$\frac{\vdash \$\psi}{\neg \$\psi \vdash \$\varphi}\ \mathsf{neg_L} \qquad \frac{\$\varphi \vdash \bot}{\vdash \neg \$\varphi}\ \mathsf{neg_R}$$

$$\frac{\neg \$\varphi \vdash \bot}{\vdash \$\varphi}\ \mathsf{classical} \qquad \frac{\$\psi_1 \vdash \$\varphi \qquad \$\psi_2 \vdash \$\varphi}{\$\psi_1 \lor \$\psi_2 \vdash \$\varphi}\ \mathsf{disj_L}$$

$$\frac{\vdash \$\varphi_1}{\vdash \$\varphi_1 \lor \$\varphi_2}\ \mathsf{disj_{R1}} \qquad \frac{\vdash \$\varphi_2}{\vdash \$\varphi_1 \lor \$\varphi_2}\ \mathsf{disj_{R2}}$$

$$\frac{\vdash \$\psi_1 \qquad \$\psi_2 \vdash \$\varphi}{\$\psi_1 \Rightarrow \$\psi_2 \vdash \$\varphi}\ \mathsf{imp_L} \qquad \frac{\$\psi \vdash \$\varphi}{\vdash \$\psi \Rightarrow \$\varphi}\ \mathsf{imp_R}$$

$$\frac{\$\psi_1 \Rightarrow \$\psi_2, \$\psi_2 \Rightarrow \$\psi_1 \vdash \$\varphi}{\$\psi_1 \Leftrightarrow \$\psi_2 \vdash \$\varphi}\ \mathsf{eqv_L} \qquad \frac{\$\varphi_1 \vdash \$\varphi_2 \qquad \$\varphi_2 \vdash \$\varphi_1}{\vdash \$\varphi_1 \Leftrightarrow \$\varphi_2}\ \mathsf{eqv_R}$$

$$\frac{\$\psi(x) \vdash \$\varphi}{\exists x \cdot \$\psi(x) \vdash \$\varphi}\ \mathsf{ex_L} \quad (x \text{ fresh}) \qquad \frac{\vdash \mathsf{D}(\$y) \qquad \vdash \$\varphi(\$y)}{\vdash \exists x \cdot \$\varphi(x)}\ \mathsf{ex_R}$$

The sixth and last block contains rules that convert appropriate formulae into rewrite rules:

$$\frac{\$x \equiv_{\mathsf{eb}} \$y}{\$x \;\equiv\; \$y}\;\equiv_{\mathsf{eb}}\_\mathsf{reflection} \qquad \frac{\$x \sqsubseteq_{\mathsf{eb}} \$y}{\$x \;\sqsubseteq\; \$y}\;\sqsubseteq_{\mathsf{eb}}\_\mathsf{reflection}$$

The Standard Model. The *standard* structure $(\mathsf{EB\_Log}, [\![\cdot]\!])$ over the signature of Log definitionally extends the standard model of $\mathsf{Log_0}$, and defines the denotations of additional operators and binders as follows.

- The denotations of $\top$, $\bot$, $\neg$, and $\Leftrightarrow$ are the strict extensions of True, False, $\neg$, and $\longleftrightarrow$, respectively.
- The denotations of disjunction and implication are chosen such that $\$\varphi \vee \$\psi$ is equivalent to $\neg(\neg\$\varphi \wedge \neg\$\psi)$ and $\$\varphi \Rightarrow \$\psi$ is equivalent to $\neg\$\varphi \vee \$\psi$.
- The denotation of the existential quantifier is chosen such that $\exists x \cdot \$\varphi(x)$ is equivalent to $\neg(\forall x \cdot \neg\$\varphi(x))$.
- The denotations of the remaining operators are defined by

$$[\![\equiv_{\mathsf{eb}}]\!]\ x\ y = (x = y)\uparrow, \qquad [\![\sqsubseteq_{\mathsf{eb}}]\!]\ x\ y = (x \sqsubseteq y)\uparrow,$$
$$[\![\mathsf{T}_{\mathsf{eb}}]\!]\ \varphi = (\mathsf{T}\ \varphi)\uparrow, \qquad [\![\mathsf{WT}_{\mathsf{eb}}]\!]\ \varphi = (\mathsf{WT}\ \varphi)\uparrow.$$

It is straightforward to show (with Isabelle) that the rules of Log are sound with respect to the underlying standard structure.

Proposition 3.3. *The structure* $(\mathsf{EB\_Log}, [\![\cdot]\!])$ *constitutes a model of* Log.

3.1.4. The Theory $\mathsf{Set_0}$

The theory $\mathsf{Set_0}$ extends Log with set membership and a restricted version of set comprehension. More precisely, $\mathsf{Set_0}$ introduces

- the *set* type operator $\mathcal{P}$ of arity 1,
- the operator $\in \wr (\alpha, \mathcal{P}(\alpha)) \rightharpoonup \mathcal{B}$ (*membership*),
- the binder $\mathsf{collect}^{\dagger} \wr (\alpha \rightharpoonup \mathcal{B}) \rightharpoonup \mathcal{P}(\alpha)$ (*simple set comprehension*).

Simple set comprehension as such is not available in Rodin, but it is a special case of Rodin's *set comprehension*, which will be defined in Section 3.3.2.

In concrete syntax, the term $(\mathsf{collect}\, x \wr \nu \cdot \varphi) \wr \mathcal{P}(\nu)$ is written $\{x \wr \nu \mid \varphi\}$. Variables of set type are usually denoted R, S or $\$R, \$S$.

Apart from rules about well-definedness, the theory $\mathsf{Set_0}$ adds versions of beta-reduction and extensionality to Log:

$$\equiv \begin{array}{l} \mathsf{D}(\$x \in \$R) \\ \\ \mathsf{D}(\$x) \wedge \mathsf{D}(\$R) \end{array} \quad \mathsf{mem_D} \qquad \equiv \begin{array}{l} \mathsf{D}(\{x \mid \$\varphi(x)\}) \\ \\ \forall x \cdot \mathsf{D}(\$\varphi(x)) \end{array} \quad \mathsf{collect_D}$$

$$\$y \in \{x \mid \$\varphi(x)\} \;\sqsubseteq\; \$\varphi(\$y) \quad \mathsf{beta} \qquad \frac{\vdash\; x \in \$R \Leftrightarrow x \in \$S}{\vdash\; \$R = \$S}\ \mathsf{set\_extn} \quad (x \text{ fresh})$$

As an aside, note that universal (and therefore existential) quantification can be expressed in terms of simple set comprehension, similarly as in HOL [see e.g. Gordon and Pitts, 1993]:

$$\forall x \cdot \$\varphi(x) \equiv \big(\{x \mid \mathsf{T_{eb}}(\neg\$\varphi(x))\} = \{x \mid \bot\} \Rightarrow \bullet\big) \;\Rightarrow \{x \mid \mathsf{T_{eb}}(\$\varphi(x))\} = \{x \mid \top\}.$$

The Standard Model. The *standard* structure $(\mathsf{EB\_Set0}, [\![\cdot]\!])$ over the signature of $\mathsf{Set_0}$ definitionally extends the standard model of Log, and defines the denotation of $\mathcal{P}$ as set, the denotation of $\in$ as the strict extension of $\in$, and the denotation of collect as the smashed extension of Collect. Using Isabelle, I have proved that the rules of $\mathsf{Set_0}$ are sound with respect to the underlying standard structure.

Proposition 3.4. *The structure* $(\mathsf{EB\_Set0}, [\![\cdot]\!])$ *constitutes a model of* $\mathsf{Set_0}$.

3.1.5. The Theory Choice

The theory Choice extends $\mathsf{Set_0}$ with the *axiom of choice*, which is stated in terms of the *choice* binder $\mathsf{some}^\dagger \between (\alpha \rightharpoonup \mathcal{B}) \rightharpoonup \alpha$ and the following rules:

$$\mathsf{D}(\mathsf{some}\, x \cdot \$\varphi(x)) \;\equiv\; \Big(\forall x \cdot \mathsf{D}(\$\varphi(x))\Big) \wedge \Big(\exists x \cdot \$\varphi(x)\Big) \quad \mathsf{choice_D}$$

$$\frac{}{\vdash\; (\mathsf{some}\, x \cdot \$\varphi(x)) \in \{x \mid \$\varphi(x)\}}\ \mathsf{choice}$$

Note that, if $\mathsf{choice_D}$ is replaced by

$$\mathsf{D}(\mathsf{some}\, x \cdot \$\varphi(x)) \;\equiv\; \exists x \cdot \mathsf{T_{eb}}(\$\varphi(x)),$$

the choice binder fails to be monotonic. As monotonicity is useful for directed rewriting, I have decided to make the choice binder smashed (and therefore monotonic).

Actually, none of the sources about Event-B that I am aware of postulates the axiom of choice. I take the liberty to include the axiom of choice in Event-B's semantics for the following reasons.

1. I am not aware of an application of Event-B in which the axiom of choice seems an inadequate assumption.

2. The ML prover, which has been designed for classical B and is also available in Rodin, applies skolemization without restricting the use of set comprehensions[2]. As pointed out by Miller [1992], the combination of skolemization and comprehension axioms entails the axiom of choice. ML is therefore potentially unsound with respect to a semantics without the axiom of choice!

3. Similarly, if I excluded the axiom of choice from Event-B's standard model, only the version of Isabelle/HOL without the axiom of choice could be soundly integrated as a theorem prover into Rodin. An integration of this restricted version of Isabelle/HOL is considerably less attractive because it lacks useful features like the integer type and link-ups to automated theorem provers such as E [Schulz, 2002], Spass [Weidenbach et al., 2009], Vampire [Riazanov and Voronkov, 2002], and Z3 [de Moura and Bjørner, 2008].

The Standard Model. The *standard* structure $(\mathsf{EB\_Choice}, [\![\cdot]\!])$ over the signature of Choice definitionally extends the standard model of $\mathsf{Set_0}$, and defines the denotation of some such that

$$[\![\mathsf{some}\, x \cdot \varphi]\!] = \begin{cases} (\mathsf{SOME}\ x.\ \mathsf{T}\ [\![\varphi]\!])\uparrow & ((\forall x.\ \mathsf{WD}\ [\![\varphi]\!]) \wedge (\exists x.\ \mathsf{T}\ [\![\varphi]\!])) \\ \bullet & \text{(otherwise).} \end{cases}$$

I have proved (with Isabelle) that the rules of Choice are sound with respect to the underlying standard structure.

Proposition 3.5. *The structure* $(\mathsf{EB\_Choice}, [\![\cdot]\!])$ *constitutes a model of* Choice.

3.1.6. The Theory $\mathsf{Int_0}$

The theory $\mathsf{Int_0}$ extends $\mathsf{Set_0}$ with the type of integers as well as corresponding operators. The rules of $\mathsf{Int_0}$ essentially characterize Event-B's integers as a well-ordered integral domain and have been influenced by the exposition of Mendelson [1973, Ch. 3]. The axioms of a well-ordered integral domain are known to characterize the integers up to isomorphism [see Mendelson, 1973, p. 151, for a precise definition and proof of this statement].

The signature of $\mathsf{Int_0}$ introduces the type $\mathcal{Z}$ of integers and the operators

$$\begin{array}{lll} 0 \wr \mathcal{Z}, & 1 \wr \mathcal{Z}, & + \wr (\mathcal{Z}, \mathcal{Z}) \rightharpoonup \mathcal{Z}, \\ \mathsf{uminus} \wr \mathcal{Z} \rightharpoonup \mathcal{Z}, & * \wr (\mathcal{Z}, \mathcal{Z}) \rightharpoonup \mathcal{Z}, & \leq \wr (\mathcal{Z}, \mathcal{Z}) \rightharpoonup \mathcal{B}, \\ \mathsf{min} \wr \mathcal{P}(\alpha) \rightharpoonup \mathcal{Z}. & & \end{array}$$

Terms of the form $\mathsf{uminus}(t)$ are usually written $-t$.

The rules that $\mathsf{Int_0}$ adds to $\mathsf{Set_0}$ are presented in five blocks. The first block contains rules describing the domains of the operators introduced by $\mathsf{Int_0}$:

[2] I am grateful to Thierry Lecomte for bringing this to my attention.

$$
\begin{array}{lcl}
\mathsf{D}(0) & \equiv & \top, \\
\mathsf{D}(1) & \equiv & \top, \\
\mathsf{D}(\$x + \$y) & \equiv & \mathsf{D}(\$x) \wedge \mathsf{D}(\$y), \\
\mathsf{D}(-\$x) & \equiv & \mathsf{D}(\$x), \\
\mathsf{D}(\$x * \$y) & \equiv & \mathsf{D}(\$x) \wedge \mathsf{D}(\$y), \\
\mathsf{D}(\$x \leq \$y) & \equiv & \mathsf{D}(\$x) \wedge \mathsf{D}(\$y), \\
\mathsf{D}(\mathsf{min}(\$R)) & \equiv & \mathsf{D}(\$R) \wedge (\exists w \cdot w \in \$R) \wedge (\exists b \cdot \forall x \cdot x \in \$R \Rightarrow b \leq x).
\end{array}
$$

The rules in the second block state that $\mathcal{Z}$ forms an *integral domain*:

$$\frac{}{\vdash (\$x + \$y) + \$z = \$x + (\$y + \$z)}\ \mathsf{add\_associativity}$$

$$\frac{}{\vdash \$x + \$y = \$y + \$x}\ \mathsf{add\_commutativity}$$

$$\frac{}{\vdash \$x + 0 = \$x}\ \mathsf{add\_neutral} \qquad \frac{}{\vdash \$x + (-\$x) = 0}\ \mathsf{add\_inverse}$$

$$\frac{}{\vdash (\$x * \$y) * \$z = \$x * (\$y * \$z)}\ \mathsf{mult\_associativity}$$

$$\frac{}{\vdash \$x * \$y = \$y * \$x}\ \mathsf{mult\_commutativity} \qquad \frac{}{\vdash \$x * 1 = \$x}\ \mathsf{mult\_neutral}$$

$$\frac{}{\vdash \$x * (\$y + \$z) = (\$x * \$y) + (\$x * \$z)}\ \mathsf{distributivity}$$

$$\frac{}{\$x * \$y = 0 \ \vdash\ \$x = 0 \vee \$y = 0}\ \mathsf{no\_zero\_divisors}$$

The rules in the third block postulate that the integral domain $\mathcal{Z}$ is *ordered*. Mendelson [1973] defines ordered integral domains in terms of $<$. It is easy to check that my formulation in terms of $\leq$ is equivalent, assuming that $\$x < \$y \equiv \$x \leq \$y \wedge \$x \neq \$y$.

$$\frac{}{\vdash \$x \leq \$x}\ \mathsf{reflexivity} \qquad \frac{}{\$x \leq \$y,\ \$y \leq \$z\ \vdash\ \$x \leq \$z}\ \mathsf{transitivity}$$

$$\frac{}{\vdash \$x \leq \$y \vee \$y \leq \$x}\ \mathsf{totality} \qquad \frac{}{\$x \leq \$y,\ \$y \leq \$x \ \vdash\ \$x = \$y}\ \mathsf{antisymmetry}$$

$$\frac{}{\$x \leq \$y \ \vdash\ \$x + \$z \leq \$y + \$z}\ \mathsf{add\_comp}$$

$$\frac{}{\$x \leq \$y,\ 0 \leq \$z \ \vdash\ \$x * \$z \leq \$y * \$z}\ \mathsf{mult\_comp}$$

The theory stated so far posses a model that interprets $\mathcal{Z}$ as a type with only one element. This model is excluded by the fourth block:

$$\frac{}{\vdash\ 0 \neq 1}\ \mathsf{infinity}.$$

The name infinity is due to the fact that the underlying rule imposes the existence of an infinite number of integers [Mendelson, 1973, p. 129, p. 136].

The rules in the fifth and last block state that $\mathcal{Z}$ constitutes a *well-ordered* integral domain. They exclude the model that interprets $\mathcal{Z}$ as the rationals.

$$\frac{}{\vdash\ \mathsf{min}(\$R) \in \$R}\ \mathsf{min\_mem},\qquad \frac{\vdash\ \$x \in \$R}{\vdash\ \mathsf{min}(\$R) \leq \$x}\ \mathsf{min\_bound}.$$

The Standard Model. The *standard* structure $(\mathsf{EB\_Int0}, [\![\cdot]\!])$ of Int_0 definitionally extends the standard model of Set_0. The denotation of $\mathcal{Z}$ is int, and the denotation of min is as follows:

$$[\![\mathsf{min}]\!]\ \bullet = \bullet,$$

$$[\![\mathsf{min}]\!]\ R\uparrow = \begin{cases} \mathsf{Least}\ R & (R \neq \varnothing \wedge (\exists b \cdot \forall x \cdot x \in R \Rightarrow b \leq x)) \\ \bullet & (\text{otherwise}). \end{cases}$$

Every remaining operator f_{EB} of Int_0 is interpreted as the strict extension of the HOL constant f_{HOL} according to the following table:

f_{EB}	0	1	+
f_{HOL}	0 :: int	1 :: int	plus :: int ⇒ int ⇒ int

f_{EB}	uminus	*	≤
f_{HOL}	uminus :: int ⇒ int	times :: int ⇒ int ⇒ int	less_eq :: int ⇒ int ⇒ bool

I have used Isabelle to prove that every rule introduced by Int_0 is sound with respect to the underlying standard structure.

Lemma 3.6. *The structure* $(\mathsf{EB\_Int0}, [\![\cdot]\!])$ *constitutes a model of* $\mathsf{Int_0}$.

3.1.7. The Theory Prod

The theory Prod extends Log with the notion of Cartesian product. It introduces the *product* type operator $\star$ of arity 2 and the *product* operator $\mapsto \circledast (\alpha, \beta) \rightharpoonup \alpha \star \beta$. The additional rules of Prod are as follows:

$$\mathsf{D}(\$x \mapsto \$y) \;\equiv\; \mathsf{D}(\$x) \wedge \mathsf{D}(\$y) \quad \mathsf{prod\_D}$$

$$\$x_1 \mapsto \$x_2 = \$y_1 \mapsto \$y_2 \;\sqsubseteq\; \$x_1 = \$y_1 \wedge \$x_2 = \$y_2 \quad \mathsf{prod\_inj}$$

$$\frac{\vdash\ \mathsf{D}(\$x) \qquad \$x = x_1 \mapsto x_2 \ \vdash\ \$\varphi}{\vdash\ \$\varphi}\ \mathsf{prod\_surj} \quad (x_1, x_2 \text{ fresh})$$

The Standard Model. The *standard* structure $(\mathsf{EB\_Prod}, [\![\cdot]\!])$ over the signature of Prod definitionally extends the standard model of Log, and defines the denotations of $\star$ as prod and of $\mapsto$ as the strict extension of Pair. Using Isabelle, it is easy to show that the rules of Prod are sound with respect to the underlying standard structure.

Proposition 3.7. *The structure* $(\mathsf{EB\_Prod}, [\![\cdot]\!])$ *constitutes a model of* Prod.

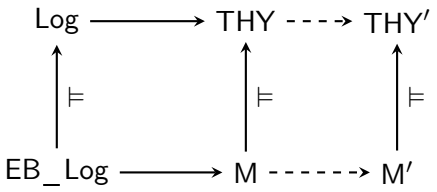
3.1.8. The Theory Core

The theory Core results from merging $\mathsf{Int_0}$, Choice, and Prod. The *standard* structure $(\mathsf{EB\_Core}, [\![\cdot]\!])$ of Core results from merging the standard models of $\mathsf{Int_0}$, Choice, and Prod. It is easy to check that $(\mathsf{EB\_Core}, [\![\cdot]\!])$ constitutes a model of Core.

3.2. Definitional Extensions

During the development of Event-B's core theories, I have introduced new symbols and rules in an ad-hoc manner, proving soundness of new rules directly with Isabelle/HOL. Although this approach has been effective so far, I regard it as impractical in general, because it requires Rodin users who want to extend theories to become acquainted with Isabelle/HOL. The subject of this section is to develop methods for extending theories that can be applied without performing proofs in Isabelle/HOL. Figure 3.2 illustrates the main properties of these methods on an intuitive level.

Most extension methods defined in this section apply only to theories extending Log and models extending EB_Log. This restriction is necessary, because some methods give rise to proof obligations that are expressed using symbols from Log; these proof obligations fulfill their purposes only if symbols of Log are interpreted according to EB_Log.



Vertical arrows connect a model with its theory. Horizontal arrows indicate theory or structure extensions. Dashed arrows indicate definitional extensions.

Figure 3.2.: Definitional extensions

The extension methods are *definitional* in the following sense. Suppose the theory THY has the model M, and some definitional extension method can be used to extend THY to THY′. Then there exists a model M′ of THY′ that *definitionally* extends M. This has the effect that the notion of validity is preserved in the sense that a sequent over the signature of THY is valid with respect to M′ iff it is valid with respect to M.

In fact, if a theory THY with a model M is extended to THY′ using one of the extension methods of this section, then the model M′ of THY′ that definitionally extends M is uniquely determined. In this context, I consider two structures over a given signature as equal iff they induce the same notion of validity. Uniqueness of M′ implies that the additional rules of THY′ fully specify the denotations of additional symbols.

There is a wide variety of extension methods that could be treated within this section. I restrict my focus on the methods that are required to extend the theory Core to Event-B's main theory EventB (see Section 3.3). That excludes methods for defining new types, datatypes, and records, as well as methods for defining functions recursively.

Related Work. The usage of the term "definitional" and the methods for defining new operators have been strongly influenced by Gordon and Pitts [1993]; the underlying idea goes already back to Russell [1948, p. 71], who compares arbitrary extensions with "theft" and definitional extensions with "honest toil". Adding binders and rules to a theory involves some difficulties that are unique in the setting of Event-B and Rodin; one of these difficulties stems from the unavailability of operator variables in Rodin (cf. Section 3.2.4).

An alternative requirement on theory extensions is *conservativity* [see e.g. Shoenfield 1967, p. 41 or van Dalen 2004, p. 104]: a theory THY′ *conservatively* extends a theory THY iff THY′ extends THY and the sequents over THY provable in THY′ are provable in THY as well. In the context of this document, it is more appropriate to require that extensions are definitional because the semantics of the symbols of interest is ultimately given by a model; the corresponding proof calculus merely illustrates this semantics, but does not define it.

The notions of "definitional" and "conservative" are incomparable: Veloso and

Veloso [1991] point out that conservative extensions need not be definitional, and according to Gödel's first incompleteness theorem, definitional extensions need not be conservative[3].

Maamria and Butler [2010] show how to reason about soundness of directed rewrite rules in the context of Event-B. They only consider rules that are built from ordinary variables and strict operators. My work covers a substantially more general class of rules and corrects a mistake concerning the application of rules. (The proviso above (4.2) on Page 11 in [Maamria and Butler, 2010] allows one to apply conditional rewrite rules in an unsound way.)

3.2.1. New Rules

To describe the proof obligations that need to be proved when introducing new rules, I define some auxiliary notation:

$$\mathcal{T}^{-1}(\underline{\boldsymbol{\psi}} \vdash \varphi) := \mathsf{T}_{\mathsf{eb}}(\psi_1) \wedge \cdots \wedge \mathsf{T}_{\mathsf{eb}}(\psi_{|\underline{\boldsymbol{\psi}}|}) \Rightarrow \mathsf{WT}_{\mathsf{eb}}(\varphi),$$
$$\mathcal{WT}^{-1}(\underline{\boldsymbol{\psi}} \vdash \varphi) := \psi_1 \wedge \cdots \wedge \psi_{|\underline{\boldsymbol{\psi}}|} \Rightarrow \varphi.$$

The mappings $\mathcal{T}^{-1}$, $\mathcal{WT}^{-1}$ should be viewed as syntactic transformations, similarly as substitutions.

Lemma 3.8. *Let* $(\mathsf{M}, [\![\cdot]\!])$ *be a structure extending* EB_Log *and* Γ *a sequent over the signature of* $(\mathsf{M}, [\![\cdot]\!])$. *Then,*

$$\mathsf{T} \ [\![\mathcal{T}^{-1}(\Gamma)]\!] = [\![\Gamma]\!],$$
$$\mathsf{WT} \ [\![\mathcal{WT}^{-1}(\Gamma)]\!] = [\![\Gamma]\!]$$

are valid with respect to M.

Suppose THY is a theory extending Log with a model M extending EB_Log. Adding a rule r to THY is *admissible* iff the following proof obligation, depending on the rule to be added, is valid[4] with respect to M:

Rule	Proof Obligations
$\dfrac{\underline{\boldsymbol{\Gamma}}}{\Gamma_0}$ ($\underline{\boldsymbol{x}}$ fresh)	$\forall \underline{\boldsymbol{x}} \cdot \mathcal{T}^{-1}(\Gamma_1) \wedge \cdots \wedge \mathcal{T}^{-1}(\Gamma_{\lvert\underline{\boldsymbol{\Gamma}}\rvert}) \vdash \mathcal{WT}^{-1}(\Gamma_0)$
$\dfrac{\varphi}{t \equiv u}$	$\mathsf{WT}_{\mathsf{eb}}(\varphi) \vdash t \equiv_{\mathsf{eb}} u$
$\dfrac{\varphi}{t \sqsubseteq u}$	$\mathsf{WT}_{\mathsf{eb}}(\varphi) \vdash t \sqsubseteq_{\mathsf{eb}} u$

[3] According to Gödel's first incompleteness theorem, there is a sequent of Core that is valid but not provable. A definitional extension of Core that is not conservative adds a rule to Core that proves exactly that sequent.

[4] Alternatively, I could require that the proof obligation is provable in THY. The current setting is slightly more general: it allows users either to prove the proof obligation with Rodin or its denotation with Isabelle/HOL.

Some readers may object that these proof obligations are so straightforward that the whole issue of introducing new rules seems trivial. It was indeed not difficult to find these proof obligations. The real difficulty was to present Event-B's logic in such a way that these proof obligations can be stated conveniently. This involved introducing T_{eb}, $\mathsf{WT}_{\mathsf{eb}}$, $\equiv_{\mathsf{eb}}$, $\sqsubseteq_{\mathsf{eb}}$, inventing operator variables, and developing the underlying meta-theory (mainly in Sections 2.2.7 and 2.3.2).

The following theorem shows that adding a rule to a theory is definitional iff it is admissible.

Theorem 3.9. *Let* THY *be a theory extending* Log *and* M *a model of* THY *extending* $\mathsf{EB\_Log}$. *Suppose that* THY' *results from adding the rule* r *(over the signature of* THY*) to* THY. *The addition of* r *to* THY *is admissible iff it is definitional. Moreover, every model of* THY' *definitionally extending* M *induces the same notion of validity as* M.

Proof. Using Lemma 3.8, it is easy to check that the rule r is sound with respect to M iff the corresponding admissibility proof obligation is valid with respect to M. This gives the first part of the assertion.

For the second part, suppose M' is an arbitrary model of THY' definitionally extending M. Hence, a sequent Γ is valid with respect to M' iff it is valid with respect to M. This gives the second part of the assertion. □

3.2.2. New Operators

3.2.2.1. Operator Definitions

An *(ordinary) operator definition* is written

$$f(\underline{\$\boldsymbol{x}} \wr \underline{\boldsymbol{\nu}}) \wr \mu \equiv t \wr \mu, \tag{3.1}$$

where $\underline{\$\boldsymbol{x}}$ has pairwise distinct elements. Given a theory $\mathsf{THY} = (\Sigma, \mathsf{rules})$, the operator definition in (3.1) yields the theory obtained by adding the operator $f \wr \underline{\boldsymbol{\nu}} \rightharpoonup \mu$ to Σ and the rule $f(\underline{\$\boldsymbol{x}}) \equiv t$ to rules. Applying the operator definition in (3.1) to THY is *admissible* iff the following conditions hold:

1. The symbol f is a non-logical symbol, but not a type operator, operator, or binder of Σ.
2. The elements of $\underline{\boldsymbol{\nu}}\,\mu$ are types over Σ, and $t \wr \mu$ is a term over Σ.
3. All variables free in $t \wr \mu$ (including type variables) occur in $\underline{\$\boldsymbol{x}}$, $\underline{\boldsymbol{\nu}}$, or μ.

Condition 3 excludes operator definitions like $c \wr \mathcal{B} \equiv \forall x \wr \alpha, y \cdot x = y$. Such an operator definition fails to be definitional, because it generates a rule that precludes models in which some type has exactly one inhabitant and another type has more than one inhabitant. The underlying phenomenon has already been identified by Gordon and Pitts [1993].

The following theorem shows that operator definitions are definitional.

Theorem 3.10. *Suppose that applying the operator definition* $f(\underline{\$\boldsymbol{x}}) \equiv t$ *to the theory* THY *with the model* $(\mathsf{M}, [\![\cdot]\!])$ *is admissible and results in* THY′. *Then there is a unique structure* $(\mathsf{M}', [\![\cdot]\!]')$ *that is a model of* THY′ *and a definitional extension of* $(\mathsf{M}, [\![\cdot]\!])$.

Proof. I first prove the existence of $(\mathsf{M}', [\![\cdot]\!]')$ and then its uniqueness. The structure $(\mathsf{M}', [\![\cdot]\!]')$ results from extending $(\mathsf{M}, [\![\cdot]\!])$ as follows. The denotation function $[\![\cdot]\!]'$ coincides with $[\![\cdot]\!]$ except that $[\![f]\!]'$ is a constant not used in M. Condition 1 rules out the pathological situation where $[\![f]\!]$ is already present in the signature underlying M. The theory M′ results from adding the definition

$$\textbf{definition} \quad [\![f]\!]' \; [\![\underline{\$\boldsymbol{x}}]\!] = [\![t]\!]$$

to M. Condition 3 enforces HOL's side-conditions on definitions [see Gordon and Pitts, 1993, Wenzel, 1997]. The structure $(\mathsf{M}', [\![\cdot]\!]')$ obviously constitutes a definitional extension of $(\mathsf{M}, [\![\cdot]\!])$ and a model of THY′.

Recall that two structures are considered equal iff they induce the same notion of validity. Intuitively, I show uniqueness of $(\mathsf{M}', [\![\cdot]\!]')$ by giving a characterization of validity that is independent from $(\mathsf{M}', [\![\cdot]\!]')$. Let Γ' be a sequent over the signature of THY′ and $(\mathsf{M}', [\![\cdot]\!]')$ a model of THY′ that definitionally extends $(\mathsf{M}, [\![\cdot]\!])$. The sequent Γ results from eliminating all occurrences of f from Γ' by repeatedly applying the rewrite rule $f(\underline{\$\boldsymbol{x}}) \equiv t$. Then,

$$\begin{aligned} & \Gamma' \text{ is valid with respect to } (\mathsf{M}', [\![\cdot]\!]') \\ \text{iff} \quad & \Gamma \text{ is valid with respect to } (\mathsf{M}', [\![\cdot]\!]') \\ \text{iff} \quad & \Gamma \text{ is valid with respect to } (\mathsf{M}, [\![\cdot]\!]). \end{aligned}$$

The first equivalence holds, because the rule $f(\underline{\$\boldsymbol{x}}) \equiv t$ is sound with respect to $(\mathsf{M}', [\![\cdot]\!]')$. The second equivalence holds, because $(\mathsf{M}', [\![\cdot]\!]')$ definitionally extends $(\mathsf{M}, [\![\cdot]\!])$. Overall, the models of THY′ definitionally extending $(\mathsf{M}, [\![\cdot]\!])$ are uniquely determined, because the last line is independent from $(\mathsf{M}', [\![\cdot]\!]')$. □

Monotonicity. To make directed rewriting applicable to the arguments of a newly defined operator, monotonicity needs to be proved. The following proposition gives the corresponding proof obligation:

Proposition 3.11. *Let* THY *be a theory extending* Log *and* M *a model of* THY *extending* EB_Log. *Suppose that applying the operator definition* $f(\underline{\$\boldsymbol{x}}) \equiv t$ *to* THY *is admissible and results in* THY′. *Then* f *is monotonic with respect to the (unique) model* M′ *of* THY′ *definitionally extending* M *iff the following proof obligation is valid with respect to* M*:*

$$\$x_1 \sqsubseteq_{\mathsf{eb}} \$y_1, \ldots, \$x_{|\underline{\$\boldsymbol{x}}|} \sqsubseteq_{\mathsf{eb}} \$y_{|\underline{\$\boldsymbol{x}}|} \;\vdash\; t \sqsubseteq_{\mathsf{eb}} (t[\underline{\$\boldsymbol{x}} := \underline{\$\boldsymbol{y}}]). \tag{3.2}$$

Here, the elements of $\underline{\$\boldsymbol{y}}$ *are pairwise distinct and do not occur in* $\underline{\$\boldsymbol{x}}$.

Proof. The assertion follows from the observation that the denotation of (3.2) is equivalent to the monotonicity of f. □

Proving monotonicity proof obligations can sometimes be avoided. If the right-hand side of an operator definition is built from monotonic operators and binders, then the resulting operator is monotonic.

Proposition 3.12. *Suppose that applying the operator definition* $f(\underline{\$\boldsymbol{x}}) \equiv t$ *to the theory* THY *is admissible and results in* THY′. *If the operators and binders in* t *are monotonic with respect to the model* M *of* THY, *then* f *is monotonic with respect to the (unique) model* M′ *of* THY′ *definitionally extending* M.

Proof. The assertion can be proved by structural induction. □

3.2.2.2. Strict Operator Definitions

It is sometimes clumsy to apply ordinary operator definitions for defining strict operators. For example, one might define intersection by

$$\$R \cap_l \$S \equiv \{x \mid x \in \$R \wedge x \in \$S\}.$$

This version of intersection however fails to be strict, because $\bullet \cap_l \varnothing$ is equivalent to $\varnothing$. Ordinary operator definitions bear the risk that users inadvertently define operators to be lazy.

Strict intersection may be defined by

$$\$R \cap \$S \equiv \mathsf{if}\ \mathsf{D}(\$R) \wedge \mathsf{D}(\$S)\ \mathsf{then}\ \{x \mid x \in \$R \wedge x \in \$S\}\ \mathsf{else}\ \bullet.$$

To achieve some degree of convenience, it is useful to additionally introduce the rewrite rules

$$\begin{aligned} \mathsf{D}(\$R \cap \$S) &\equiv \mathsf{D}(\$R) \wedge \mathsf{D}(\$S), \\ \mathsf{D}(\$R \cap \$S) &\sqsubseteq \{x \mid x \in \$R \wedge x \in \$S\}. \end{aligned}$$

Since many operators in Event-B are strict, I develop a dedicated method for defining them conveniently.

A *strict* operator definition is written as

$$f(\underline{\boldsymbol{x}} \mathbin{⦂} \underline{\boldsymbol{\nu}}) \mathbin{⦂} \mu \sqsubseteq t \mathbin{⦂} \mu \quad \text{with} \quad \mathsf{D}(f(\underline{\boldsymbol{x}})) \equiv \varphi \mathbin{⦂} \mathcal{B}, \tag{3.3}$$

where $\underline{\boldsymbol{x}}$ has pairwise distinct elements. Suppose $\mathsf{THY} = (\Sigma, \mathsf{rules})$ is a theory extending Log and M a model of THY extending EB_Log. The strict operator definition in (3.3) maps THY to the theory obtained by adding the operator $f \mathbin{⦂} \underline{\boldsymbol{\nu}} \rightharpoonup \mu$ to Σ and

the following rewrite rules to rules:

$$f(\underline{\$\boldsymbol{x}}) \equiv \mathsf{if}\ \mathsf{D}(\$x_1) \wedge \cdots \wedge \mathsf{D}(\$x_{|\underline{\$\boldsymbol{x}}|}) \wedge \varphi'\ \mathsf{then}\ t'\ \mathsf{else}\ \bullet, \tag{3.4}$$

$$\mathsf{D}(f(\underline{\$\boldsymbol{x}})) \equiv \mathsf{D}(\$x_1) \wedge \cdots \wedge \mathsf{D}(\$x_{|\underline{\$\boldsymbol{x}}|}) \wedge \varphi', \tag{3.5}$$

$$f(\underline{\$\boldsymbol{x}}) \sqsubseteq t'. \tag{3.6}$$

Here, the sequence $\underline{\$\boldsymbol{x}}$ has pairwise distinct elements, $|\underline{\$\boldsymbol{x}}| = |\underline{\boldsymbol{x}}|$, and t' and φ' result from replacing every free occurrence of x_i by $\$x_i$, for $1 \leq i \leq |\underline{\boldsymbol{x}}|$, in t and φ, respectively.

Applying the strict operator definition in (3.3) to THY is *admissible* iff the following conditions hold:

1. The symbol f is a non-logical symbol, but not a type operator, operator, or binder of Σ.
2. The elements of $\underline{\boldsymbol{\nu}}\,\mu$ are types over Σ, and $t \wr \mu$ and $\varphi \wr \mathcal{B}$ are terms over Σ.
3. All variables free in $t \wr \mu$ or $\varphi \wr \mathcal{B}$ (including type variables) occur in $\underline{\boldsymbol{x}}$, $\underline{\boldsymbol{\nu}}$, or μ.
4. The sequents $\vdash \mathsf{D}(\varphi)$ and $\varphi \vdash \mathsf{D}(t)$ are valid with respect to M.

It remains to show that strict operator definitions are definitional.

Theorem 3.13. *Let* THY *be a theory extending* Log *and* $(\mathsf{M}, [\![\cdot]\!])$ *a model of* THY *extending* $\mathsf{EB\_Log}$. *Suppose that applying the strict operator definition in (3.3) to* THY *is admissible and results in* THY'. *Then there is a unique structure* $(\mathsf{M}', [\![\cdot]\!]')$ *that is a model of* THY' *and a definitional extension of* $(\mathsf{M}, [\![\cdot]\!])$.

Proof. The theory THY' results from applying the operator definition (3.4) and subsequently adding the rules (3.5) and (3.6) to THY. The admissibility conditions 1–3 of the strict operator definition entail the admissibility conditions of the operator definition (3.4). The admissibility condition 4 entails that it is admissible to introduce the rules (3.5) and (3.6).

As the extension of THY to THY' is definitional, there exists a model $(\mathsf{M}', [\![\cdot]\!]')$ of THY' that definitionally extends $(\mathsf{M}, [\![\cdot]\!])$. Uniqueness can be shown similarly as for ordinary operator definitions (cf. Theorem 3.10). □

Note that operators introduced by strict operator definitions are strict and therefore also monotonic.

Definite Variant. The *definite strict operator definition*

$$f(\underline{\boldsymbol{x}} \wr \underline{\boldsymbol{\nu}}) \wr \mu \sqsubseteq t \wr \mu$$

is a short-hand for the strict operator definition

$$f(\underline{\boldsymbol{x}} \wr \underline{\boldsymbol{\nu}}) \wr \mu \sqsubseteq t \wr \mu \qquad \text{with} \qquad \mathsf{D}(f(\underline{\boldsymbol{x}})) \equiv \top.$$

The admissibility conditions 1–3 remain unchanged. Condition 4 simplifies to the requirement that $\vdash \mathsf{D}(t)$ is valid.

Simplifications. The following lemma is useful to discharge proof obligations arising from strict operator definitions:

Proposition 3.14. *Suppose the term t is built from ordinary variables, definite operators, and definite binders. Then the sequent $\vdash \mathsf{D}(t)$ is valid.*

Proof. The assertion can be proved by structural induction. □

3.2.3. New Binders

When developing binder definitions for Event-B, one has to cope with a mismatch between the types of denotations of binders and of operator variables. To understand this point, it is instructive to focus on binders binding one variable and taking one argument. Such a binder denotes a function of type $(\llbracket\nu\rrbracket \Rightarrow \llbracket\mu\rrbracket\uparrow) \Rightarrow \xi\uparrow$. To define such a binder, it is useful to have a variable whose denotation is of type $\llbracket\nu\rrbracket \Rightarrow \llbracket\mu\rrbracket\uparrow$; this variable serves as place-holder for the argument of the binder.

Event-B's logic does not provide such a variable, but it provides an operator variable $\$f$ whose denotation is of type $\llbracket\nu\rrbracket\uparrow \Rightarrow \llbracket\mu\rrbracket\uparrow$. If $\$f$ is used as a place-holder in the definition of a binder, the problem is that $\$f$ provides more information than the binder can absorb, namely $\$f(\bullet)$. To ensure that binder definitions are definitional, it is necessary that the right-hand side of a binder definition does not depend on $\$f(\bullet)$. In general, this can be enforced by a proof obligation; in practice, it is likely that this proof obligation is proved automatically by a trivial tactic.

An alternative approach is to invent a new kind of variable whose denotation is of type $\llbracket\nu\rrbracket \Rightarrow \llbracket\mu\rrbracket\uparrow$. The logic of partial functions by Farmer [1990] provides such variables. These variables help to formulate binder definitions somewhat more elegantly. But they are generally inadequate as place-holders in rules, because they cannot be soundly instantiated with lazy operators. I have decided against introducing another kind of variable, because that would complicate the design of Event-B's logic for the small benefit of avoiding a proof obligation that can often be hidden from the user anyway.

A *binder definition* is written

$$(Q\underline{\boldsymbol{x}} \wr \underline{\boldsymbol{\nu}} \cdot \underline{\$\boldsymbol{f}}(\underline{\boldsymbol{x}}) \wr \underline{\boldsymbol{\mu}}) \wr \xi \equiv t \wr \xi, \tag{3.7}$$

where $\underline{\boldsymbol{x}}$ and $\underline{\$\boldsymbol{f}}$ are sequences of pairwise distinct ordinary or operator variables. The notation $\underline{\$\boldsymbol{f}}(\underline{\boldsymbol{x}}) \wr \underline{\boldsymbol{\mu}}$ stands for $\$f_1(\underline{\boldsymbol{x}}) \wr \mu_1, \ldots, \$f_{|\underline{\$\boldsymbol{f}}|}(\underline{\boldsymbol{x}}) \wr \mu_{|\underline{\boldsymbol{\mu}}|}$.

Suppose $\mathsf{THY} = (\Sigma, \mathsf{rules})$ is a theory extending Log with a model M extending $\mathsf{EB\_Log}$. The binder definition in (3.7) maps THY to the theory obtained by adding the binder $Q \wr (\underline{\boldsymbol{\nu}} \rightharpoonup \underline{\boldsymbol{\mu}}) \rightharpoonup \xi$ to Σ and the rewrite rule in (3.7) to rules. Applying the binder definition in (3.7) to THY is *admissible* iff the following conditions hold:

1. The symbol Q is a non-logical symbol, but not a type operator, operator, or binder of Σ.

2. The elements of $\underline{\boldsymbol{\nu}}$, $\underline{\boldsymbol{\mu}}$, and ξ are types over Σ, and $t \wr \xi$ is a term over Σ.

3. The term $t \wr \xi$ does not contain free ordinary variables, and each type or operator variable occurring in t belongs to $\underline{\$\boldsymbol{f}}$, $\underline{\boldsymbol{\nu}}$, $\underline{\boldsymbol{\mu}}$, or ξ.

4. The sequent

$$\begin{aligned} & \forall \underline{\boldsymbol{x}} \cdot \$f_1(\underline{\boldsymbol{x}}) \equiv_{\mathsf{eb}} \$g_1(\underline{\boldsymbol{x}}) \\ & \ldots \\ & \forall \underline{\boldsymbol{x}} \cdot \$f_{|\underline{\$\boldsymbol{f}}|}(\underline{\boldsymbol{x}}) \equiv_{\mathsf{eb}} \$g_{|\underline{\$\boldsymbol{f}}|}(\underline{\boldsymbol{x}}) \\ \vdash \quad & t \equiv_{\mathsf{eb}} t[\underline{\$\boldsymbol{f}}(\underline{\$\boldsymbol{x}}) := \underline{\$\boldsymbol{g}}(\underline{\$\boldsymbol{x}})] \end{aligned}$$

is valid with respect to M. Here, the elements of $\underline{\$\boldsymbol{g}}$ are pairwise distinct and do not occur in $\underline{\$\boldsymbol{f}}$.

Condition 4 ensures that t does not depend on $\$f(\bullet)$. It rules out binder definitions like

$$Qx \cdot \$f(x) \wr \mathcal{B} \equiv \$f(\bullet). \tag{3.8}$$

The result of extending Log with the rule (3.8) is inconsistent. An arbitrary sequent can be proved based on the following observations:

$$\begin{aligned} Qx \cdot \mathsf{D}(x) \quad &\equiv \quad \mathsf{D}(\bullet) \quad \equiv \quad \bot, \\ Qx \cdot \mathsf{D}(x) \quad &\equiv \quad Qx \cdot \top \quad \equiv \quad \top. \end{aligned}$$

Finally, I show that binder definitions are definitional.

Theorem 3.15. *Let* THY *be a theory extending* Log *and* $(\mathsf{M}, [\![\cdot]\!])$ *a model of* THY *extending* $\mathsf{EB\_Log}$. *Suppose that applying the binder definition in (3.7) to* THY *is admissible and results in* THY'. *Then there is a unique structure* $(\mathsf{M}', [\![\cdot]\!]')$ *that constitutes a model of* THY' *and definitionally extends* $(\mathsf{M}, [\![\cdot]\!])$.

Proof. Let n be the length of $\underline{\boldsymbol{x}}$ and m the length of $\underline{\$\boldsymbol{f}}$. I define the structure $(\mathsf{M}', [\![\cdot]\!]')$ as follows. The denotation function $[\![\cdot]\!]'$ coincides with $[\![\cdot]\!]$ except that $[\![Q]\!]'$ is a constant not used in M. Note that $[\![Q]\!]'$ cannot be defined by

$$\textbf{definition} \quad [\![Q]\!]' \ \underline{\$\boldsymbol{f}} = [\![t]\!],$$

because $[\![Q]\!]'$ is of type

$$([\![\nu_1]\!] \Rightarrow \ldots \Rightarrow [\![\nu_n]\!] \Rightarrow [\![\mu_1]\!]\uparrow) \Rightarrow \ldots \Rightarrow ([\![\nu_1]\!] \Rightarrow \ldots \Rightarrow [\![\nu_n]\!] \Rightarrow [\![\mu_m]\!]\uparrow) \Rightarrow [\![\xi]\!]\uparrow$$

and $\$f_i$ is of type

$$[\![\nu_1]\!]\uparrow \Rightarrow \ldots \Rightarrow [\![\nu_{|\underline{\boldsymbol{\nu}}|}]\!]\uparrow \Rightarrow [\![\mu_i]\!]\uparrow \qquad (1 \leq i \leq m).$$

Instead, the theory M' is the result of adding the definition

$$\textbf{definition} \quad [\![Q]\!]' \ g_1 \ \ldots \ g_m = [\![t]\!][\underline{\$\boldsymbol{f}} := \mathsf{arg\_lift}\, g_1, \ldots, \mathsf{arg\_lift}\, g_m]$$

to M, which makes use of the function arg_lift defined such that

$$\mathsf{arg\_lift}\ f\ \underline{\boldsymbol{x}}\uparrow = f\ \underline{\boldsymbol{x}}.$$

(The notation $\underline{\boldsymbol{x}}\uparrow$ stands for $x_1\uparrow\ \ldots\ x_{|\underline{\boldsymbol{x}}|}\uparrow$.) Now, the definition of $[\![Q]\!]'$ is type correct. Conditions 1–3 enforce HOL's side-conditions on definitions [see Gordon and Pitts, 1993, Wenzel, 1997].

It remains to show that the rule $Q\underline{\boldsymbol{x}} \cdot \underline{\$\boldsymbol{f}}(\underline{\boldsymbol{x}}) \equiv t$ is sound, which amounts to prove

$$[\![t]\!][\underline{\$\boldsymbol{f}} := \mathsf{arg\_lift}(\lambda\underline{\boldsymbol{x}}.\ \$f_1\ \underline{\boldsymbol{x}}\uparrow), \ldots, \mathsf{arg\_lift}(\lambda\underline{\boldsymbol{x}}.\ \$f_m\ \underline{\boldsymbol{x}}\uparrow)] = [\![t]\!]. \tag{3.9}$$

Note that Condition 4 entails the validity of

$$\left(\bigwedge_{i=1}^{|\underline{\$\boldsymbol{f}}|} \forall\underline{\boldsymbol{x}}.\ \$f_i\ \underline{\boldsymbol{x}}\uparrow = \$g_i\ \underline{\boldsymbol{x}}\uparrow\right) \longrightarrow \left([\![t]\!][\underline{\$\boldsymbol{f}} := \underline{\$\boldsymbol{g}}]\right) = [\![t]\!]. \tag{3.10}$$

In particular, the conclusion of (3.10) matches (3.9). Therefore, it remains to show

$$\forall\underline{\boldsymbol{x}}.\ \$f_i\ \underline{\boldsymbol{x}}\uparrow = \mathsf{arg\_lift}\ (\lambda\underline{\boldsymbol{x}}.\ \$f_i\ \underline{\boldsymbol{x}}\uparrow)\ \underline{\boldsymbol{x}}\uparrow \qquad (1 \leq i \leq |\underline{\$\boldsymbol{f}}|). \tag{3.11}$$

Equation (3.11) follows from the definition of arg_lift.

Uniqueness of $(\mathsf{M}', [\![\cdot]\!]')$ can be shown in a similar way as for ordinary operator definitions (cf. Theorem 3.10). □

Monotonicity. The following proposition shows how to prove that a newly defined binder is monotonic.

Proposition 3.16. *Let* THY *be a theory extending* Log *and* M *a model of* THY *extending* EB_Log. *Suppose that applying the binder definition* $Q\underline{\boldsymbol{x}} \cdot \underline{\$\boldsymbol{f}}(\underline{\boldsymbol{x}}) \equiv t$ *to* THY *is admissible and results in* THY′. *Then, the binder* Q *is monotonic with respect to the unique model of* THY′ *that definitionally extends* M *iff the following proof obligation is valid with respect to* M*:*

$$\begin{aligned} &\forall\underline{\boldsymbol{x}} \cdot \$f_1(\underline{\boldsymbol{x}}) \sqsubseteq_{\mathsf{eb}} \$g_1(\underline{\boldsymbol{x}}) \\ &\ldots \\ &\forall\underline{\boldsymbol{x}} \cdot \$f_{|\underline{\$\boldsymbol{f}}|}(\underline{\boldsymbol{x}}) \sqsubseteq_{\mathsf{eb}} \$g_{|\underline{\$\boldsymbol{f}}|}(\underline{\boldsymbol{x}}) \\ \vdash\ &t \sqsubseteq_{\mathsf{eb}} t[\underline{\$\boldsymbol{f}}(\underline{\$\boldsymbol{x}}) := \underline{\$\boldsymbol{g}}(\underline{\$\boldsymbol{x}})]. \end{aligned} \tag{3.12}$$

Here, the elements of $\underline{\$\boldsymbol{g}}$ *are pairwise distinct and do not occur in* $\underline{\$\boldsymbol{f}}$.

Proof. The assertion follows from the fact that the denotation of (3.12) is equivalent to the monotonicity of Q. □

If the right-hand side of a binder definition is built from monotonic operators and binders, the resulting binder is monotonic, too.

Proposition 3.17. *Let* THY *be a theory extending* Log *and* M *a model of* THY *extending* EB_Log*. Suppose that applying the binder definition* $Q\underline{\boldsymbol{x}} \cdot \underline{\$\boldsymbol{f}}(\underline{\boldsymbol{x}}) \equiv t$ *to* THY *is admissible and results in* THY′*. If the operators and binders occurring in* t *are monotonic with respect to* M*, then so is the binder* Q *with respect to the unique model of* THY′ *that definitionally extends* M*.*

Proof. The proof proceeds by showing validity of the proof obligation of Proposition 3.16. Because of admissibility condition 4, it may be assumed without loss of generality that the elements of $\underline{\$\boldsymbol{f}}$ and $\underline{\$\boldsymbol{g}}$ have strict denotations. With this in mind, the assertion can be shown by an induction over the structure of t. □

Simplifications. The following propositions point out conditions under which the admissibility condition 4 is dispensable. First, the condition is true whenever all occurrences of operator variables in the right-hand side of the binder definition are of the form $\$f(\underline{\boldsymbol{x}})$, where $\underline{\boldsymbol{x}}$ is a sequence of ordinary variables; different occurrences of operator variables may take different ordinary variables as arguments.

Proposition 3.18. *Suppose* THY *is a theory extending* Log *and* $Q\underline{\boldsymbol{x}} \cdot \underline{\$\boldsymbol{f}}(\underline{\boldsymbol{x}}) \equiv t$ *a binder definition such that the admissibility conditions 1–3 are true. Moreover suppose that every occurrence of* $\$f_i$ *in* t*, for* $1 \leq i \leq \left|\underline{\$\boldsymbol{f}}\right|$*, is of the form* $\$f_i(\underline{\boldsymbol{y}})$*, where* $\underline{\boldsymbol{y}}$ *is a sequence of ordinary variables. Then the admissibility condition 4 is true.*

Proof. It suffices to prove the sequent

$$\begin{aligned} & \forall \underline{\boldsymbol{x}} \cdot \$f_1(\underline{\boldsymbol{x}}) \equiv_{\mathsf{eb}} \$g_1(\underline{\boldsymbol{x}}) \\ & \ldots \\ & \forall \underline{\boldsymbol{x}} \cdot \$f_{\left|\underline{\$\boldsymbol{f}}\right|}(\underline{\boldsymbol{x}}) \equiv_{\mathsf{eb}} \$g_{\left|\underline{\$\boldsymbol{f}}\right|}(\underline{\boldsymbol{x}}) \\ \vdash \quad & t \equiv_{\mathsf{eb}} t[\underline{\$\boldsymbol{f}}(\underline{\$\boldsymbol{x}}) := \underline{\$\boldsymbol{g}}(\underline{\$\boldsymbol{x}})]. \end{aligned}$$

First, repeatedly apply the following instances of $\equiv_{\mathsf{eb}}\_\mathsf{reflection}$:

$$\frac{\$f_i(\underline{\boldsymbol{y}}) \equiv_{\mathsf{eb}} \$g_i(\underline{\boldsymbol{y}})}{\$f_i(\underline{\boldsymbol{y}}) \;\equiv\; \$g_i(\underline{\boldsymbol{y}})}\ .$$

It remains to prove two kinds of sequents: sequents of the form

$$\forall \underline{\boldsymbol{x}} \cdot \$f_i(\underline{\boldsymbol{x}}) \equiv_{\mathsf{eb}} \$g_i(\underline{\boldsymbol{x}}), \ldots \vdash \forall \underline{\boldsymbol{y}}' \cdot \$f_i(\underline{\boldsymbol{y}}) \equiv_{\mathsf{eb}} \$g_i(\underline{\boldsymbol{y}}), \tag{3.13}$$

where $\underline{\boldsymbol{y}}'$ is a subsequence of $\underline{\boldsymbol{y}}$ and $1 \leq i \leq \left|\underline{\$\boldsymbol{f}}\right|$, and the sequent

$$\ldots \vdash t[\underline{\$\boldsymbol{f}}(\underline{\$\boldsymbol{x}}) := \underline{\$\boldsymbol{g}}(\underline{\$\boldsymbol{x}})] \equiv_{\mathsf{eb}} t[\underline{\$\boldsymbol{f}}(\underline{\$\boldsymbol{x}}) := \underline{\$\boldsymbol{g}}(\underline{\$\boldsymbol{x}})]. \tag{3.14}$$

The sequent (3.14) can be proved by unfolding the definition of $\equiv_{\mathsf{eb}}$ and with some propositional reasoning. The sequents (3.13) can be proved with all_R, all_L, D_var, and hyp.

Without the restriction that all occurrences of $\$f_i$ in t are of the form $\$f_i(\underline{\boldsymbol{y}})$, it would still be possible to apply $\equiv_{\mathsf{eb}}\_\mathsf{reflection}$ leading to the sequent (3.14); but the corresponding versions of (3.13) would in general be unprovable. □

The admissibility condition 4 is also true if the monotonicity proof obligation according to Proposition 3.16 is valid.

Proposition 3.19. *Suppose* THY *is a theory extending* Log *with a model* M *extending* EB_ Log*. Moreover, suppose that* $Q\underline{\boldsymbol{x}} \cdot \underline{\$\boldsymbol{f}}(\underline{\boldsymbol{x}}) \equiv t$ *is a binder definition such that the admissibility conditions 1–3 are true. If the proof obligation (3.12) of Proposition 3.16 is valid with respect to* M*, then applying* $Q\underline{\boldsymbol{x}} \cdot \underline{\$\boldsymbol{f}}(\underline{\boldsymbol{x}}) \equiv t$ *to* THY *is admissible.*

Proof. Note that if the proof obligation (3.12) of Proposition 3.16 is valid with respect to M, then so is the proof obligation (3.12) with $\underline{\$\boldsymbol{f}}$ and $\underline{\$\boldsymbol{g}}$ interchanged. From this it is straightforward to derive admissibility condition 4. □

3.2.4. Operator Variables in a Monotonic Setting

The definitional extension methods described so far are difficult to implement, as the underlying proof obligations involve operator variables and non-monotonic operators, which are not available in Rodin. A natural solution is to just implement support for operator variables and the missing (non-monotonic) operators. Such an extension would however not come cheaply, because monotonicity assumptions are hard-wired in several places, in particular in Rodin's term rewriter. Therefore, I develop an alternative approach, which essentially translates proof obligations from the general setting developed in this document into the restricted setting supported by Rodin.

In a first step, it needs to be clarified what the "restricted setting supported by Rodin" actually is. Given a theory THY, Rodin only supports terms built from monotonic symbols, namely terms in mono(THY) according to the following definition:

Definition 3.20 (Monotonic Fragment)**.** Let THY be a theory with a model M. Then, the *monotonic fragment* mono(THY) *of* THY is the set of terms built from ordinary variables and monotonic operators or binders (with respect to M) of the signature underlying THY. □

Non-monotonic operators, most significantly the well-definedness operator D, play an important role in proofs; consider for example the rules cut and ex_R. Yet, Rodin does not explicitly support non-monotonic symbols. Instead, Rodin replaces a term $f(\underline{\boldsymbol{t}})$, with a non-monotonic operator f, by an equivalent term built from only monotonic operators and binders. Such an approach obviously requires that appropriate replacements exist within mono(THY), the set of terms supported by Rodin. Using the notion of *definability* given by the following definition, the approach taken by Rodin requires that the non-monotonic operators of THY are *definable* in mono(THY).

Definition 3.21 (Definability)**.** Let L be a set of terms. The operator f is *definable* in L (*with respect to a structure* M) iff every formula $f(\underline{\boldsymbol{t}})$ with $t_1, \ldots, t_{|\underline{\boldsymbol{t}}|}$ in L is equivalent (with respect to M) to a term in L. The binder Q is *definable* in L (*with respect to a structure* M) iff every formula $Q\underline{\boldsymbol{x}} \cdot \underline{\boldsymbol{t}}$ with $t_1, \ldots, t_{|\underline{\boldsymbol{t}}|}$ in L is equivalent (with respect to M) to a term in L. □

In fact, the non-monotonic operators and binders of the theories developed in this chapter are definable in the respective monotonic fragments.

Proposition 3.22. *Let* THY *be a theory derived from* Core *using the extension methods of Sections 3.2.1–3.2.3. Then every non-monotonic operator or binder of* THY *is definable in* $\mathsf{mono}(\mathsf{THY})$.

Proof. It suffices to show how to rewrite a term t over THY that does not contain operator variables to an equivalent term in $\mathsf{mono}(\mathsf{THY})$. This can be achieved by repeatedly applying the following steps to t:

- Eliminate operators and binders introduced by ordinary operator and binder definitions using the rewrite rules introduced by their definitions.
- Eliminate the non-monotonic operators T_{eb}, $\mathsf{WT}_{\mathsf{eb}}$, $\sqsubseteq_{\mathsf{eb}}$, $\equiv_{\mathsf{eb}}$ with the respective rewrite rules of the theory Log.
- Eliminate the well-definedness operator D by applying rewrite rules of the form

  $$\mathsf{D}(f(\underline{\$\boldsymbol{x}})) \equiv \varphi \qquad \text{or} \qquad \mathsf{D}(Q\underline{\boldsymbol{x}} \cdot \underline{\$\boldsymbol{g}(\boldsymbol{x})}) \equiv \varphi.$$

  Note that such rewrite rules exist for each monotonic operator of Core, each binder of Core, and each operator introduced by a strict operator definition.

It is easy to check that this strategy terminates and yields a term in $\mathsf{mono}(\mathsf{THY})$. □

The main question of this section is how to translate – in a validity preserving manner – sequents with operator variables to sequents without. A natural solution is to represent the operator variables to be eliminated by newly introduced operators. Special care is needed to represent operator variables by *monotonic* operators and to ensure that D remains definable in the monotonic fragment of the resulting theory.

My solution is based on two ideas:

1. Introduce for every operator variable $\$f$ two new operators f' and df'. For simplicity, assume that $\$f$ takes one argument. Replace every occurrence of $\$f$ by f'. Introduce the rewrite rules

   $$\begin{aligned} \mathsf{D}(f'(\$x)) &\equiv df'(\$x), \\ \mathsf{D}(df'(\$x)) &\equiv \top \end{aligned}$$

   for D-definability.

2. The operators f' and df' are not necessarily monotonic. To recover monotonicity of, say, f', introduce a fresh operator f by $f'(\$x) \equiv f(\mathsf{D}(\$x), \$x)$. It is possible to establish D-definability and monotonicity of f by defining appropriate rewrite rules.

The following theorem gives the technical details. For simplicity, the assertion is stated for operator variables that takes exactly one argument. The lengthier version of the theorem with an arbitrary number of arguments holds as well.

Theorem 3.23. *Let* THY *be a theory extending* Log *and* $\$f \Colon \nu \rightharpoonup \mu$ *an operator variable. Suppose that* f *and* df *are non-logical symbols not used in the signature of* THY. *Define* THY′ *as the theory extending* THY *by the operators* $f \Colon \nu \rightharpoonup \mu$ *and* $df \Colon \nu \rightharpoonup \mathcal{B}$, *and the rules*

$$\mathsf{D}(f(\$\varphi, \$x)) \equiv \mathsf{T}_{\mathsf{eb}}(df(\$\varphi, \$x)), \tag{3.15}$$
$$\mathsf{D}(df(\$\varphi, \$x)) \equiv \mathsf{D}(\$\varphi) \wedge (\$\varphi \Rightarrow \mathsf{D}(\$x)), \tag{3.16}$$
$$f(\bot, \$x) \equiv f(\bot, \bullet), \tag{3.17}$$
$$df(\bot, \$x) \equiv df(\bot, \bullet). \tag{3.18}$$

Then the following is true:

1. *Every model* M *of* THY *that extends* EB_Log *can be definitionally extended to a model* M′ *of* THY′ *such that every sequent* Γ *over* THY *is valid with respect to* M *iff* $\Gamma[\$f(\$x) := f(\mathsf{D}(\$x), \$x)]$ *is valid with respect to* M′.

2. *The operators* f *and* df *are monotonic with respect to every model of* THY′ *that extends* EB_Log.

Proof. For Claim 1, define the structure M′ as an extension of M by a constant specification [Gordon and Pitts, 1993] postulating that $[\![f]\!]$ and $[\![df]\!]$ satisfy the denotations of (3.15–3.18). This constant specification requires a proof that the denotations of (3.15–3.18) are satisfiable, which is witnessed by

$$[\![f]\!]\ \varphi\ x := \bullet,$$
$$[\![df]\!]\ \varphi\ x := (\mathsf{if}\ \mathsf{WD}\ \varphi \wedge (\mathsf{T}\ \varphi \longrightarrow \mathsf{WD}\ x)\ \mathsf{then}\ \mathsf{False}{\uparrow}\ \mathsf{else}\ \bullet).$$

The structure M′ definitionally extends M, because constant specifications are definitional. By definition, M′ is a model of THY′.

Let Γ be a sequent over THY and denote $\Gamma[\$f(\$x) := f(\mathsf{D}(\$x), \$x)]$ by Γ'. If Γ is valid with respect to M, Γ' is valid with respect to M′ because M′ extends M and validity is closed under substitution. Now suppose Γ' is valid with respect to M′. By

exporting Γ' into M, it can be concluded that

$$\begin{array}{l}
\forall f\ df\ . \\
\quad (\forall \varphi\ x.\ \mathsf{WD}\ (f\ \varphi\ x) \longleftrightarrow \mathsf{T}\ (df\ \ \varphi\ x)) \wedge \\
\quad (\forall \varphi\ x.\ \mathsf{WD}\ (df\ \ \varphi\ x) \longleftrightarrow \mathsf{WD}\,\varphi \wedge (\mathsf{T}\,\varphi \longrightarrow \mathsf{WD}\,x)) \wedge \\
\quad (\forall x.\ f\ \mathsf{False}{\uparrow}\ x = f\ \mathsf{False}{\uparrow}\ \bullet) \wedge \\
\quad (\forall x.\ df\ \ \mathsf{False}{\uparrow}\ x = df\ \ \mathsf{False}{\uparrow}\ \bullet) \\
\longrightarrow \\
\quad [\![\Gamma]\!][\$f := (\lambda x.\ f\ (\mathsf{WD}\,x){\uparrow}\ x)]
\end{array} \tag{3.19}$$

is valid in M. Instantiate the outermost quantifier in (3.19) as follows:

$$\begin{array}{rl}
f\ \varphi\ x := & \mathsf{if}\ \mathsf{WD}\,\varphi \wedge (\mathsf{T}\,\varphi \longrightarrow \mathsf{WD}\,x) \\
& \mathsf{then}\ \$f\ (\mathsf{if}\ \mathsf{T}\,\varphi\ \mathsf{then}\ x\ \mathsf{else}\ \bullet) \\
& \mathsf{else}\ \bullet, \\
df\ \ \varphi\ x := & \mathsf{if}\ \mathsf{WD}\,\varphi \wedge (\mathsf{T}\,\varphi \longrightarrow \mathsf{WD}\,x) \\
& \mathsf{then}\ (\mathsf{WD}\ (\$f\ (\mathsf{if}\ \mathsf{T}\,\varphi\ \mathsf{then}\ x\ \mathsf{else}\ \bullet)))\uparrow \\
& \mathsf{else}\ \bullet.
\end{array}$$

Under this instantiation, the premises of the implication in (3.19) are true and the conclusion is equivalent to $[\![\Gamma]\!]$. Therefore, Γ is valid in M.

For Claim 2, it suffices to prove monotonicity of $[\![f]\!]$ and $[\![df]\!]$ from the denotations of (3.15–3.18), which is straightforward. In fact, the rules (3.17–3.18) have been introduced to prevent non-monotonic denotations of f and df. □

Suppose the theory THY, with the model M, has been derived from Core using the extension methods of the preceding sections. Theorem 3.23 and Proposition 3.22 show how to eliminate operator variables and non-monotonic operators and binders from an arbitrary sequent Γ of THY:

1. Repeatedly apply Theorem 3.23 to eliminate all operator variables from Γ. Call the resulting sequent $\tilde{\Gamma}$, the resulting theory THY', and the resulting model M'.

2. Eliminate the non-monotonic operators and binders from $\tilde{\Gamma}$ by applying the procedure in the proof of Proposition 3.22 and the rules (3.15–3.16) obtained from applying Theorem 3.23. Call the resulting sequent Γ'.

Note that Γ' is valid with respect to M' iff Γ is valid with respect to M. Moreover, Γ' is built from formulae in $\mathsf{mono}(\mathsf{THY}')$.

The overhead of embedding terms into the monotonic fragment of the underlying theory is exponential in the worst case. I nevertheless expect that the actual overhead is acceptable in a wide range of practical applications for the following reasons:

- The embedding into the monotonic fragment, in particular the procedure in the proof of Proposition 3.22, has much in common with the problem of unlifting, which I examine in Section 5.2; the optimizations for unlifting can be straightforwardly adopted to the embedding into the monotonic fragment.
- Since the proof obligations associated with definitional extensions are usually rather small, an exponential blow-up is often acceptable. This is also supported by the fact that definitional extensions are one-time investments.

That said, the examinations of this section show that supporting only monotonic symbols has a price, both in terms of computational complexity (at least from a theoretical perspective) and in terms of intricacy of the required algorithms.

3.3. Derived Theories

This section concludes the development of Event-B's main theory. I define the remaining operators available in Rodin based on the theory Core, using the definitional theory extension methods of the preceding section. I have developed *standard* models for the theories developed in this section, i.e., models that definitionally extend the standard model of the theory Core. These standard models can be found in [Schmalz]. Since it is known that standard models exist and are uniquely determined (cf. Section 3.2), I refrain from defining these models explicitly within this document.

All operators and binders introduced in this section are proved to be monotonic. Proof obligations that can be discharged by Propositions 3.12, 3.14, 3.17, or 3.18 are omitted. I have proved the remaining proof obligations by using Isabelle/HOL.

3.3.1. The Theory Bool

The theory Bool definitionally extends Set_0 with the following derived operators related to the booleans:

$$\begin{aligned} \mathsf{BOOL} &\equiv \{x \mathbin{⦂} \mathcal{B} \mid \top\}, \\ \mathsf{bool}(x) &\equiv x, \\ \mathsf{TRUE} &\equiv \top, \\ \mathsf{FALSE} &\equiv \bot. \end{aligned}$$

The operators bool, TRUE, and FALSE are redundant in the setting of this document. I have included them for compatibility with Rodin; in Rodin these operators are required because of the distinction between "predicates" and "expressions" (cf. Section 2.4).

3.3.2. The Theory Set

The theory Set definitionally extends Set_0. It introduces the following operators by strict operator definitions:

$$
\begin{array}{lcl}
x \notin R & \sqsubseteq & \neg(x \in R), \\
R \subseteq S & \sqsubseteq & \forall x \cdot x \in R \Rightarrow x \in S, \\
R \not\subseteq S & \sqsubseteq & \neg R \subseteq S, \\
R \subset S & \sqsubseteq & R \subseteq S \land \neg S \subseteq R, \\
R \not\subset S & \sqsubseteq & \neg R \subset S, \\
\varnothing & \sqsubseteq & \{x \mid \bot\}, \\
R \cup S & \sqsubseteq & \{x \mid x \in R \lor x \in S\}, \\
R \cap S & \sqsubseteq & \{x \mid x \in R \land x \in S\}, \\
R \setminus S & \sqsubseteq & \{x \mid x \in R \land x \notin S\}, \\
\mathbb{P}(R) & \sqsubseteq & \{S \mid S \subseteq R\}, \\
\mathbb{P}_1(R) & \sqsubseteq & \{S \mid S \neq \varnothing \land S \subseteq R\}, \\
\mathsf{union}(R) & \sqsubseteq & \{x \mid \exists S \cdot S \in R \land x \in S\}, \\
\mathsf{inter}(R) & \sqsubseteq & \{x \mid \forall S \cdot S \in R \Rightarrow x \in S\} \\
 & \text{with} & \mathsf{D}(\mathsf{inter}(R)) \equiv R \neq \varnothing.^{5}
\end{array}
$$

The theory Set also introduces several infinite families of operators and binders, which intuitively represent operators that take an arbitrary number of arguments or binders that bind an arbitrary number of variables.

$$
\begin{array}{lcl}
\{\underline{\boldsymbol{x}} \cdot \$\varphi(\underline{\boldsymbol{x}}) \mid \$f(\underline{\boldsymbol{x}})\} & \equiv & \mathsf{if}\ \forall \underline{\boldsymbol{x}} \cdot \mathsf{D}(\$\varphi(\underline{\boldsymbol{x}})) \land (\$\varphi(\underline{\boldsymbol{x}}) \Rightarrow \mathsf{D}(\$f(\underline{\boldsymbol{x}}))) \\
 & & \mathsf{then}\ \{z \mid \exists \underline{\boldsymbol{x}} \cdot \$\varphi(\underline{\boldsymbol{x}}) \land z = \$f(\underline{\boldsymbol{x}})\} \\
 & & \mathsf{else}\ \bullet, \\
 & & \text{where } z \text{ does not appear in } \underline{\boldsymbol{x}}, \text{ and } |\underline{\boldsymbol{x}}| \geq 1, \\
\{\,\} & \sqsubseteq & \varnothing, \\
\{\underline{\boldsymbol{x}}\} & \sqsubseteq & \{y \mid y = x_1 \lor \cdots \lor y = x_{|\underline{\boldsymbol{x}}|}\}, \qquad (|\underline{\boldsymbol{x}}| \geq 1), \\
\bigcup \underline{\boldsymbol{x}} \cdot \$\varphi(\underline{\boldsymbol{x}}) \mid \$R(\underline{\boldsymbol{x}}) & \equiv & \mathsf{union}(\{\underline{\boldsymbol{x}} \cdot \$\varphi(\underline{\boldsymbol{x}}) \mid \$R(\underline{\boldsymbol{x}})\}), \qquad (|\underline{\boldsymbol{x}}| \geq 1), \\
\bigcap \underline{\boldsymbol{x}} \cdot \$\varphi(\underline{\boldsymbol{x}}) \mid \$R(\underline{\boldsymbol{x}}) & \equiv & \mathsf{inter}(\{\underline{\boldsymbol{x}} \cdot \$\varphi(\underline{\boldsymbol{x}}) \mid \$R(\underline{\boldsymbol{x}})\}), \qquad (|\underline{\boldsymbol{x}}| \geq 1), \\
\mathsf{partition}(R) & \sqsubseteq & R = \varnothing,
\end{array}
$$

[5] The empty set has been excluded from the domain of inter because $\{x \mid \top\}$ is not a legal term of Zermelo-Fraenkel set theory. Given that Event-B has a Hindley-Milner style type system (like higher-order logic and unlike Zermelo-Fraenkel set theory), it would be more natural to define $\mathsf{inter}(\varnothing)$ to be $\{x \mid \top\}$.

$$
\begin{array}{lcl}
\mathsf{partition}(R, S) & \sqsubseteq & S = R, \\
\mathsf{partition}(R, S_1, \ldots, S_n) & \sqsubseteq & S_1 \cup \cdots \cup S_n = R \land \\
& & S_1 \cap S_2 = \varnothing \land \cdots \land S_1 \cap S_n = \varnothing \land \\
& & S_2 \cap S_3 = \varnothing \land \cdots \land S_2 \cap S_n = \varnothing \land \\
& & \ldots \land \\
& & S_{n-1} \cap S_n = \varnothing.
\end{array}
$$

It is straightforward to prove the following semantic properties of the *set comprehension* binder $\{\underline{\boldsymbol{x}} \cdot \_ \mid \_\}$:

Lemma 3.24. *Set comprehension is definite and monotonic with respect to the standard model of* Set.

Concrete Syntax. Event-B's concrete syntax offers alternative notations for the binders introduced by Set:

Concrete Syntax	Abstract Syntax
$\{t \mid \varphi\}$	$\{\underline{\boldsymbol{x}} \cdot \varphi \mid t\}$
$\bigcup t \mid \varphi$	$\bigcup \underline{\boldsymbol{x}} \cdot \varphi \mid t$
$\bigcap t \mid \varphi$	$\bigcap \underline{\boldsymbol{x}} \cdot \varphi \mid t$

Here, the sequence $\underline{\boldsymbol{x}}$ is nonempty, has pairwise distinct elements, and contains exactly the free variables of t. Note that the notation $\{x \mid \varphi\}$ is ambiguous, as it stands both for $(\mathsf{collect}\, x \cdot \varphi)$ and $\{x \cdot \varphi \mid x\}$. This ambiguity is harmless, as $(\mathsf{collect}\, x \cdot \varphi)$ and $\{x \cdot \varphi \mid x\}$ are equivalent with respect to every model of Set.

3.3.3. The Theory Rel

The theory Rel definitionally extends Choice ∪ Set ∪ Prod by several operator definitions concerning relations, i.e., terms of type $\mathcal{P}(\nu \star \mu)$. Variables ranging over relations are denoted by r and s.

The following operators of Rel determine common properties of relations:

$$
\begin{array}{lcl}
\mathsf{dom}(r) & \sqsubseteq & \{x \mid \exists y \cdot x \mapsto y \in r\}, \\
\mathsf{ran}(r) & \sqsubseteq & \{y \mid \exists x \cdot x \mapsto y \in r\}, \\
\mathsf{total}^\dagger(r, R) & \sqsubseteq & R \subseteq \mathsf{dom}(r), \\
\mathsf{surjective}^\dagger(r, R) & \sqsubseteq & R \subseteq \mathsf{ran}(r), \\
\mathsf{functional}^\dagger(r) & \sqsubseteq & \forall x, y_1, y_2 \cdot x \mapsto y_1 \in r \land x \mapsto y_2 \in r \Rightarrow y_1 = y_2, \\
\mathsf{injective}^\dagger(r) & \sqsubseteq & \forall x_1, x_2, y \cdot x_1 \mapsto y \in r \land x_2 \mapsto y \in r \Rightarrow x_1 = x_2.
\end{array}
$$

The theory Rel introduces operators determining the relational and functional images of relations:

$$\begin{array}{lcl} \mathsf{relimg}(r, R) & \sqsubseteq & \{y \mid \exists x \cdot x \in R \land x \mapsto y \in r\}, \\ \mathsf{funimg}(r, x) & \sqsubseteq & \mathsf{some}\, y \cdot y \in \mathsf{relimg}(r, \{x\}) \\ & \text{with} & \mathsf{D}(\mathsf{funimg}(r, x)) \equiv \mathsf{functional}(r) \land x \in \mathsf{dom}(r). \end{array}$$

The definition of funimg gives rise to a non-trivial proof obligation, which I have proved using Isabelle/HOL. In concrete syntax, $\mathsf{relimg}(t, u)$ is denoted by $t[u]$ and $\mathsf{funimg}(t, u)$ by $t(u)$.

The following operators of Rel create or modify relations:

$$\begin{array}{lcl} R \times S & \sqsubseteq & \{x \mapsto y \mid x \in R \land y \in S\}, \\ \mathsf{id} & \sqsubseteq & \{x \mapsto x \mid \top\}, \\ \mathsf{prj}_1 & \sqsubseteq & \{(x_1 \mapsto x_2) \mapsto x_1 \mid \top\}, \\ \mathsf{prj}_2 & \sqsubseteq & \{(x_1 \mapsto x_2) \mapsto x_2 \mid \top\}, \\ r\sim & \sqsubseteq & \{y \mapsto x \mid x \mapsto y \in r\}, \\ r\,;\,s & \sqsubseteq & \{x \mapsto z \mid \exists y \cdot x \mapsto y \in r \land y \mapsto z \in s\}, \\ s \circ r & \sqsubseteq & r\,;\,s, \\ R \lhd r & \sqsubseteq & \{x \mapsto y \mid x \mapsto y \in r \land x \in R\}, \\ R \lhd\!\!\!\!- r & \sqsubseteq & \{x \mapsto y \mid x \mapsto y \in r \land x \notin R\}, \\ r \rhd R & \sqsubseteq & \{x \mapsto y \mid x \mapsto y \in r \land y \in R\}, \\ r \rhd\!\!\!\!- R & \sqsubseteq & \{x \mapsto y \mid x \mapsto y \in r \land y \notin R\}, \\ r \lhd\!\!\!\!- s & \sqsubseteq & (\mathsf{dom}(s) \lhd\!\!\!\!- r) \cup s, \\ r \otimes s & \sqsubseteq & \{x \mapsto (y \mapsto z) \mid x \mapsto y \in r \land x \mapsto z \in s\}, \\ r \parallel s & \sqsubseteq & \{(x_1 \mapsto y_1) \mapsto (x_2 \mapsto y_2) \mid x_1 \mapsto x_2 \in r \land y_1 \mapsto y_2 \in s\}. \end{array}$$

Finally, Rel introduces operators that generate sets of relations:

$$\begin{array}{lcl} R \leftrightarrow S & \sqsubseteq & \mathbb{P}(R \times S), \\ R \nleftrightarrow S & \sqsubseteq & \{r \mid r \in R \leftrightarrow S \land \mathsf{total}(r, R)\}, \\ R \leftrightarrow\!\!\!\!\!\rightarrow S & \sqsubseteq & \{r \mid r \in R \leftrightarrow S \land \mathsf{surjective}(r, S)\}, \\ R \nleftrightarrow\!\!\!\!\!\rightarrow S & \sqsubseteq & \{r \mid r \in R \leftrightarrow S \land \mathsf{total}(r, R) \land \mathsf{surjective}(r, S)\}, \\ R \nrightarrow S & \sqsubseteq & \{r \mid r \in R \leftrightarrow S \land \mathsf{functional}(r)\}, \\ R \rightarrow S & \sqsubseteq & \{r \mid r \in R \leftrightarrow S \land \mathsf{functional}(r) \land \mathsf{total}(r, R)\}, \end{array}$$

[6] Some authors [e.g. Abrial, 2010] write r^{-1} instead of $r\sim$. Rodin uses the notation $r\sim$.

$$
\begin{array}{lcl}
R \rightarrowtail S & \sqsubseteq & \{r \mid r \in R \leftrightarrow S \wedge \mathsf{functional}(r) \wedge \mathsf{injective}(r)\}, \\
R \rightarrowtail S & \sqsubseteq & \{r \mid r \in R \leftrightarrow S \wedge \mathsf{functional}(r) \wedge \mathsf{injective}(r) \wedge \mathsf{total}(r, R)\}, \\
R \twoheadrightarrow S & \sqsubseteq & \{r \mid r \in R \leftrightarrow S \wedge \mathsf{functional}(r) \wedge \mathsf{surjective}(r, S)\}, \\
R \rightarrow\!\!\!\!\rightarrow S & \sqsubseteq & \{r \mid r \in R \leftrightarrow S \wedge \mathsf{functional}(r) \wedge \mathsf{total}(r, R) \wedge \mathsf{surjective}(r, S)\}, \\
R \rightarrowtail\!\!\!\!\rightarrow S & \sqsubseteq & \{r \mid r \in R \leftrightarrow S \wedge \mathsf{functional}(r) \wedge \mathsf{injective}(r) \wedge \\
 & & \mathsf{total}(r, R) \wedge \mathsf{surjective}(r, S)\}.
\end{array}
$$

Lambda Abstractions. Event-B's *lambda abstraction* $(\lambda t \cdot \varphi \mid u)$ is a construct of concrete syntax: $(\lambda t \cdot \varphi \mid u)$ is a short-hand for $\{\underline{\boldsymbol{x}} \cdot \varphi \mid t \mapsto u\}$, where $\underline{\boldsymbol{x}}$ contains exactly the free variables of t and has pairwise distinct elements. The term t has to be built from pairwise distinct ordinary variables and the pair operator $\mapsto$ [p. 14 Metayer and Voisin, 2009].

3.3.4. The Theory Int

The theory Int definitionally extends $\mathsf{Int}_0 \cup \mathsf{Rel}$ with derived operators related to the integers. The definitions of this section give rise to several non-trivial proof obligations, which I have proved using Isabelle/HOL.

To begin with, Int introduces several variants of $\leq$:

$$
\begin{array}{lcl}
x \geq y & \sqsubseteq & y \leq x, \\
x < y & \sqsubseteq & x \leq y \wedge x \neq y, \\
x > y & \sqsubseteq & y < x.
\end{array}
$$

Next, Int introduces *numerals* such as 2, 3, 4:

$$
\begin{array}{lcl}
2 & \sqsubseteq & 1 + 1, \\
3 & \sqsubseteq & 2 + 1, \\
4 & \sqsubseteq & 3 + 1, \\
 & \ldots &
\end{array}
$$

The theory Int also includes the following operators:

$$
\begin{array}{lcl}
\mathbb{Z} & \sqsubseteq & \{x \otimes \mathcal{Z} \mid \top\}, \\
\mathbb{N} & \sqsubseteq & \{x \mid x \geq 0\}, \\
\mathbb{N}_1 & \sqsubseteq & \{x \mid x > 0\}, \\
x \mathinner{..} z & \sqsubseteq & \{y \mid x \leq y \wedge y \leq z\}, \\
x - y & \sqsubseteq & x + (-y),
\end{array}
$$

$$\begin{array}{lcl} \mathsf{succ} & \sqsubseteq & \{x \mapsto x+1 \mid \top\}, \\ \mathsf{pred} & \sqsubseteq & \mathsf{succ}\sim. \end{array}$$

The above operators are used to express the induction rule

$$\frac{\vdash\ 0 \in \$R \qquad \vdash\ \mathsf{succ}[\$R] \subseteq \$R}{\vdash\ \mathbb{N} \subseteq \$R}\ \mathsf{Induction}.$$

Maximum is defined by

$$\begin{array}{lcl} \mathsf{max}(R) & \sqsubseteq & -\,\mathsf{min}(\{x \mid -x \in R\}) \\ & \text{with} & \mathsf{D}(\mathsf{max}(R)) \equiv R \neq \varnothing \wedge (\exists b \cdot \forall y \cdot y \in R \Rightarrow y \leq b), \end{array}$$

and division and modulo by

$$\begin{array}{lcl} x \div y^{7} & \sqsubseteq & \mathsf{if}\ \ x \geq 0 \Leftrightarrow y \geq 0 \\ & & \mathsf{then}\ \ \mathsf{max}(\{z \mid \mathsf{if}\ x \geq 0\ \mathsf{then}\ y * z \leq x\ \mathsf{else}\ y * z \geq x\}) \\ & & \mathsf{else}\ \ \mathsf{min}(\{z \mid \mathsf{if}\ x \geq 0\ \mathsf{then}\ y * z \leq x\ \mathsf{else}\ y * z \geq x\}) \\ & \text{with} & \mathsf{D}(x \div y) \equiv y \neq 0, \\ x\ \mathsf{mod}\ y & \sqsubseteq & \mathsf{some}\ z \cdot y * (x \div y) + z = x \\ & \text{with} & \mathsf{D}(x\ \mathsf{mod}\ y) \equiv x \geq 0 \wedge y > 0. \end{array}$$

Remark 3.25. Some readers may wonder why division and modulo have different domains. The domains used to be the same in earlier Rodin versions, i.e., $\mathsf{D}(x \bmod y) \equiv y \neq 0$ was sound. However, Rodin's ML prover, which has originally been designed for classical B, is possibly unsound with respect to this ancient semantics of modulo. To ensure soundness of ML, the Rodin developers restricted the domain of modulo to the one of classical B. Note that the document by Metayer and Voisin [2009] incorrectly gives the ancient domain of modulo. □

For the sake of illustration, Int includes the following rules about division and modulo:

$$\frac{}{\vdash\ \$x\ \mathsf{mod}\ \$y \in 0\,..\,\$y-1}\ \mathsf{mod\_range},$$

$$(-\$x) \div \$y\ \equiv\ -(\$x \div \$y) \quad \mathsf{div\_sign}_1,$$

[7] Abrial [2010, p. 333] postulates that $c = a \div b$ can be rewritten to $\exists r \cdot (r \in \mathbb{N} \wedge r < b \wedge a = c * b + r)$. I do not adopt his postulate because it cannot be transformed into a sound directed rewrite rule, at least if the integers are interpreted according to their standard semantics. To see the problem, instantiate b with -1.

$$\$x \div (-\$y) \;\equiv\; -(\$x \div \$y) \quad \mathsf{div\_sign_2}.$$

Exponentiation is defined by

$$\begin{array}{rcl} x \mathbin{\widehat{}} y & \sqsubseteq & \Big(\mathsf{some}\, r \cdot r \in \mathbb{N} \to \mathbb{Z} \wedge r(0) = 1 \wedge \\ & & \qquad \big(\forall z \cdot z \in \mathbb{N} \Rightarrow r(z+1) = x * r(z)\big)\Big)(y) \\ \text{with} & \mathsf{D}(x \mathbin{\widehat{}} y) \equiv x \geq 0 \wedge y \geq 0. & \end{array}$$

To clarify the definition of exponentiation, I include the following rules into Int[8]:

$$\$x \mathbin{\widehat{}} 0 \;\sqsubseteq\; 1 \quad \mathsf{expn\_base}, \qquad \frac{}{\vdash \$x \mathbin{\widehat{}} (\$y + 1) = (\$x \mathbin{\widehat{}} \$y) * \$x}\ \mathsf{expn\_step}.$$

Finally, Int provides the operators

$$\begin{array}{rcl} \mathsf{finite}(R) & \sqsubseteq & (\exists x \cdot R \rightarrowtail\!\!\!\!\!\!\rightarrow 1 \mathinner{..} x \neq \varnothing), \\ \mathsf{card}(R) & \sqsubseteq & \mathsf{some}\, x \cdot x \in \mathbb{N} \wedge R \rightarrowtail\!\!\!\!\!\!\rightarrow 1 \mathinner{..} x \neq \varnothing \\ & \text{with} & \mathsf{D}(\mathsf{card}(R)) \equiv \mathsf{finite}(R). \end{array}$$

3.3.5. The Theory EventB

Event-B's main theory EventB results from merging Bool and Int, and includes all symbols supported by Rodin. It is straightforward to merge the standard models of Bool and Int to a model of EventB that definitionally extends EB_Core; I call it the *standard model* of EventB.

3.4. Comparison to Other Expositions on Event-B's Logic

Mehta [2008] carefully describes the untyped first-order fragment of Event-B's logic, which essentially covers the monotonic fragment of Log without boolean variables and with exactly one non-boolean type. The starting point of Mehta's presentation is classical first-order logic. He introduces a syntactic transformation $\mathcal{D}$ that maps every term to a formula; the term t is said to be *well-defined* iff $\mathcal{D}(t)$ is valid. So well-definedness is handled in an additional layer on top of classical first-order logic. In a last step, Mehta introduces a non-classical notion of sequent as well as corresponding

[8] Abrial [2010, p. 332] postulates $\forall a \cdot a \mathbin{\widehat{}} 0 = \mathsf{succ}(0)$ and $\forall a, b \cdot a \mathbin{\widehat{}} \mathsf{succ}(b) = (a \mathbin{\widehat{}} b) * a$. I have not adopted these postulates, because they are invalid if the integers are interpreted according to their standard semantics. To see the problem, instantiate a with 0 and b with -1.

inference rules, which he proves to be sound by an embedding into classical first-order logic.

My presentation differs from Mehta's by making partial functions and the well-definedness operator D a genuine part of the logic. Mehta's non-classical sequents coincide with the sequents of this document (interpreted in the SW-semantics), and Mehta's inference rules are in fact contained in Log. Mehta therefore describes the same notion of validity as this document, restricted to a fragment of Event-B's logic and using an alternative presentation.

The document by Metayer and Voisin [2009] introduces (a slightly outdated version of) Event-B's concrete syntax and moreover seems to define the domains of the various operators and binders. For some operators and binders however, this document merely defines approximations of their domains, as has been decided during a meeting of the Deploy project [telcoWD]. This concerns the operators $\wedge$, $\vee$ and $\Rightarrow$, and the binders $\forall$ and $\exists$. The actual domains are defined in [Mehta, 2008] (and therefore also in this thesis). The underlying approximation technique is elaborated in Section 4.3. Except for the domain of modulo (see Remark 3.25), the domains according to the standard model of EventB coincide with the domains given in [Metayer and Voisin, 2009].

The book by Abrial [2010] provides an introduction to Event-B for Rodin users without giving the foundations of the logic. On the one hand, Abrial's "mathematical language" covers most operators and binder supported by Rodin and has therefore been useful in providing a first idea of the intended semantics. On the other hand, Abrial omits important aspects of the logic, including types and the treatment of partial functions. If the given information is taken literally, Abrial defines Event-B's logic as a version of naive set theory, which is known to be inconsistent. On an informal level, the version of Event-B's "mathematical language" illustrated in Abrial's book has many similarities with the logic of Event-B defined in this document. On a formal level, it is hard to establish a precise connection.

4. Impact of Design Decisions

The focus so far has been on the question of what the logic underlying Event-B is. Various design decisions, typically related to partial functions, have been made without explaining the underlying reasons (or lack thereof). There is often no agreement in the literature on which choice is "best" simply because different authors have different applications in mind. In this chapter, I step back and analyze the impact of the design decisions made during the evolution of Event-B's logic.

I start by evaluating the impact of directed rewriting (Section 4.1). Section 4.2 gives an overview of the main design decisions and points out several dependencies. One decision concerns the semantics of boolean connectives, and universal and existential quantifiers; Rodin implements an intricate solution that has often been misunderstood and is not documented elsewhere[1]. This issue is handled in Section 4.3. Finally, I compare Event-B's approach to partial functions to other approaches in the literature (Section 4.4).

4.1. Directed Rewriting

Recall that the directed rewrite rule $t \sqsubseteq u$ is sound iff the corresponding symmetric rewrite rule

$$\frac{\mathsf{D}(t)}{t \equiv u}$$

is sound. The advantage of directed rewriting over symmetric rewriting is that fewer well-definedness conditions (namely $\mathsf{D}(t)$ in the example) need to be checked during proofs.

There is of course no advantage if $t \equiv u$ is sound. I therefore define the notion of a *truly* directed rewrite rule: a directed rewrite rule

$$\frac{\varphi}{t \sqsubseteq u}\ \mathsf{r}$$

is *truly* directed iff r is sound and

$$\frac{\varphi}{t \equiv u}\ \mathsf{r}'$$

is unsound.

[1] That said, some readers may recognize that the underlying technique originates from classical B [Abrial and Mussat, 2002].

Table 4.1.: Statistics on truly directed rewrite rules in Rodin

	truly directed	symmetric	total
unconditional	165	288	453
conditional	42	11	53
total	207	299	506

Examples of truly directed rewrite rules include:

$$\$x \in \varnothing \sqsubseteq \bot, \tag{4.1}$$

$$\$x \in \$R \cap \$S \sqsubseteq \$x \in \$R \land \$x \in \$S, \tag{4.2}$$

$$0 \in \{x \mid \$\varphi(x)\} \sqsubseteq \$\varphi(0), \tag{4.3}$$

$$\$x \bmod \$x \sqsubseteq 0. \tag{4.4}$$

Each of these rules points to an entire class of truly directed rules:

- The rule (4.1) is built from strict and definite operators and binders and its left-hand side contains an operator variable that is not contained in its right-hand side.
- The rule (4.2) is an example of a rule with a "mismatch" between strict ($\cap$) and lazy ($\land$) operators.
- The rule (4.3) is truly directed due to the smashedness of an involved binder.
- The rule (4.4) is an example of a rule with a partial operator on the left-hand side.

This gives the impression that truly directed rules are quite common.

To substantiate this impression, I have analyzed the rewrite rules available in Rodin. New rules for Rodin's term rewriter are chosen and implemented based on the requests of Rodin users. The set of available rules therefore reflects which rules are important in common applications of Event-B. The details of the analysis can be found in Appendix A, and a summary is given in Table 4.1.

Overall, 36% of Rodin's unconditional rewrite rules and 79% of Rodin's conditional rewrite rules are truly directed. Thus, in a significant number of cases, directed rewriting makes conditional rewrite rules unconditional or makes the condition of a rewrite rule easier to solve. I therefore conclude that directed rewriting constitutes an important optimization of Rodin's term rewriter.

The reader may have the impression that directed rewriting mainly compensates for problems introduced by the fact that Event-B's logic explicitly supports partial functions. But this is not entirely true. In logics with only total functions it is quite

common to approximate partial functions by underspecified total functions. In such a logic, $x \bmod 0$ denotes an unspecified integer. Therefore $x \bmod x$ is equivalent to 0 only if $x \neq 0$. In Event-B, the condition $x \neq 0$ can be avoided by restating the rule as $\$x \bmod \$x \sqsubseteq 0$. Thus, directed rewriting not only compensates for problems introduced by explicit partiality, but also makes rules unconditional that are commonly conditional in logics of total functions. Rodin implements 35 such rules.

Safety. It is highly desirable that automated proof tactics are *safe*, i.e., they never transform a valid sequent into an invalid one during a backwards proof. Unsafe tactics need to be applied with care because they may drive the proof attempt into a dead end. Directed rewriting is unsafe in general, as the rule $\$x \bmod \$x \sqsubseteq 0$ transforms the valid sequent $\vdash 0 \bmod 0 = 1$ into the invalid sequent $\vdash 0 = 1$. It is however quite difficult to reproduce such an unsafe behavior of directed rewriting in Rodin; the question is therefore under which preconditions directed rewriting is safe. It suffices to consider the case of unconditional rewrite rules, because it makes sense to assume that conditional rules are applied only if their conditions can be solved.

Recall that sequents are interpreted in SW-semantics by default, and "valid" is consequently short for SW-valid. For the purposes of this section, it is necessary to recall WS-semantics. A tactic is *WS-safe* iff it never transforms a WS-valid sequent into a WS-invalid one during a backwards proof. Unconditional directed rewriting is not SW-safe, but it is WS-safe.

From this observation, it can be derived that directed rewriting is safe under certain preconditions.

Theorem 4.1. *Assume that directed rewriting is only applied to sequents* Γ *such that*

$$\Gamma \textit{ is SW-valid implies that } \Gamma \textit{ is WS-valid.} \tag{4.5}$$

Then, directed rewriting is safe.

Proof. Suppose the sequent under consideration is SW-valid. By assumption, it is WS-valid as well. Hence, the result of directed rewriting is WS-valid. As WS-validity implies SW-validity, the result of directed rewriting is SW-valid. □

How strong is Condition (4.5) of Theorem 4.1? I separately address the questions of how it can be established for proof obligations and how it can be preserved for intermediate sequents of a proof.

Note that Condition (4.5) is satisfied by sequents of the form

$$\mathsf{D}(\psi_1), \ldots, \mathsf{D}(\psi_{|\underline{\boldsymbol{\psi}}|}), \mathsf{D}(\varphi), \underline{\boldsymbol{\psi}} \vdash \varphi.$$

Adding the well-definedness conditions $\mathsf{D}(\psi_i)$, for $1 \leq i \leq |\underline{\boldsymbol{\psi}}|$, or $\mathsf{D}(\varphi)$ to the sequent $\underline{\boldsymbol{\psi}} \vdash \varphi$ is sound. Thus, Condition (4.5) can be established for proof obligations by a slight modification of Rodin's proof obligation generator.

Next, note that (4.5) is preserved by WS-safe tactics. In particular, it is preserved by unconditional directed and symmetric rewriting. Inference rules are not WS-safe in general. Clearly, an unsafe inference rule such as cut is not WS-safe either. If the user applies unsafe inference rules, directed rewriting subsequently becomes unsafe as well; I do not view this as a problem, as the user should anyway be careful when applying unsafe rules like cut.

Inference rules can also be safe and WS-unsafe:

$$\frac{\vdash \bullet}{\vdash \top}\ .$$

But it is always possible to transform (in a soundness preserving manner) a safe inference rule into a WS-safe one by adding well-definedness conditions to the antecedents. Such a transformation is actually unnecessary for the inference rules available in Rodin: by inspecting the list of available rules[2], I have observed that every safe inference rule of Rodin is WS-safe as well. So Condition (4.5) is preserved if the user sticks to applying rewrite rules and safe inference rules.

In summary, directed rewriting is unsafe in general; but after slight modifications of the proof obligation generator and the available inference rules, directed rewriting is safe whenever only safe inference rules are applied within proofs. For Rodin, modifications of the proof calculus are unnecessary.

4.2. Dependencies Between Design Decisions

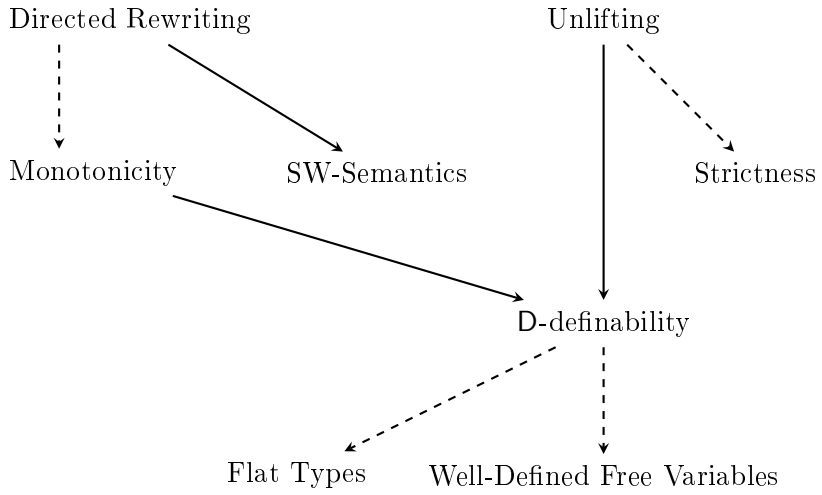
Figure 4.1 illustrates the high-level structure of the dependencies between features of Event-B's logic. In the following, I address them one by one.

Directed Rewriting. Directed rewriting has strongly influenced the design of Event-B's logic. Soundness of directed rewriting rests on the assumption that sequents are interpreted in SW-semantics, and only the arguments of monotonic operators and binders may be rewritten.

SW-Semantics. If sequents are not interpreted in SW-semantics, applying directed rewrite rules to the hypotheses or the goal of a sequent is unsound. Hence, SW-semantics is a precondition of directed rewriting. As pointed out in Section 2.3.1, there is no agreement in the literature on which sequent semantics is "best". I view directed rewriting as a novel argument in favor of SW-semantics.

Monotonicity. *Monotonicity* refers to the fact that Rodin supports only the monotonic fragment of a given theory. If non-monotonic symbols are made available, directed rewriting still remains sound in the monotonic fragment. So by admitting

[2]Statements about the rules available in Rodin refer to the rules published in `http://wiki.event-b.org` on April 5, 2011. Appendix A gives more details.



A solid arrow from P to Q indicates that Q is *necessary* to establish P. A dashed arrow from P to Q indicates that Q is *helpful* to establish P.

Figure 4.1.: Dependencies between features of Event-B's logic

non-monotonic symbols, the applicability of directed rewriting is not restricted. A monotonicity postulate simplifies the implementation of rewriting, as there is no need to implement a symmetric rewriting engine to cope with non-monotonic symbols. But directed rewriting can very well coexist with non-monotonic symbols.

Unlifting. Although Event-B's logic features explicit partial functions, Rodin has always been connected to theorem provers that do not. Rodin therefore implements an embedding, called *unlifting*, from Event-B's logic of partial functions into the fragment of Event-B's logic that supports only total functions. Algorithms for unlifting have been proposed in several contexts [see e.g. Abrial and Mussat, 2002, Berezin et al., 2005, Darvas et al., 2008, Woodcock et al., 2009]. In Section 5.2, I describe my implementation of unlifting for Event-B.

Strictness. An important subproblem of unlifting is to eliminate D from well-definedness conditions $\mathsf{D}(t)$. Strict operators f are convenient in this respect, because $\mathsf{D}(f(\underline{\$\boldsymbol{x}}))$ is equivalent to $\mathsf{D}(\$x_1) \wedge \cdots \wedge \mathsf{D}(\$x_{|\underline{\$\boldsymbol{x}}|})$; so D can be eliminated from $\mathsf{D}(f(\underline{\$\boldsymbol{x}}))$ without substantially increasing the size of the formula.

Using the example of intersection, I will illustrate the "price" of making strict operators lazy. Consider the intersection $\cap_{ll}$ that is lazy in both arguments:

$$\$R \cap_{ll} \$S \equiv \{x \mid x \in \$R \wedge x \in \$S\}.$$

It deviates from the usual strict intersection by the fact that

$$\mathsf{D}(\$R \cap_{ll} \$S) \equiv (\mathsf{D}(\$R) \wedge \mathsf{D}(\$S)) \vee (\mathsf{D}(\$R) \wedge \$R = \varnothing) \vee (\mathsf{D}(\$S) \wedge \$S = \varnothing) \quad (4.6)$$

is sound. If the rule (4.6) is used for unlifting, occurrences of $\cap_{ll}$ contribute to an exponential blowup.

Alternatively, one could introduce a version of intersection $\cap_{sl}$ that is strict in its first and lazy in its second argument. Thus, the rule

$$\mathsf{D}(\$R \cap_{sl} \$S) \equiv \mathsf{D}(\$R) \wedge (\$R \neq \varnothing \Rightarrow \mathsf{D}(\$S)) \tag{4.7}$$

is sound. If the rule (4.7) is used for unlifting, occurrences of $\cap_{sl}$ contribute to a quadratic blowup.

For the (lazy) operators $\wedge$, $\vee$, and $\Rightarrow$, a method is known that limits the exponential blowup that arises in a naive implementation of unlifting [Darvas et al., 2008]. This method can possibly be adopted to other lazy operators as well. Yet, a simpler measure against blowups during unlifting is to define as many operators as possible to be strict.

D-definability. The term D-*definability* stands for the fact that D is definable in the underlying set of available terms. Proposition 3.22 points out that D-definability holds for the monotonic fragment of every theory derived from Core with the extension methods of Section 3.2, and therefore for the set of terms supported by Rodin.

Well-definedness conditions $\mathsf{D}(t)$ naturally arise during proofs: consider the rules cut and $\mathsf{all_L}$ of the theory Log. Yet, Rodin does not explicitly support the operator D. This problem is solved with the machinery in Section 3.2.4, which rests on D-definability.

The unlifting algorithms known to me require D-definability as well. This is not surprising, as it is unclear how to unlift the formula $\mathsf{D}(f(t))$, if the domain of f is unknown. It should be noted that D-definability can often be achieved by applying the following receipt: "If D cannot be eliminated from a formula $\mathsf{D}(t)$, then give it a new name."

Flat Types. As pointed out in Section 2.2, Event-B's types are interpreted in a flat way. For example, if t is of type $\mathcal{P}(\alpha)$, its denotation is of type $(\alpha \Rightarrow \mathsf{bool})\uparrow$ and not of type $\alpha \Rightarrow \mathsf{bool}\uparrow$ or $\alpha\uparrow \Rightarrow \mathsf{bool}\uparrow$. Non-flat interpretations of types make it more complicated to achieve D-definability. For the sake of illustration, consider a non-flat function variable f whose denotation is of type $\nu \Rightarrow \mu\uparrow$. If the domain of f is unknown, it is impossible to eliminate D from terms like $\mathsf{D}(f(x))$.

In practice, this difficulty can be overcome to some extent. The trick is to exclude non-flat functions (with unknown domains) from the logic and to use total functions over restricted (and definable) domains instead. The formula $\mathsf{D}(f(x))$ is then equivalent to $x \in \mathcal{DOM}_f$, where $\mathcal{DOM}_f$ is the term defining the domain of f. PVS [Owre and Shankar, 1999] employs such an approach.

Well-Defined Free Variables. The free variables supported by Rodin are always well-defined, since ordinary variables are well-defined and operator variables are not available. This is helpful for D-definability, because "problematic" formulae like

$\mathsf{D}(\$x)$ do not arise. That said, Theorem 3.23 shows how to recover D-definability in the presence of operator variables.

As an aside, the reader may wonder why Event-B's bound variables may not be ill-defined. Well-definedness of bound variables follows from a restriction on the types that the denotations of binders may inhabit. In a more liberal approach, some binders would range over ill-defined variables as well. As an example, consider a universal quantifier $\forall'$ whose denotation is of type $(\alpha\uparrow \Rightarrow \mathsf{bool}\uparrow) \Rightarrow \mathsf{bool}\uparrow$:

$$\forall'\$x \cdot \$\varphi(\$x) \equiv (\forall x \cdot \$\varphi(x)) \wedge \$\varphi(\bullet).$$

Apart from reduced implementation effort, it does not seem beneficial to exclude such binders from the logic.

Summary. Directed rewriting and unlifting have motivated several limitations of Rodin's implementation of Event-B's logic. Most of these limitations simplify the implementation but are not strictly required for directed rewriting and unlifting. In Section 3.2.4, I explain how to overcome one of these limitations, namely the convention that free variables are well-defined. It seems possible to overcome further limitations, but the technical details still need to be worked out.

4.3. Kleene versus McCarthy Semantics

The semantics of disjunction (and correspondingly conjunction and implication) has been discussed at length. Opponents of Kleene disjunction $\vee$ (as adopted by Event-B's logic) usually favor McCarthy disjunction $\vee_{\mathrm{MC}}$ defined by

$$\$\varphi \vee_{\mathrm{MC}} \$\psi \equiv \mathsf{if}\ \$\varphi\ \mathsf{then}\ \top\ \mathsf{else}\ \$\psi.$$

The McCarthy disjunction is strict in its first and lazy in its second argument. Concerning the quantifiers $\forall$ and $\exists$, Event-B's logic adopts Kleene quantifiers; the McCarthy versions of $\forall$ and $\exists$ are smashed.

As the main argument in favor of McCarthy disjunction, McCarthy [1963] himself stresses its intuitive operational interpretation; indeed, McCarthy disjunction closely resembles the disjunction provided by Ada, C, Java, and several other programming languages.

Another argument in favor of McCarthy disjunction [pointed out by Abrial and Mussat, 2002, Rushby et al., 1998] is that it allows for a more efficient implementation of unlifting. This argument is inspired by the observation that the rule

$$\mathsf{D}(\$\varphi \vee_{\mathrm{MC}} \$\psi) \equiv \mathsf{D}(\$\varphi) \vee (\neg\$\varphi \Rightarrow \mathsf{D}(\$\psi)) \tag{4.8}$$

leads to a polynomial blowup during unlifting, whereas the corresponding rule for the Kleene disjunction

$$\mathsf{D}(\$\varphi \vee \$\psi) \equiv (\mathsf{D}(\$\varphi) \vee \mathsf{D}(\$\psi)) \vee (\mathsf{D}(\$\varphi) \vee \$\varphi) \vee (\mathsf{D}(\$\psi) \vee \$\psi) \tag{4.9}$$

leads to an exponential blowup. In my view, this efficiency argument has become outdated: Darvas et al. [2008] have recently developed an unlifting technique that avoids using the rule (4.9). The complexity of this unlifting technique is still exponential in the worst case; but I will show in Section 5.2.2 that the complexity is linear except in corner cases and substantiate this observation with empirical results. With McCarthy disjunction on the other hand, unlifting leads to a quadratic blow-up even in common cases such as iterated disjunctions.

One argument in favor of Kleene disjunction is that it makes the formula

$$x \bmod 2 = 2 \vee (-x) \bmod 2 = 2 \tag{4.10}$$

valid, whereas the corresponding formula

$$x \bmod 2 = 2 \vee_{\mathrm{MC}} (-x) \bmod 2 = 2$$

is not[3]. Validity of the latter formula cannot be recovered by reordering disjuncts. As supporters of McCarthy disjunction, Shankar and Owre [1999] object that "In practice, we have yet to encounter a need for this kind of expressiveness".

Another argument in favor of Kleene disjunction [given by Cheng and Jones, 1990, Darvas et al., 2008, Jones, 2006] is commutativity; note that the McCarthy disjunction fails to be commutative. Although this argument has strongly influenced the decision for Kleene semantics in the implementation of Event-B's logic, commutativity rules for conjunction or disjunction have never been implemented. To understand the real benefits of commutativity, I have analyzed the rules implemented by Rodin. Interestingly, only two of these rules would become unsound if symbols were interpreted in McCarthy semantics instead of Kleene semantics, namely:

$$\$\varphi \Rightarrow \$\psi \;\sqsubseteq\; \neg\$\psi \Rightarrow \neg\$\varphi, \qquad (\$\varphi_1 \vee \$\varphi_2) \Rightarrow \$\psi \;\sqsubseteq\; (\$\varphi_1 \Rightarrow \$\psi) \wedge (\$\varphi_2 \Rightarrow \$\psi).$$

In turn, some other rules have preconditions that could be dispensed with if the involved symbols were interpreted in McCarthy semantics:

$$\frac{\mathsf{D}(\$y)}{\forall x \cdot x = \$y \Rightarrow \$\varphi(x) \;\sqsubseteq\; \$\varphi(\$y)}, \qquad \frac{\mathsf{D}(\$y)}{\exists x \cdot x = \$y \wedge \$\varphi(x) \;\sqsubseteq\; \$\varphi(\$y)}.$$

I therefore view the positive impact of Kleene semantics on Rodin's proof calculus as rather small.

Rodin implements a sophisticated approximation technique that aims to combine the benefits of Kleene and McCarthy semantics. By default, symbols are interpreted according to Kleene semantics. A well-definedness condition $\mathsf{D}(t)$ is however approximated by $\mathsf{D}(t')$, where t' results from replacing Kleene symbols by the corresponding McCarthy symbols (e.g., $\vee$ by $\vee_{\mathrm{MC}}$). This approximation is sound for two

[3] This example has originally been stated using subtraction restricted to the natural numbers and goes back to Jones [2006].

reasons. First, $\mathsf{D}(t')$ implies $\mathsf{D}(t)$, because Rodin supports only monotonic symbols and McCarthy semantics imposes smaller domains than Kleene semantics. Second, well-definedness conditions in Rodin arise only in goals of sequents, and it is sound to strengthen the goal of the sequent to be proved.

This approximation technique has several consequences:

1. Boolean connectives in Event-B components (contexts or machines) are effectively interpreted in McCarthy semantics and therefore closely correspond to their counterparts in several programming languages.

2. Unlifting has polynomial runtime.

3. Within proofs, conjunction and disjunction are commutative, and the contraposition law is sound; but only the contraposition law has been implemented.

4. Within Event-B components, conjunction and disjunction are not commutative. The contraposition law does not hold either.

5. The approximation prevents users from introducing the commutativity rules for conjunction and disjunction with Rodin's rule definition facilities.

6. The approximation prevents users from proving theorems such as (4.10).

7. The approximation has prevented the implementation of the rules $\mathsf{D_L}$ and $\mathsf{D_R}$ (of Min), which makes the resulting proof calculus incomplete.

8. The approximation is complicated, has consequently confused Rodin developers, and has caused at least one soundness bug [Schmalz, 2012].

The current solution has the advantages of McCarthy semantics (Points 1 and 2), it does not exploit (Point 3) or not have (Points 4 - 6) the advantages of Kleene semantics, and it has additional disadvantages (Point 7 - 8). It is therefore unclear why the current solution should be preferred over interpreting all symbols in McCarthy semantics.

4.4. Alternative Approaches to Partial Functions

There are numerous ways of modeling partial functions; Abrial and Mussat [2002], Cheng and Jones [1990], Farmer [1990], Jones [2006] and Müller and Slind [1997] provide comparisons between various approaches preceding Event-B. The purpose of this section is to compare Event-B with some of these approaches; because of the number of approaches that have been proposed, this comparison cannot be complete. For uniformity, examples are given in the notation of Event-B or HOL.

PVS. PVS (the "prototype verification system") [Owre et al., 1992] models partial functions as total functions over restricted domains. These domains are formalized with a sophisticated type system that supports predicate subtypes and dependent types [Rushby et al., 1998, Shankar and Owre, 1999]. If well-definedness in Event-B is identified with type-correctness in PVS, the operators and binders of PVS are monotonic, types are flat, variables are well-defined, and sequents have SW-semantics. Unlike in Event-B, the boolean connectives are interpreted in McCarthy semantics, and quantifiers are smashed.

Overall, I view my presentation of Event-B's logic as more lightweight than the semantics specification by Owre and Shankar [1999]. One reason for the heaviness of the PVS semantics specification is the decision to interpret PVS terms in a restricted version of Zermelo-Fraenkel set theory instead of a typed logic. Another reason is that PVS has features (such as type judgements) that are not available in Event-B.

PVS provides similar methods as Event-B for defining new operators and binders, which are viewed as constants of appropriate higher-order types. Methods for introducing new inference rules seem not to be available.

I am not aware of a paper explaining the foundations of term rewriting in PVS; the following considerations are therefore based on the prover guide [Shankar et al., 2001] and experimentation. PVS implements a number of directed rewrite rules; an example is $0 * \$x \sqsubseteq 0$ [Shankar et al., 2001, p. 86]. The semantics specification of PVS [Owre and Shankar, 1999] does not provide an explanation why this kind of rewriting is sound; but the results of this document on soundness (Section 2.3.2.3) and safety (Section 4.1) apply to PVS as well.

It is possible to declare conditional equalities $\varphi \Rightarrow t = u$ as rewrite rules. This method for introducing new rewrite rules is however substantially weaker than the corresponding method in Event-B, for two reasons. First, PVS does not support an analog to Event-B's congruence method for binders (cf. Definition 2.16). That prevents users from applying the rule $x \in \varnothing \equiv \bot$ to $\forall x \cdot x \in \varnothing$. Second, PVS uses ordinary variables as place-holders that are instantiated when rewrite rules are matched. These variables can only be instantiated with type-correct terms, and the type-correctness needs to be checked whenever the rule is instantiated. In comparison to Event-B, the method for introducing new rewrite rules is simpler, because constructions such as in Section 3.2.4 can be avoided; the price is an increased effort (in terms of type-correctness checks) when applying user supplied rewrite rules during proofs.

LPF. LPF (the "logic of partial functions") by Barringer et al. [1984], and Jones and Middelburg [1994] is the logic underlying VDM, which has been implemented in, e.g., the Mural system [Jones et al., 1991]. As in Event-B, functions in LPF may be partial, i.e., they may fail to yield a result for certain arguments. The logic correspondingly provides undefined (i.e., ill-defined) terms. Variables in LPF range over defined values, non-monotonic operators are available, and boolean connectives and quantifiers are interpreted in Kleene semantics.

Terms in LPF are a priori untyped, i.e., $\varnothing + 0$ is considered a legal term; LPF still provides (flat) types, including predicate subtypes, and type-correctness of terms may be proved separately. In particular, it cannot be proved that $\varnothing + 0$ inhabits one of LPF's types. Unlike in Event-B, sets in LPF are always finite.

LPF interprets sequents in SS-semantics. It is consequently sound to apply directed rewrite rules to hypotheses, as long as only the arguments of monotonic operators and binders are rewritten; but it is unsound to apply directed rewrite rules to goals. In fact, Dawson [1998] already proposes a version of unconditional directed rewriting for quantifier free LPF; I complement his work by providing evidence that directed rewriting is relevant in practice and by my results on safety.

LPF employs a method for defining (recursive) functions, but it seems not to provide ways for defining new binders or introducing new rules.

Semi-classical Logics. Semi-classical logics support partial functions in a setting with only two truth values. Predicates consequently yield truth or falsity if one of their arguments is undefined and can therefore not be monotonic. Several semi-classical logics have been proposed: LUTINS [Farmer et al., 1993] and its predecessors [Farmer, 1990, 1993], some formulations of Z [Spivey, 1988, 1992], and the logics by [Beeson, 1985, Burge, 1974, Feferman, 1990, Schock, 1968]; the underlying ideas can already be found in papers by Russell [such as 1905]. In the following, I focus on LUTINS, because it is the only logic in this list that constitutes the foundation of a theorem prover[4].

LUTINS is a higher-order logic of partial functions, the logic underlying the IMPS theorem prover [Farmer et al., 1993, 1996]. To understand how undefinedness is propagated in LUTINS, consider a LUTINS function f of type $\nu_1\!\uparrow \Rightarrow \cdots \Rightarrow \nu_n\!\uparrow \Rightarrow \mu\!\uparrow$, where μ is an atomic (i.e., non-functional) type. The semantics of LUTINS adopts the following conventions:

1. If μ is the boolean type, then f denotes a definite function.
2. If f is a variable or a lambda abstraction and μ is the boolean type, then applying f to an undefined argument yields falsity.
3. If f is a variable or a lambda abstraction and μ is not the boolean type, then f is a strict partial function.
4. If f is a variable or a constant that does not take arguments (i.e., $n = 0$), then f is defined.

LUTINS also provides analogs to Event-B's strong equality $\equiv_{\mathsf{eb}}$ and lazy operators like Event-B's conditional. This is the reason why Conditions 2 and 3 restrict f to variables and lambda abstractions.

[4] Of course, there are several theorem provers for Z: ProofPower [Arthan, 1996], CADiZ [CADiZ/Ref, Toyn and McDermid, 1995], HOL-Z [Brucker et al., 2003, Kolyang et al., 1996], Z/EVES [Saaltink 1997 and p. 328 of Woodcock et al. 2009]; none of them implements Z as a semi-classical logic.

As a consequence of these conventions, the following is a theorem:

$$\mathsf{if}\ 1 \div 0 = 0\ \mathsf{then}\ 1 \div 0 \neq 0\ \mathsf{else}\ \neg(1 \div 0 \neq 0).$$

Farmer [1990] claims that this kind of treating undefined terms "correspond[s] ... to how mathematicians usually reason with nondenoting terms". I leave it to the reader to draw his own conclusions.

LUTINS provides non-flat function types such as $(\mathsf{int} \Rightarrow \mathsf{int}\uparrow) \Rightarrow (\mathsf{int} \Rightarrow \mathsf{int}\uparrow)$. An inhabitant of this type is the partial function $\lambda f \cdot \lambda x \cdot f\ (x + 1)$. Event-B's type system does not provide the non-flat functions types of LUTINS, but of course the corresponding functions can be expressed in Event-B as sets of pairs.

IMPS (the "interactive mathematical proof system") provides methods for defining new operators and binders as constants of appropriate higher-order types. Users may supply rewrite rules in the form of conditional equalities. Directed rewriting is unsound (and therefore not available) because predicates fail to be monotonic. Since variables (which are always defined) are used as place-holders in user-supplied rewrite rules, definedness conditions need to be checked whenever a rewrite rule is instantiated; in this regard, the situation is rather similar as in PVS. Farmer et al. [1993] therefore develop a sophisticated machinery for discharging definedness condition; by using operator variables instead of ordinary variables as place-holders in rules, Event-B reduces the need for such a machinery.

LCF. The logic LCF (the "logic of computable functions") [see e.g. Paulson, 1987] also supports partial functions, but deviates from Event-B in several ways. Terms of LCF denote (three-valued) booleans, pairs, and partial functions; other type operators may be introduced as well. The main difference to Event-B and LPF lies in the fact that types are not necessarily flat. LCF therefore supports products of type $\alpha\uparrow \times \beta\uparrow$, and functions of type $\alpha\uparrow \Rightarrow \beta\uparrow$; as type operators can be arbitrarily nested, functions of type $(\alpha_1\uparrow \Rightarrow \alpha_2\uparrow) \Rightarrow (\beta_1\uparrow \Rightarrow (\beta_2\uparrow \times \beta_3\uparrow))$ are available as well. The order relation $\sqsubseteq$ is lifted to non-flat types in a natural way; LCF functions are required to be monotonic and continuous with respect to this order.

Unlike in LUTINS, function variables in LCF may denote lazy functions. Unlike in LCF, functions in LUTINS may be non-monotonic. The set of functions definable in LCF is therefore incomparable with the set of functions definable in LUTINS.

Formulae of LCF are built from strong equality $\equiv$, the order relation $\sqsubseteq$, and the usual first-order connectives and quantifiers. While terms of boolean type are true, false, or undefined, a formula is either true or false but never undefined. In this respect, LCF resembles a semi-classical logic. Note that LCF supports non-monotonic operators – but only at the formula level. Unlike in Event-B, variables range over all values of the underlying type; so a variable may be undefined.

LCF formulae may syntactically include LCF terms, but LCF terms may not include LCF formulae. Moreover, the binders of EventB, most significantly set comprehension, fail to be continuous. For these reasons, an embedding of Event-B's logic into LCF would be quite complicated.

It is unclear whether directed rewriting has applications in LCF, because formulae are built from $\equiv$ and $\sqsubseteq$. The challenge is to reorganize proofs in terms of sequents that have SW-semantics. The monotonicity constraint on functions would certainly be helpful when developing directed rewriting for LCF.

HOLCF [Müller et al., 1999], the modern implementation of LCF, integrates the term language of LCF into higher-order instead of first-order logic. It combines the definitional extension methods of HOL with the definitional extension methods of LCF. It is therefore not surprising that HOLCF's facilities for defining new operators and binders, and for introducing new rules are more general than Event-B's, because of, e.g., fix-point recursion, nested natural deduction rules, and sort restrictions on type variables. There remain several opportunities for transferring theory extension methods from HOLCF to Event-B.

Classical Logics. Although classical logics do not provide partial functions per se, they can still be used to model partial functions. One approach is to avoid functions altogether and to express a function f taking n arguments by a predicate p_f taking $n+1$ arguments; $p_f(x_1, \ldots, x_n, y)$ is used to express $f(x_1, \ldots, x_n) = y$. This approach makes sense from a theoretical point of view, but it is hard to write specifications without using functions. I could not find clear proponents of this approach in the literature; Jones [2006] calls it "clumsy" and "heavy".

Another approach, advocated by Gries and Schneider [1995], Müller and Slind [1997] (but rejected by Jones [1995]), is to model partial functions as total functions that yield unspecified results if applied to arguments outside of their domains. This approach is called *underspecification*; it is widely applied, e.g., in several formulations of Z [Arthan, 1996, CADiZ/Ref, Kolyang et al., 1996, Valentine, 1998], in theorem provers for higher-order logic [Gordon and Melham, 1993, Nipkow et al., 2002], and in several automated theorem provers for first-order logic [e.g., de Moura and Bjørner, 2008, Riazanov and Voronkov, 2002].

In the underspecification approach, $1 \div 0$ denotes an unknown integer. This leads to complications when reasoning about expressions of a programming language: $1 \div 0 = 1 \div 0$ is a theorem even though the corresponding programming language expression does typically not evaluate to true but instead leads to a runtime error. Proponents of the underspecification approach assume that the user of the logic is aware of this discrepancy and able to cope with it. Event-B does not make such an assumption.

Some authors propose to interpret conjectures given by the user in a logic of partial functions, but perform the actual reasoning in a classical logic; this approach has been implemented in Z/EVES [Saaltink, 1997] and in some of Rodin's link-ups to automated theorem provers. Woodcock et al. [2009] develops a general framework for reasoning with a theorem prover for one logic about conjectures of another logic. On the one hand, such an approach combines the advantages of logics of partial functions with the advantages of classical logics: partial functions have an intuitive semantics and powerful automated theorem provers for classical logic can be applied. On the other hand, classical logics are not always more suitable for reasoning about

partial functions, as I have explained in Section 4.1 for term rewriting. Another disadvantage of this approach is that it involves two different semantics, which may complicate the architecture (and therefore compromise correctness) of tools.

5. Automated Theorem Proving

Rodin's theorem provers suffer four kinds of problems: unsoundness, incompleteness, and limitations in performance and adaptability.

Unsoundness. Several versions of Rodin's provers are known to be unsound. Both Rodin's built-in tactics and the so-called external provers (in particular newPP and ML) have been affected [Schmalz, 2012]. Although the known soundness bugs have been fixed, there is no guarantee that the current versions of Rodin's provers are sound. This unsoundness raises a credibility problem; it is a serious obstacle to the use of Rodin for the development of safety critical systems.

Incompleteness. There are surprisingly simple theorems that Rodin's (automated and manual) theorem provers cannot discharge, at least if soundness bugs are not exploited. An example is $3 \mathbin{\hat{}} 3 = 27$.

Performance. Informally, the term *performance* refers to the ability to discharge as many proof obligations as possible in as little time as possible. Rodin users have frequently complained about the performance of Rodin's automated provers, substantiated their complaints with examples (such as $3 \mathbin{\hat{}} 3 = 27$), and expressed that the current performance of Rodin's theorem provers inhibits the use of Rodin.

There are of course applications of Event-B in which Rodin's theorem provers perform very well [Abrial, 2010]; there are however other cases, in which users had to carry out numerous manual proofs, although the proof obligations appear to be rather simple [Ilić et al.].

Adaptability. There are three ways of improving the performance of Rodin's automated theorem provers. First, users may add or remove sub-tactics from Rodin's auto-tactic or change timeouts and parameters of relevance filters [Röder, 2010]. Second, users may use the theory plug-in [Butler and Maamria, 2010, Maamria and Butler, 2010] to introduce inference and rewrite rules, prove their soundness, and apply them automatically during proofs. As the theory plug-in is a recent development, it still suffers restrictions that competing tools like Isabelle's simplifier and classical reasoner do not have. Third, users may change the source code. Overall, it is quite challenging to improve the performance of Rodin's automated provers without compromising soundness.

Overview. In this chapter, I report on my integration of Isabelle/HOL as an automated theorem prover into Rodin. The integration is based on the semantic embedding defined in Section 2.2 of Chapter 2. In a first step, the proof obligation of interest is translated to its denotation. Then (Section 5.2), option types are removed from the conjecture, because they complicate the automated proof search. Finally, a proof method, called `axe`, is invoked to prove the remaining subgoals. In Section 5.3, I report on the process of developing `axe`.

Since Isabelle/HOL follows the well-known LCF approach [Gordon et al., 1979] and is strongly committed to definitional extensions, it is rather unlikely that it produces an unsound proof. Unsoundness may still be introduced by an incorrect implementation of the denotation function, i.e., the translation from Event-B to HOL. I view this as unlikely, because the implementation of the denotation function is concise and straightforward. As opposed to Rodin's theorem provers, the risk of introducing bugs in the future is negligible because there is no need to constantly change the implementation.

According to Gödel's first incompleteness theorem, a consistent, complete, and decidable proof calculus for higher-order logic does not exist [see e.g. Andrews, 2002]. Theorem provers for higher-order logic [such as Gordon and Melham, 1993, Harrison, 2009a, Nipkow et al., 2002] typically provide proof calculi that are complete with respect to *general model semantics* [see e.g. Andrews, 2002]. Validity in the general sense implies validity in the standard sense, but not the other way around. It seems plausible that formulae that are valid in the standard sense and invalid in the general sense do not arise during the formal verification of industrial systems. The integration of Isabelle/HOL into Rodin enables users to prove every sequent whose denotation is valid in the general sense.

A priori, it is clear that by integrating Isabelle/HOL into Rodin, Rodin gets a theorem prover with strong soundness guarantees and a complete proof calculus (in the general sense). The purpose of this chapter is to investigate performance and adaptability.

5.1. Preliminaries

5.1.1. Benchmarks

Most of the experiments of this chapter are based on one of the following two benchmarks. The *Abrial* benchmark consists of Abrial's Event-B models about the examples described in his book [Abrial, 2010]. The models are freely available [Abrial]. The involved proof obligations cover a broad range of application domains. The total number of proof obligations is 2029, and the maximum size[1] of a proof obligation is about 4000.

[1] The size of a proof obligation refers to the size of its denotation. The size of a higher-order logic term is the number of the leaves of its syntax tree.

Varpaaniemi [2010] from Space Systems Finland has developed an Event-B model for a space craft subsystem involved in the BepiColombo mission to Mercury. The *BepiColombo* benchmark consists of Varpaaniemi's model; it contains 625 proof obligations whose size ranges up to 48000.

The purpose of the BepiColombo model is to point out difficulties in the modeling process, and not to cover all requirements on the real system. In this respect, BepiColombo has been quite successful [Ilić et al.]: among other things, the authors of BepiColombo emphasize that "The only really time-consuming activity in the pilot development is proofs – a considerable amount of time is needed for producing a proof, even when it is needed only to reuse an existing proof". This shows that BepiColombo is a challenging industrial case study, not in terms of the total number of proof obligations, but in terms of the effort for developing proofs. Improvements of the prover performance are therefore highly desirable.

5.1.2. Hardware and Software

By default, I have executed experiments on ETH's high-performance computing cluster Brutus. I have run the 64bit version of `Isabelle2011` on machines with a 64bit version of GNU Linux (Kernel version 2.6.18), AMD Opteron 8380 processors with 16 processor cores (2.5 GHz), and 32 GB of main memory. For each experiment, only 1 processor core and 8 GB of main memory has been used. The 64bit version of Isabelle had to be used, because the target machines do not support the 32bit version of Isabelle. The generous amount of main memory is motivated by the fact that the 64bit version of Isabelle consumes more memory than the 32bit version, and that the operating system applies a rather conservative memory allocation strategy[2]. The experiments have been run in batch mode with the usedir options `"-p0 -q1 -M1 -v true"` and the ML option `"-H 2000"`.

Statements on Rodin refer to Rodin 2.2.2 with the following plug-ins:

- Atelier B provers 1.2.2,
- Event-B EMF Framework 3.4.0,
- Export to Isabelle 0.5.0,
- Scala Library for Eclipse 2.9.1,
- Modularisation Plugin 2.2.1,
- Records 1.0.1,
- Relevance Filter 1.1.1,
- Event-B Theory Feature 1.3.0,

[2] To be precise, memory overcommitting is disabled. More information on memory allocation strategies can be found in the man page of `proc`.

- other plug-ins that are unlikely to influence the results of the experiments.

Since it will be difficult to obtain this Rodin installation in the future, I will store an archive of it together with the data used in the experiments. Rodin has been executed with the Java virtual machine 1.6.0_26 by Oracle with the non-standard command line arguments

```
"-Xmx1800m -Xss1m -server -XX:GCTimeRatio=10
-XX:+DoEscapeAnalysis -XX:MaxPermSize=256m".
```

Since it is difficult to disable Rodin's graphical user interface, it is hard to run Rodin on Brutus. I have therefore executed experiments with Rodin on a HP Compaq 6910p laptop with a 32bit version of GNU Linux (Kernel version 2.6.32), a dual core Intel Centrino vPro processor (2.0 GHz) and 4 GB of main memory. To enable comparison of runtimes between Rodin and Isabelle, the experiments with Isabelle in Sections 5.3.1.5 and 5.3.2.2 had to be performed on this laptop as well. For these experiments, I have used the 32bit version of `Isabelle2011` with the usedir options `"-p0 -q1 -M1 -v true"` and the ML option `"-H 4000"`.

The purpose of most experiments was to determine the runtime of proof methods. The term *runtime* always refers to wall-clock time.

5.2. Unlifting

A *lifted* (HOL) type is a type of the form $\nu\uparrow$. Lifted types are heavily used to express the denotations of Event-B terms. This is the main reason why Isabelle's automated proof methods (such as `simp`, `safe`, `auto`, `metis`, `sledgehammer`, and `smt`) perform poorly if they are directly invoked on the denotations of sequents or rules. A way to avoid this problem is to eliminate lifted types before invoking Isabelle's automated methods.

The problem of *unlifting* is to compute for a formula φ a formula φ' without lifted types such that φ is valid iff φ' is valid. Since this research is dedicated to Event-B, I only consider the problem of unlifting denotations of sequents and rules. To simplify the presentation, I focus on denotations of sequents and rules over the signature Σ_{ul} providing the following symbols:

- Type operators: $\mathcal{B}$, $\mathcal{Z}$, $\mathcal{P}$
- Operators: $=$, $\wedge$, D, mod
- Binders: $\forall$, $\mathsf{collect}$, some

It is easy to generalize the results of this section to the signature of EventB; the missing details can be found in the implementation [Schmalz]. The unlifting algorithm described in this section can also be generalized to unlift the denotations of sequents or rules drawn from definitional extensions of EventB; as soon as arbitrary

definitional extensions come into play, it is however hard to give useful complexity results.

Unlifting algorithms have already been presented elsewhere [Abrial and Mussat, 2002, Berezin et al., 2005, Darvas et al., 2008, Owre and Shankar, 1999, Woodcock et al., 2009]. The novelties in this work are a parametric complexity analysis, an empirical evaluation on benchmarks from the industry, a method for unlifting denotations of sequents with operator variables, and an implementation of unlifting based on Isabelle's simplifier [Nipkow, 1989]. The implementation based on Isabelle has the advantage that the unlifting algorithm is sound by construction and can be optimized easily without compromising soundness.

Proceeding. I first present a naive unlifting algorithm in a setting without operator variables (Section 5.2.1). Then (Section 5.2.2), I recall the more efficient algorithm by Darvas et al. [2008]. In Section 5.2.3, I explain my extension for operator variables. The complexity of the unlifting algorithm is analyzed in Section 5.2.4. Finally, I report on my implementation (Section 5.2.5) and experimental evaluation (Section 5.2.6).

5.2.1. Trivial Algorithm

To examine the problem of unlifting, it is useful to introduce the HOL constant $\downarrow$ of type $\alpha\uparrow \Rightarrow \alpha$ that is defined by

$$x\uparrow\downarrow = x;$$

note that $\bullet\downarrow$ is an unknown inhabitant of the type α.

A straightforward unlifting algorithm repeatedly applies the rewrite rules in Figure 5.1. Readers not familiar with Isabelle/HOL should view Isabelle's meta-equality $\equiv$ as an alternative notation for HOL's object equality $=$. The only difference is that $\equiv$ has a lower priority than any other HOL symbol; so $\mathsf{T}\,\varphi \equiv \mathsf{WD}\,\varphi \wedge \varphi\downarrow$ is to be read as $(\mathsf{T}\,\varphi) \equiv (\mathsf{WD}\,\varphi \wedge \varphi\downarrow)$.

Proposition 5.1. *Suppose that φ is the denotation of a sequent or rule over the signature Σ_{ul}. If φ does not contain denotations of operator variables, then φ can be unlifted by repeatedly applying the rules in Figure 5.1.*

Proof. Recall the following observation about term rewriting. When rewriting a formula $\psi_1 \longrightarrow \psi_2$, the premise ψ_1 may be assumed while rewriting the conclusion ψ_2. Likewise, when rewriting $\psi_1 \wedge \psi_2$, the first conjunct ψ_1 may be assumed while rewriting the second conjunct ψ_2. The local assumption ψ_1 may subsequently be used to discharge preconditions of rules that are used to rewrite ψ_2.

This observation can be used to construct a proof that well-definedness of t may be assumed whenever the algorithm tries to rewrite a term of the form $t\downarrow$. Therefore, the preconditions of the rules in Group 3 can always be solved while repeatedly applying the rules in Groups 1–3 to φ. The proof proceeds as follows.

Group 1:

$$\mathsf{T}\,\varphi \;\equiv\; \mathsf{WD}\,\varphi \wedge \varphi\!\downarrow \qquad\qquad \mathsf{F}\,\varphi \;\equiv\; \mathsf{WD}\,\varphi \wedge \neg\varphi\!\downarrow$$

$$\mathsf{WT}\,\varphi \;\equiv\; \mathsf{WD}\,\varphi \longrightarrow \varphi\!\downarrow \qquad\qquad x \sqsubseteq y \;\equiv\; \mathsf{WD}\,x \longrightarrow \mathsf{WD}\,y \wedge x\!\downarrow = y\!\downarrow$$

$$(x :: \nu\!\uparrow) = y \;\equiv\; \mathsf{WD}\,x = \mathsf{WD}\,y \wedge (\mathsf{WD}\,x \wedge \mathsf{WD}\,y \longrightarrow x\!\downarrow = y\!\downarrow)$$

Group 2:

$$\mathsf{WD}\ (x\!\uparrow) \;\equiv\; \mathsf{True} \qquad\qquad \mathsf{WD}\ (\llbracket = \rrbracket\ x\ y) \;\equiv\; \mathsf{WD}\,x \wedge \mathsf{WD}\,y$$

$$\mathsf{WD}\ (\llbracket \wedge \rrbracket\ \varphi\ \psi) \;\equiv\; (\mathsf{T}\,\varphi \wedge \mathsf{T}\,\psi) \vee \mathsf{F}\,\varphi \vee \mathsf{F}\,\psi$$

$$\mathsf{WD}\ (\llbracket \mathsf{D} \rrbracket\ x) \;\equiv\; \mathsf{True} \qquad\qquad \mathsf{WD}\ (\llbracket \mathsf{mod} \rrbracket\ x\ y) \;\equiv\; \mathsf{WD}\,x \wedge \mathsf{WD}\,y \wedge x\!\downarrow \geq 0 \wedge y\!\downarrow > 0$$

$$\mathsf{WD}\ (\llbracket \forall \rrbracket\ \varphi) \;\equiv\; (\forall x.\ \mathsf{T}\ (\varphi\ x)) \vee (\exists x.\ \mathsf{F}\ (\varphi\ x))$$

$$\mathsf{WD}\ (\llbracket \mathsf{collect} \rrbracket\ \varphi) \;\equiv\; \forall x.\ \mathsf{WD}\ (\varphi\ x)$$

$$\mathsf{WD}\ (\llbracket \mathsf{some} \rrbracket\ \varphi) \;\equiv\; (\forall x.\ \mathsf{WD}\ (\varphi\ x)) \wedge (\exists x.\ (\varphi\ x)\!\downarrow)$$

Group 3:

$$(x\!\uparrow)\!\downarrow \;\equiv\; x \qquad\qquad (\llbracket \mathsf{D} \rrbracket\ x)\!\downarrow \;\equiv\; \mathsf{WD}\,x$$

$$\frac{\mathsf{WD}\ (\llbracket = \rrbracket\ x\ y)}{(\llbracket = \rrbracket\ x\ y)\!\downarrow \;\equiv\; x\!\downarrow = y\!\downarrow} \qquad\qquad \frac{\mathsf{WD}\ (\llbracket \wedge \rrbracket\ \varphi\ \psi)}{(\llbracket \wedge \rrbracket\ \varphi\ \psi)\!\downarrow \;\equiv\; \varphi\!\downarrow \wedge \psi\!\downarrow}$$

$$\frac{\mathsf{WD}\ (\llbracket \mathsf{mod} \rrbracket\ x\ y)}{(\llbracket \mathsf{mod} \rrbracket\ x\ y)\!\downarrow \;\equiv\; x\!\downarrow\ \mathsf{mod}\ y\!\downarrow} \qquad\qquad \frac{\mathsf{WD}\ (\llbracket \forall \rrbracket\ \varphi)}{(\llbracket \forall \rrbracket\ \varphi)\!\downarrow \;\equiv\; \forall x.\ (\varphi\ x)\!\downarrow}$$

$$\frac{\mathsf{WD}\ (\llbracket \mathsf{collect} \rrbracket\ \varphi)}{(\llbracket \mathsf{collect} \rrbracket\ \varphi)\!\downarrow \;\equiv\; \{x.\ (\varphi\ x)\!\downarrow\}} \qquad\qquad \frac{\mathsf{WD}\ (\llbracket \mathsf{some} \rrbracket\ \varphi)}{(\llbracket \mathsf{some} \rrbracket\ \varphi)\!\downarrow \;\equiv\; \mathsf{SOME}\ x.\ (\varphi\ x)\!\downarrow}$$

Figure 5.1.: A trivial unlifting algorithm

First, note that the input φ does not contain subterms of the form $t\downarrow$; hence when the algorithm starts, none of the rules in Group 3 can be applied and therefore no preconditions need to be solved.

Some of the rules in Groups 1 and 2 generate subterms of the form $t\downarrow$; the rule $\mathsf{T}\,\varphi \equiv \mathsf{WD}\,\varphi \wedge \varphi\downarrow$ is an example. As this rule generates a formula of the form $\mathsf{WD}\,\varphi \wedge \varphi\downarrow$, $\mathsf{WD}\,\varphi$ may be assumed whenever rewriting the generated subterm $\varphi\downarrow$. In the same way, it can be checked for every rule in Group 1 or 2 that generates a subterm of the form $t\downarrow$ that the condition $\mathsf{WD}\,t$ may be assumed while rewriting the term $t\downarrow$.

Some of the rules in Group 3 generate subterms of the form $t\downarrow$ as well. The formula $(\llbracket=\rrbracket\ x\ y)\downarrow$ may be rewritten to $x\downarrow = y\downarrow$ provided $\llbracket=\rrbracket\ x\ y$ is well-defined. But if $\llbracket=\rrbracket\ x\ y$ is well-defined, then x and y are well-defined as well. Hence, when rewriting $x\downarrow$ or $y\downarrow$, well-definedness of x or y may be assumed, respectively.

A more intricate example concerns the rule that rewrites $(\llbracket\wedge\rrbracket\ \varphi\ \psi)\downarrow$ to $\varphi\downarrow \wedge \psi\downarrow$. This rewriting step is possible only if $\llbracket\wedge\rrbracket\ \varphi\ \psi$ is well-defined. So suppose that $\llbracket\wedge\rrbracket\ \varphi\ \psi$ is well-defined. The condition $\mathsf{WD}\,\varphi$ may be assumed when rewriting $\varphi\downarrow$ for the following reasons: First consider the case that φ is indeed well-defined; then it is sound to assume $\mathsf{WD}\,\varphi$. Second consider the case that φ is ill-defined. In this case, ψ equals $\mathsf{False}\uparrow$ because it has been assumed that $\llbracket\wedge\rrbracket\ \varphi\ \psi$ is well-defined. Hence, the truth value of $\varphi\downarrow \wedge \psi\downarrow$ does not depend on $\varphi\downarrow$, and $\varphi\downarrow$ may consequently be replaced by an arbitrary term. Therefore, it is sound to assume $\mathsf{WD}\,\varphi$ while rewriting $\varphi\downarrow$. A similar argument explains why $\mathsf{WD}\,\psi$ may be assumed when rewriting $\psi\downarrow$ in $\varphi\downarrow \wedge \psi\downarrow$.

For the remaining rules of Group 3 it can be shown in a similar way that $\mathsf{WD}\,t$ may be assumed when rewriting a subterm of the form $t\downarrow$ generated by one of the rules in Group 3.

Overall, I have shown that the preconditions of the rules in Group 3 can always be solved during the process of repeatedly applying rewrite rules from Figure 5.1 to φ. With this in mind, it is easy to check that the rules in Groups 1–3 form a confluent and terminating rewrite calculus, and the normal form of φ does not contain lifted types. It can also be shown (using Isabelle/HOL) that the rules in Figure 5.1 are sound; the normal form of φ is therefore valid iff φ is valid. □

5.2.2. Efficient Algorithm

The complexity of the unlifting algorithm described in Section 5.2.1 is high: unlifting the denotation of $\vdash \mathsf{D}(x_1 \wedge \cdots \wedge x_n)$ yields a formula of exponential size (in n). Darvas et al. [2008] therefore propose a more efficient algorithm, which additionally applies the rules in Group 4 of Figure 5.2. Whenever a rule in Group 4 can be applied to a subterm, one of the rules in Group 1 can be applied to the same subterm as well. This non-confluence is resolved by applying rules of Group 1 only to subterms that are not matched by rules of Group 4.

The efficient algorithm unlifts the denotation of $\vdash \mathsf{D}(x_1 \wedge \cdots \wedge x_n)$ to

$$\mathsf{True} \longrightarrow ((\mathsf{True} \wedge x_1) \wedge \cdots \wedge (\mathsf{True} \wedge x_n)) \vee (\mathsf{True} \wedge \neg x_1) \vee \cdots \vee (\mathsf{True} \wedge \neg x_n),$$

Group 4:

$$\mathsf{T}\ ([\![\wedge]\!]\ \varphi\ \psi) \ \equiv\ \mathsf{T}\,\varphi \wedge \mathsf{T}\,\psi \qquad \mathsf{F}\ ([\![\wedge]\!]\ \varphi\ \psi) \ \equiv\ \mathsf{F}\,\varphi \vee \mathsf{F}\,\psi$$

$$\mathsf{T}\ ([\![\forall]\!]\ \varphi) \ \equiv\ \forall x.\ \mathsf{T}\ (\varphi\ x) \qquad \mathsf{F}\ ([\![\forall]\!]\ \varphi) \ \equiv\ \exists x.\ \mathsf{F}\ (\varphi\ x)$$

Figure 5.2.: Additional rules of the efficient unlifting algorithm

which is of linear size in n.

Proposition 5.2. *Suppose that φ is the denotation of a sequent or rule over the signature Σ_{ul}. If φ does not contain denotations of operator variables, then φ can be unlifted by repeatedly applying the rules in Figures 5.1 and 5.2.*

Proof. The proof is similar to the proof of Proposition 5.1. □

5.2.3. Operator Variables

Since inference and rewrite rules naturally contain operator variables, it is desirable to extend the unlifting algorithm of Section 5.2.2 to a setting with operator variables. For simplicity, it is assumed that there is only one operator variable $\$f \mathbin{:} \nu \rightharpoonup \mu$ that takes exactly one argument; it is straightforward to develop a version for several operator variables that take arbitrary numbers of arguments.

The extended unlifting algorithm proceeds in two phases: a *preprocessing phase* and an *unlifting phase*. During the preprocessing phase, the algorithm eliminates the operator variable $\$f$ from the input. The unlifting phase is essentially an invocation of the algorithm of the preceding section.

Suppose the theory underlying the standard model of Σ_{ul} is called THY. In the preprocessing phase, the algorithm first adds the uninterpreted constant

$$f :: [\![\nu]\!]\!\uparrow \Rightarrow [\![\mu]\!]\!\uparrow$$

to THY, yielding the theory THY_f. Then, the algorithm defines the constants df and sf by

$$df\ (d :: \mathsf{bool})\ (s :: [\![\nu]\!]) = \mathsf{WD}\ (f\ (\mathsf{if}\ d\ \mathsf{then}\ s\!\uparrow\ \mathsf{else}\ \bullet)), \tag{5.1}$$

$$sf\ (d :: \mathsf{bool})\ (s :: [\![\nu]\!]) = (f\ (\mathsf{if}\ d\ \mathsf{then}\ s\!\uparrow\ \mathsf{else}\ \bullet))\!\downarrow. \tag{5.2}$$

The resulting theory is denoted by $\mathsf{THY}_{f,df,sf}$.

Suppose that φ is the denotation of an Event-B sequent or rule over Σ_{ul}. The result of the preprocessing phase is $\varphi' := \varphi[\$f := f]$.

The unlifting phase proceeds similar as in Section 5.2.2. The following rules are used to cope with occurrences of f:

$$\mathsf{WD}\ (f\ x) \equiv df\ (\mathsf{WD}\,x)\ x\!\downarrow, \tag{5.3}$$

$$(f\ x)\!\downarrow \equiv sf\ (\mathsf{WD}\,x)\ x\!\downarrow. \tag{5.4}$$

The rules (5.3–5.4) are sound because they can be derived from (5.1–5.2).

The extension of the unlifting algorithm described so far is problematic, because some unlifted formulae can only be proved by using (5.1–5.2), which contain lifted types. The problem can be seen when trying to prove the result of unlifting

$$\neg \mathsf{WD}\ \ f \longrightarrow \mathsf{WD}\ (\varphi\ f) = \mathsf{WD}\ (\varphi\ \bullet).$$

To overcome this problem, define the theory $\mathsf{THY}_{df,sf}$ by adding the constants df and sf and the following axioms to THY:

$$\neg\, df\ \ d\ s \longrightarrow sf\ \ d\ s = \bullet\!\downarrow, \tag{5.5}$$

$$df\ \ \mathsf{False}\ s = df\ \ \mathsf{False\ undefined}, \tag{5.6}$$

$$sf\ \ \mathsf{False}\ s = sf\ \ \mathsf{False\ undefined}. \tag{5.7}$$

The constant $\mathsf{undefined} :: \alpha$ is declared in the main theory of HOL; its denotation is unspecified.

The following theorem shows that the input of the unlifting algorithm is valid over THY iff the output is valid over $\mathsf{THY}_{df,sf}$. As the theory $\mathsf{THY}_{df,sf}$ does not contain (5.1–5.2), a theorem prover that does not support lifted types may be used to reason about the output.

Theorem 5.3. *Let φ be the denotation of a sequent or rule over Σ_{ul} that does not contain an operator variable different from $\$f$. Moreover let φ'' be the result of unlifting φ. Then, φ is valid over THY iff φ'' is valid over $\mathsf{THY}_{df,sf}$.*

Proof. Since f is an uninterpreted constant, φ is valid over THY iff φ' is valid over THY_f. As $\mathsf{THY}_{f,df,sf}$ is a definitional extension of THY_f, φ' is valid over THY_f iff φ' is valid over $\mathsf{THY}_{f,df,sf}$. As the unlifting phase rewrites formulae to equivalent formulae, φ' is valid over $\mathsf{THY}_{f,df,sf}$ iff φ'' is valid over $\mathsf{THY}_{f,df,sf}$. It remains to prove that φ'' is valid over $\mathsf{THY}_{f,df,sf}$ iff φ'' is valid over $\mathsf{THY}_{df,sf}$.

First suppose that φ'' is valid over $\mathsf{THY}_{df,sf}$. It is easy to check that (5.5–5.7) are valid in $\mathsf{THY}_{f,df,sf}$. Thus, φ'' is valid in $\mathsf{THY}_{f,df,sf}$.

Second suppose that φ'' is valid over $\mathsf{THY}_{f,df,sf}$. Define the theory $\mathsf{THY}'_{f,df,sf}$ by extending $\mathsf{THY}_{df,sf}$ with the following constant definition:

$$f\ x = \mathsf{if}\ df\ (\mathsf{WD}\ \ x)\ x\!\downarrow\ \mathsf{then}\ (sf\ (\mathsf{WD}\ \ x)\ x\!\downarrow)\!\uparrow\ \mathsf{else}\ \bullet.$$

It can be checked that (5.1–5.2) are valid in $\mathsf{THY}'_{f,df,sf}$. Hence, φ'' is valid in $\mathsf{THY}'_{f,df,sf}$. As φ'' does not contain f and $\mathsf{THY}'_{f,df,sf}$ is a definitional extension of $\mathsf{THY}_{df,sf}$, φ'' is valid in $\mathsf{THY}_{df,sf}$ as well. □

5.2.4. Complexity Analysis

Darvas et al. [2008] observe that the output of their unlifting algorithm is of at most quadratic size with respect to the size of the input. This result is however restricted to a first-order setting without equivalence. In a first-order logic with equivalence,

the output of the efficient unlifting algorithm may be of exponential size. This exponential blow-up can be observed when unlifting $\mathsf{T}\ [\![\varphi_n]\!]$, where φ_n is defined as follows:

$$\begin{aligned} \varphi_0 &:= x, \\ \varphi_{i+1} &:= (x = (x \wedge \varphi_i)), \text{ for } i \geq 0. \end{aligned}$$

The reason for the blow-up is that the rule

$$\mathsf{WD}\ ([\![\wedge]\!]\ \varphi\ \psi)\ \equiv\ (\mathsf{T}\ \varphi \wedge \mathsf{T}\ \psi) \vee \mathsf{F}\ \varphi \vee \mathsf{F}\ \psi$$

is applied repeatedly. Since equivalences frequently arise in practice, the complexity analysis and the empirical evaluation by Darvas et al. [2008] do not show that their unlifting algorithm (or a trivial extension thereof) has a reasonable performance in practical applications.

It is quite straightforward to find an upper bound on the size of the unlifting output based on the *size* and the *depth* of the input formula. The result of unlifting the formula φ with the algorithm developed in Sections 5.2.1–5.2.3 is denoted by $\mathcal{U}(\varphi)$. The *size* of a HOL term is the number of the involved variables, constants, and abstractions:

Definition 5.4 (Size)**.** The *size* $|t|$ of a HOL term t is defined as follows:

- $|c| := 1$, where c is a constant or variable,
- $|t\,u| := |t| + |u|$,
- $|\lambda x.\ t| := 1 + |t|$.

□

The *depth* of an Event-B term is the length of the longest path from the root to a leaf of the syntax tree:

Definition 5.5 (Depth)**.** The *depth* $\mathcal{DEP}(t)$ of an Event-B term t is defined as follows:

- $\mathcal{DEP}(x) := 0$,
- $\mathcal{DEP}(f(\underline{\boldsymbol{t}})) := 1 + \mathsf{max}(\mathcal{DEP}_R(t_1), \ldots, \mathcal{DEP}_R(t_{|\underline{\boldsymbol{t}}|}))$,
- $\mathcal{DEP}(\$f(\underline{\boldsymbol{t}})) := 1 + \mathsf{max}(\mathcal{DEP}_R(t_1), \ldots, \mathcal{DEP}_R(t_{|\underline{\boldsymbol{t}}|}))$,
- $\mathcal{DEP}(Q\underline{\boldsymbol{x}} \cdot \underline{\boldsymbol{t}}) := 1 + \mathsf{max}(\mathcal{DEP}_R(t_1), \ldots, \mathcal{DEP}_R(t_{|\underline{\boldsymbol{t}}|}))$.

The *depth of a sequent or rule* is the maximum of $\mathcal{DEP}(t)$, where t ranges over the terms from which the sequent or rule is built. □

The following proposition gives an upper bound on the size of the unlifting output. Although the unlifting algorithm has been presented in the context of Σ_{ul}, the theorem applies to denotations of terms drawn from the signature of EventB. The missing rules of the unlifting algorithm are easy to discover and can be found in the implementation [Schmalz].

Proposition 5.6. *Let* r *be a sequent or rule over* EventB*. Then,*

$$|\mathcal{U}([\![\mathsf{r}]\!])| \in \mathcal{O}(|[\![\mathsf{r}]\!]| \cdot 4^{\mathcal{DEP}(\mathsf{r})}).$$

Proof. The following claims can be proved by a structural induction, provided c and d are sufficiently large:

- $|\mathcal{U}(\mathsf{T}\ [\![\varphi]\!])| \leq c \cdot |[\![\varphi]\!]| \cdot 4^{\mathcal{DEP}(\varphi)} - d$,
- $|\mathcal{U}(\mathsf{F}\ [\![\varphi]\!])| \leq c \cdot |[\![\varphi]\!]| \cdot 4^{\mathcal{DEP}(\varphi)} - d$,
- $|\mathcal{U}(\mathsf{WD}\ [\![t]\!])| \leq c \cdot |[\![t]\!]| \cdot 4^{\mathcal{DEP}(t)} - d$,
- if $\mathsf{WD}\ [\![t]\!]$ is known to be true, then $|\mathcal{U}([\![t]\!]{\downarrow})| \leq c \cdot |[\![t]\!]| \cdot 4^{\mathcal{DEP}(t)} - d$.

Here, t and φ range over terms and formulae of EventB. The above claims hold for $d := 0$, but an induction proof requires that d is significantly greater than 0. It is straightforward to derive the assertion of the Proposition from the above claims.

The base of the power has been chosen to be 4 because the right-hand sides of unlifting rules contain at most 4 free occurrences of free variables of the respective left-hand sides. An example of a rule that replicates a subterm four times is

$$\begin{array}{l} \quad \mathsf{T}\ ([\![\mathsf{funimg}]\!]\ x\ y) \\ \equiv \\ \quad \mathsf{WD}\ \ x \wedge \mathsf{WD}\ \ y \wedge ([\![\mathsf{functional}]\!]\ x){\downarrow} \wedge ([\![\in]\!]\ y\ ([\![\mathsf{dom}]\!]\ x)){\downarrow} \wedge ([\![\mathsf{funimg}]\!]\ x\ y){\downarrow}. \end{array}$$

□

The upper bound of Proposition 5.6 is presumably not tight because it seems impossible to construct an example in which *every* unlifting step replicates a subterm four times. The overhead of unlifting is quite high when unlifting $\mathsf{T}\ [\![\varphi_n]\!]$, where φ_n is defined by

$$\begin{aligned} \varphi_0 &:= x, \\ \varphi_{n+1} &:= x \equiv_{\mathsf{eb}} (\forall x \cdot \varphi_n), \text{ for } n \geq 0. \end{aligned}$$

The size of $\mathcal{U}(\mathsf{T}\ [\![\varphi_n]\!])$ is in $\Theta(5^n) = \Theta(\sqrt{5}^{\mathcal{DEP}(\varphi_n)}) \approx \Theta(2.2^{\mathcal{DEP}(\varphi_n)})$. I was unable to find a sequence of formulae for which the asymptotic overhead of unlifting is higher.

Since inference and rewrite rules are typically of moderate size and depth, the upper bound of Proposition 5.6 shows that the result of unlifting denotations of rules is likely to be of an acceptable size. The situation is different for sequents: in

the BepiColombo benchmark (cf. Section 5.2.6), it is easy to find sequents of depth 10 and more. According to Proposition 5.6, unlifting the denotations of these sequents may increase the size of the input by a factor of 1000000 or more. The upper bound of Proposition 5.6 does therefore not assert that unlifting denotations of sequents is practically feasible.

To give a more useful upper bound, I consider additional parameters of the input, namely the *R-alternation depth* and the *P-depth*, which are defined below. Non-monotonic operators and operator variables tend to make unlifting expensive, but do not arise in Rodin's proof obligations. I therefore focus on sequents built from formulae in $\mathsf{mono}(\mathsf{EventB})$.

The *R-alternation depth* of a term counts the maximum number of alternations between symbols that do not belong to R and symbols that do.

Definition 5.7 (R-Alternation Depth)**.** Let R be a set of Event-B operators and binders. The R-alternation depth $\mathcal{ALT}_R(t)$ of an Event-B term t is defined as follows:

- $\mathcal{ALT}_R(x) := 0$,
- $\mathcal{ALT}_R(\$f(\underline{\boldsymbol{t}})) := \mathsf{max}(\mathcal{ALT}_R(t_1), \ldots, \mathcal{ALT}_R(t_{|\underline{\boldsymbol{t}}|}))$,
- if $f \in R$, then $\mathcal{ALT}_R(f(\underline{\boldsymbol{t}})) := \mathsf{max}(\mathcal{ALT}_R(t_1), \ldots, \mathcal{ALT}_R(t_{|\underline{\boldsymbol{t}}|}))$,
- if $f \notin R$, then $\mathcal{ALT}_R(f(\underline{\boldsymbol{t}})) := \mathsf{max}(\mathcal{ALT}'_R(t_1), \ldots, \mathcal{ALT}'_R(t_{|\underline{\boldsymbol{t}}|}))$,
- if $Q \in R$, then $\mathcal{ALT}_R(Q\underline{\boldsymbol{x}} \cdot \underline{\boldsymbol{t}}) := \mathsf{max}(\mathcal{ALT}_R(t_1), \ldots, \mathcal{ALT}_R(t_{|\underline{\boldsymbol{t}}|}))$,
- if $Q \notin R$, then $\mathcal{ALT}_R(Q\underline{\boldsymbol{x}} \cdot \underline{\boldsymbol{t}}) := \mathsf{max}(\mathcal{ALT}'_R(t_1), \ldots, \mathcal{ALT}'_R(t_{|\underline{\boldsymbol{t}}|}))$,

- $\mathcal{ALT}'_R(x) := 0$,
- $\mathcal{ALT}'_R(\$f(\underline{\boldsymbol{t}})) := \mathsf{max}(\mathcal{ALT}'_R(t_1), \ldots, \mathcal{ALT}'_R(t_{|\underline{\boldsymbol{t}}|}))$,
- if $f \in R$, then $\mathcal{ALT}'_R(f(\underline{\boldsymbol{t}})) := 1 + \mathsf{max}(\mathcal{ALT}_R(t_1), \ldots, \mathcal{ALT}_R(t_{|\underline{\boldsymbol{t}}|}))$,
- if $f \notin R$, then $\mathcal{ALT}'_R(f(\underline{\boldsymbol{t}})) := \mathsf{max}(\mathcal{ALT}'_R(t_1), \ldots, \mathcal{ALT}'_R(t_{|\underline{\boldsymbol{t}}|}))$,
- if $Q \in R$, then $\mathcal{ALT}'_R(Q\underline{\boldsymbol{x}} \cdot \underline{\boldsymbol{t}}) := 1 + \mathsf{max}(\mathcal{ALT}_R(t_1), \ldots, \mathcal{ALT}_R(t_{|\underline{\boldsymbol{t}}|}))$,
- if $Q \notin R$, then $\mathcal{ALT}'_R(Q\underline{\boldsymbol{x}} \cdot \underline{\boldsymbol{t}}) := \mathsf{max}(\mathcal{ALT}'_R(t_1), \ldots, \mathcal{ALT}'_R(t_{|\underline{\boldsymbol{t}}|}))$.

The *R-alternation depth of a sequent or rule* is the maximum of $\mathcal{ALT}_R(t)$, where t ranges over the terms from which the sequent or rule is built. □

The *R-depth* of a term counts the maximum number of nested occurrences of symbols in R.

Definition 5.8 (R-Depth)**.** Let R be a set of Event-B operators, operator variables, and binders. The *R-depth* $\mathcal{DEP}_R(t)$ of a term t is defined as follows:

- $\mathcal{DEP}_R(x) := 0$,
- if $f \in R$, then $\mathcal{DEP}_R(f(\underline{\boldsymbol{t}})) := 1 + \mathsf{max}(\mathcal{DEP}_R(t_1), \ldots, \mathcal{DEP}_R(t_{|\underline{\boldsymbol{t}}|}))$,
- if $f \notin R$, then $\mathcal{DEP}_R(f(\underline{\boldsymbol{t}})) := \mathsf{max}(\mathcal{DEP}_R(t_1), \ldots, \mathcal{DEP}_R(t_{|\underline{\boldsymbol{t}}|}))$,
- if $\$f \in R$, then $\mathcal{DEP}_R(\$f(\underline{\boldsymbol{t}})) := 1 + \mathsf{max}(\mathcal{DEP}_R(t_1), \ldots, \mathcal{DEP}_R(t_{|\underline{\boldsymbol{t}}|}))$,
- if $\$f \notin R$, then $\mathcal{DEP}_R(\$f(\underline{\boldsymbol{t}})) := \mathsf{max}(\mathcal{DEP}_R(t_1), \ldots, \mathcal{DEP}_R(t_{|\underline{\boldsymbol{t}}|}))$,
- if $Q \in R$, then $\mathcal{DEP}_R(Q\underline{\boldsymbol{x}} \cdot \underline{\boldsymbol{t}}) := 1 + \mathsf{max}(\mathcal{DEP}_R(t_1), \ldots, \mathcal{DEP}_R(t_{|\underline{\boldsymbol{t}}|}))$,
- if $Q \notin R$, then $\mathcal{DEP}_R(Q\underline{\boldsymbol{x}} \cdot \underline{\boldsymbol{t}}) := \mathsf{max}(\mathcal{DEP}_R(t_1), \ldots, \mathcal{DEP}_R(t_{|\underline{\boldsymbol{t}}|}))$.

The *R-depth of a sequent or rule* is the maximum of $\mathcal{DEP}_R(t)$, where t ranges over the terms from which the sequent or rule is built. □

The following theorem is the main result of this section.

Theorem 5.9. *Suppose Γ is a sequent built from formulae in* mono(EventB). *Define P as the set that contains every indefinite operator or binder of* EventB *as well as set comprehension, $\bigcup$, and* cond. *Let n be the size of $[\![\Gamma]\!]$, a the $\{\forall, \exists, \wedge, \vee, \Rightarrow\}$-alternation depth of Γ, and d its P-depth. Then the result of unlifting $[\![\Gamma]\!]$ is of size $\mathcal{O}(2^a \cdot n \cdot (d+1))$.*

Proof. Let $R := \{\forall, \exists, \wedge, \vee, \Rightarrow\}$. The proof rests on the following observations:

- $|\mathcal{U}(\mathsf{T}\ [\![\varphi]\!])| \leq 180 \cdot 2^{\mathcal{ALT}_R(\varphi)} \cdot |\varphi| \cdot (\mathcal{DEP}_P(\varphi) + 1) - 60 \cdot |\varphi|$,
- $|\mathcal{U}(\mathsf{F}\ [\![\varphi]\!])| \leq 180 \cdot 2^{\mathcal{ALT}_R(\varphi)} \cdot |\varphi| \cdot (\mathcal{DEP}_P(\varphi) + 1) - 60 \cdot |\varphi|$,
- if $\mathsf{WD}\ [\![t]\!]$ is known to be true, then $|\mathcal{U}([\![t]\!]\downarrow)| \leq 20 \cdot |t|$,
- $|\mathcal{U}(\mathsf{WD}\ [\![t]\!])| \leq 360 \cdot 2^{\mathcal{ALT}_R(t)} \cdot |t| \cdot (\mathcal{DEP}_P(t) + 1) - 120 \cdot |t|$.

Here, t and φ range over terms and formulae of mono(EventB). I have proved slightly stronger versions of these observation by a structural induction using Isabelle/HOL. The details can be found in Appendix B.

Suppose Γ has the form $\psi_1, \ldots, \psi_k \vdash \varphi$. The main assertion can be proved as follows:

$$\begin{aligned}
|\mathcal{U}\,[\![\Gamma]\!]| &= \sum_{i=1}^{k} |\mathcal{U}(\mathsf{T}\ [\![\psi_i]\!])| + |\mathcal{U}(\mathsf{WT}\ [\![\varphi]\!])| + \mathcal{O}(k) \\
&\leq \sum_{i=1}^{k} \mathcal{O}(2^a \cdot |[\![\psi_i]\!]| \cdot (d+1)) + \mathcal{O}(2^a \cdot |[\![\varphi]\!]| \cdot (d+1)) \\
&\leq \mathcal{O}(2^a \cdot n \cdot (d+1)).
\end{aligned}$$

□

Informally, Theorem 5.9 asserts that nesting of indefinite operators and binders, conditionals, and set comprehensions (measured with the parameter d) may lead to a quadratic blow-up during unlifting. Moreover, alternations between $\forall, \exists, \wedge, \vee, \Rightarrow$ and the other symbols of EventB (measured with the parameter a) may lead to an exponential blow-up. It seems plausible that the parameters a and d are quite low for many sequents arising in practical applications of Event-B; Theorem 5.9 asserts that the denotations of such sequents can be unlifted with constant overhead.

A minor drawback of Theorem 5.9 is that it gives the incorrect impression that alternations between negations and, say, conjunctions lead to an exponential blow-up. In fact, negations can be unlifted with the following rules:

$$\mathsf{T}\ (\llbracket\neg\rrbracket\ \varphi) \equiv \mathsf{F}\,\varphi \quad \mathsf{F}\ (\llbracket\neg\rrbracket\ \varphi) \equiv \mathsf{T}\,\varphi \quad \mathsf{WD}\ (\llbracket\neg\rrbracket\ \varphi) \equiv \mathsf{WD}\,\varphi \quad \frac{\mathsf{WD}\ (\llbracket\neg\rrbracket\ \varphi)}{(\llbracket\neg\rrbracket\ \varphi)\downarrow\ \equiv\ \neg(\varphi\downarrow)}$$

So negations do not contribute to a blow-up during unlifting. Formally, it can be proved that $|\mathcal{U}(t)| \leq |\mathcal{U}(t')| + i$, where i is the number of negations in t and t' the result of removing all negations from t. Using the terminology in Theorem 5.9,

$$|\mathcal{U}(t)| \leq \mathcal{O}(2^{a'} \cdot |t| \cdot (d+1)),$$

where a' is the $\{\forall, \exists, \wedge, \vee, \Rightarrow\}$-alternation depth of t'.

5.2.5. Implementation

I have implemented the unlifting algorithm described in the preceding sections based on Isabelle's simplifier [Nipkow, 1989]. The source code can be obtained from [Schmalz]. The unlifting process is invoked with `ebsimp'`[3] and performs the following steps:

1. preprocessing (`ebpre'`),
   a) bottom-up unlifting (`ebpresimp'`),
   b) speculative splitting (`ebpresplit`),
   c) if the output of `ebpresplit` differs from its input:
      a second invocation of `ebpresimp'`,
2. unlifting (`ebtrans`),
   a) eliminating operator variables (`ebtranssplit`),
   b) unfolding auxiliary definitions (`ebtranssimp1`),
   c) unlifting (`ebtranssimp2`),
3. postprocessing (`ebpost`).

Preprocessing and postprocessing apply heuristics to improve readability of the result. The unlifting algorithm is executed in the unlifting phase.

[3] Names of proof methods are somewhat ad-hoc and subject to change in future versions. The prime indicates that an older version of the same method, called `ebsimp`, exists or has existed.

Preprocessing. The preprocessing phase consists of *bottom-up unlifting* and *speculative splitting*. Informally, bottom-up unlifting rewrites terms that are obviously well-defined to the form $t\!\uparrow$ and terms that are obviously ill-defined to $\bullet$. Using the example of conjunction, this is achieved by the following rewrite rules:

$$[\![\wedge]\!]\ \varphi\!\uparrow\ \psi\!\uparrow\ \equiv\ (\varphi \wedge \psi)\!\uparrow, \qquad\qquad [\![\wedge]\!]\ \bullet\ \bullet\ \equiv\ \bullet.$$

Bottom-up unlifting has the effect that $\mathsf{WD}\ [\![x \wedge y]\!]$ is unlifted to True instead of $(\mathsf{True} \wedge x) \wedge (\mathsf{True} \wedge y) \vee (\mathsf{True} \wedge \neg x) \vee (\mathsf{True} \wedge \neg y)$.

Bottom-up unlifting also applies propositional simplification rules such as

$$\begin{aligned}
[\![\wedge]\!]\ \varphi\ \mathsf{False}\!\uparrow\ &\equiv\ \mathsf{False}\!\uparrow,\\
[\![\wedge]\!]\ \mathsf{False}\!\uparrow\ \psi\ &\equiv\ \mathsf{False}\!\uparrow,\\
[\![\wedge]\!]\ \varphi\ \mathsf{True}\!\uparrow\ &\equiv\ \varphi,\\
[\![\wedge]\!]\ \mathsf{True}\!\uparrow\ \psi\ &\equiv\ \psi.
\end{aligned}$$

This has the effect that $\mathsf{WD}\ [\![\$x \wedge \top]\!]$ is unlifted to dx instead of

$$(dx \wedge sx) \wedge (\mathsf{True} \wedge \mathsf{True}) \vee (dx \wedge \neg sx) \vee (\mathsf{True} \wedge \neg \mathsf{True}).$$

The method `ebpresplit` for *speculative splitting* tries to replace variables $\$x$ of type $\nu\!\uparrow$ (i.e., denotations of operator variables that do not take arguments) by terms of the form $x\!\uparrow$ or $\bullet$; it therefore reduces the need for unlifting operator variables and the corresponding overhead. Given a variable $\$x :: \nu\!\uparrow$, `ebpresplit` first performs a case split with the cases (1) $\$x = x\!\uparrow$, where x is a fresh variable, and (2) $\$x = \bullet$. The assumption $\$x = x\!\uparrow$ or $\$x = \bullet$ is used to eliminate occurrences of $\$x$ from the input formula. Next, `ebpresplit` tries to solve at least one of the two cases with `ebpresimp'`, `ebtrans`, and `ebpost`; if none of the two cases can be solved, the case split is rescinded.

The second invocation of `ebpresimp'` is likely to succeed because successful invocation of `ebpresplit` introduces subterms of the form $x\!\uparrow$ or $\bullet$. An alternative version of `ebpre'` invokes `ebpresplit` and `ebpresimp'` in turns until the proof state does not change anymore; I have refrained from invoking `ebpresplit` more than once because a second invocation is quite expensive and unlikely to succeed.

Unlifting. The method `ebtranssplit` eliminates operator variables similarly as described in Section 5.2.3. There are two major differences. First, `ebtranssplit` leaves the underlying theory unchanged; in the notation of Section 5.2.3, the symbols *df* and *sf* are variables that are constrained by suitable hypotheses instead of constants that are introduced by constant definitions. The implemented version is somewhat more flexible, but requires a deeper understanding of Isabelle.

Second, if $\$f$ is of type $\nu \rightharpoonup \mu$, then *df* is of type $[\![\nu]\!]\!\uparrow \Rightarrow \mathsf{bool}$ and *sf* of type $[\![\nu]\!]\!\uparrow \Rightarrow [\![\mu]\!]$. (In Section 5.2.3, *df* was of type $\mathsf{bool} \Rightarrow [\![\nu]\!] \Rightarrow \mathsf{bool}$, and *sf* was of

type $\mathsf{bool} \Rightarrow [\![\nu]\!] \Rightarrow [\![\mu]\!]$.) Moreover, *df* and *sf* are constrained by the following meta-equalities:

$$\begin{array}{lcl} \mathsf{WD}\ (\$f\ x) & \equiv & df\ (\mathsf{if}\ \mathsf{WD}\, x\ \mathsf{then}\ x\ \mathsf{else}\ \bullet), \\ (\$fx)\downarrow & \equiv & sf\ (\mathsf{if}\ \mathsf{WD}\, x\ \mathsf{then}\ x\ \mathsf{else}\ \bullet). \end{array}$$

The advantage of the Isabelle implementation over the algorithm in Section 5.2.3 is a more compact and natural representation: additional hypotheses or axioms (such as (5.5–5.7)) can be avoided. The algorithm in Section 5.2.3 is slightly more general, because it removes all occurrences of lifted types from the input formula; the Isabelle implementation outputs a formula that may still contain lifted types (such as in the type of $\bullet$) and therefore requires that lifted types are available in the underlying logic.

The method `ebtranssimp1` establishes certain preconditions of `ebtranssimp2`, but has no logical significance and may disappear in future releases. The method `ebtranssimp2` executes the actual unlifting algorithm (Section 5.2.2).

To improve readability of the output, `ebtranssimp2` makes use of rewrite rules that are not contained in Figures 5.1 and 5.2: an example is the rule $\mathsf{T}\, x\!\uparrow \equiv x$. Moreover, I have modified some of the rules to produce a more readable output: An example is the rule $\mathsf{F}\ [\![\wedge]\!]\ \varphi\ \psi \equiv \neg(\neg(\mathsf{F}\ [\![\wedge]\!]) \wedge \neg(\mathsf{F}\ [\![\wedge]\!]))$. The double negation has the effect that the denotation of Event-B's conjunction is translated to a HOL conjunction, and not to a disjunction as in Figure 5.2.

To avoid applications of rules like

$$\mathsf{WD}\ ([\![\wedge]\!]\ \varphi\ \psi) \ \equiv\ (\mathsf{T}\,\varphi \wedge \mathsf{T}\,\psi) \vee \mathsf{F}\,\varphi \vee \mathsf{F}\,\psi,$$

`ebtranssimp2` instead applies the following rules whenever possible:

$$\begin{array}{lcl} \mathsf{WD}\ ([\![\wedge]\!]\ \bullet\ \psi) & \equiv & \mathsf{F}\,\psi, \\ \mathsf{WD}\ ([\![\wedge]\!]\ \varphi\ \bullet) & \equiv & \mathsf{F}\,\varphi, \\ \mathsf{WD}\ ([\![\wedge]\!]\ \varphi\!\uparrow\ \psi) & \equiv & \varphi \longrightarrow \mathsf{WD}\,\psi, \\ \mathsf{WD}\ ([\![\wedge]\!]\ \varphi\ \psi\!\uparrow) & \equiv & \psi \longrightarrow \mathsf{WD}\,\varphi. \end{array}$$

The implementation of `ebtranssimp2` is fairly sophisticated because it has to tackle the following challenges:

- Solve the well-definedness conditions of the rules in Group 3 of Figure 5.1. This is established by suitable congruence rules [Nipkow et al., 2002, p. 175].
- If subterms are frequently duplicated in intermediate steps, the runtime of `ebsimp'` is exponential even though the result is of polynomial size. This problem is avoided by using let bindings at intermediate stages. Note that the output of `ebsimp'` never contains let bindings.

- Isabelle's simplifier by default applies the most general matching rewrite rule; for unlifting it is often necessary to apply a more specific rewrite rule. This is ensured by using *simplification procedures* (i.e., an ML function that generates appropriate rewrite rules on the fly [Wenzel, 2011, p. 138]) and auxiliary constants.

- Set comprehensions bind an arbitrary number of variables. Correspondingly, an infinite number of unlifting rules is required to unlift set comprehensions. These rules are represented by suitable simplification procedures.

Despite these complications, it pays off to implement unlifting as an Isabelle tactic: thanks to Isabelle's LCF architecture, the unlifting tactic is sound, and it is easy to implement further optimizations without compromising soundness.

Postprocessing. The `ebpost` method applies basic propositional and first-order rewrite rules to the result of `ebtrans`. Its purpose is not only to improve readability of the output but also to support `ebpresplit` in solving trivial cases.

5.2.6. Empirical Evaluation

I have evaluated my implementation of unlifting on three benchmarks: the Abrial and BepiColombo benchmarks, described in Section 5.1.1, and the *Rules* benchmark. The *Rules* benchmark consists of denotations of rules and has been included because none of the other benchmarks contains operator variables.

When a new inference or rewrite rule is implemented in Rodin's built-in theorem prover, the rule is announced on one of Rodin's mailing lists to give other developers the opportunity to prevent implementation of unsound rules. The Rules benchmark contains the denotations of 58 rules, taken from the latest announcements.

The output sizes and runtimes for the three benchmarks are depicted in Figures 5.3–5.5. In the Abrial and BepiColombo benchmark, unlifting typically decreases the size of the formula by a factor of 2.5; this decrease of size is presumably caused by a less verbose representation of the output: an ordinary variable x is for example represented by $x\uparrow$ (size 2) in the input and by x (size 1) in the output. Some proof obligations are proved during unlifting, and their output size is consequently zero. In the Rules benchmark, unlifting increases the size of the processed formula by a factor of up to 5.6, but the maximum output size remains in reasonable bounds. A manual inspection of the rules with the highest blow-up has revealed that the blow-up is caused by the treatment of operator variables.

One may speculate that the output sizes result from aggressive postprocessing by `ebpost` instead of an efficient implementation of the main unlifting phase `ebtrans`; this can be refuted by observing the runtime of unlifting: if the implementation of `ebtrans` was inefficient, the overall runtime would be high. But the runtime of unlifting is below 0.9 seconds in the Abrial benchmark, 2.5 seconds per 10000 size units in the BepiColombo benchmark, and up to 0.3 seconds in the Rules benchmark.

The runtimes in the BepiColombo benchmark are more concentrated on a straight line than the runtimes in the Abrial benchmark. I expect this is a consequence of the fact that the BepiColombo benchmark originates from a single application domain, while the Abrial benchmark is more heterogeneous.

There are several outliers in the BepiColombo benchmark where the runtime is significantly higher than 2.5 seconds per 10000 size units. It turns out on manual inspection that these increased runtimes are caused by the garbage collector, i.e., they are indirectly caused by previous computations.

Overall, these measurements show that my implementation of unlifting has a reasonable performance on an important class of formulae. Although small examples can be constructed for which unlifting takes several hours, days, or years, such examples do not occur in the benchmarks. The runtime of unlifting is not negligible (i.e., sometimes in the order of several seconds), but it is proportional to the size of the input for the examples considered during the experiments.

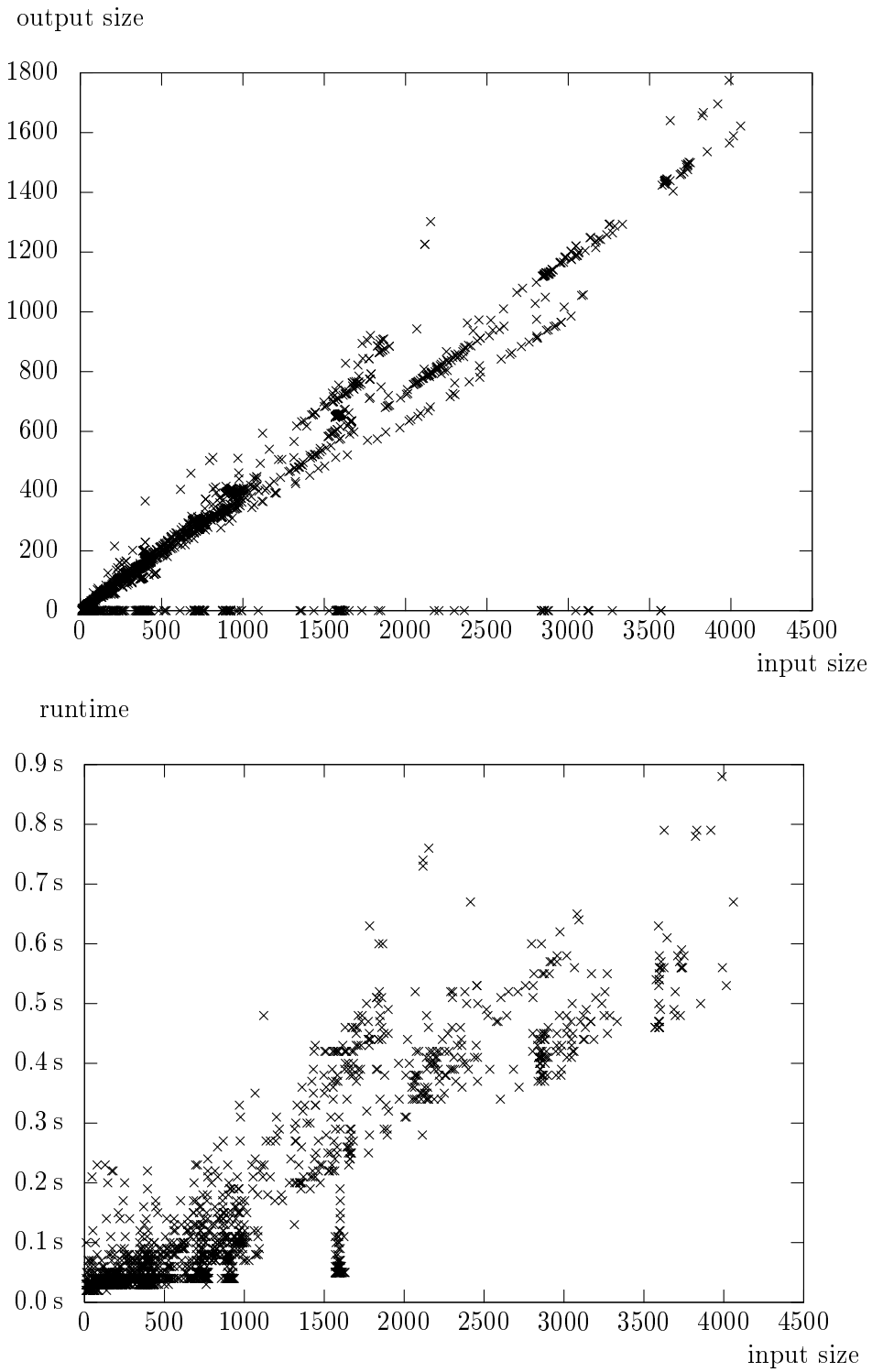


Figure 5.3.: Performance of unlifting in the Abrial benchmark

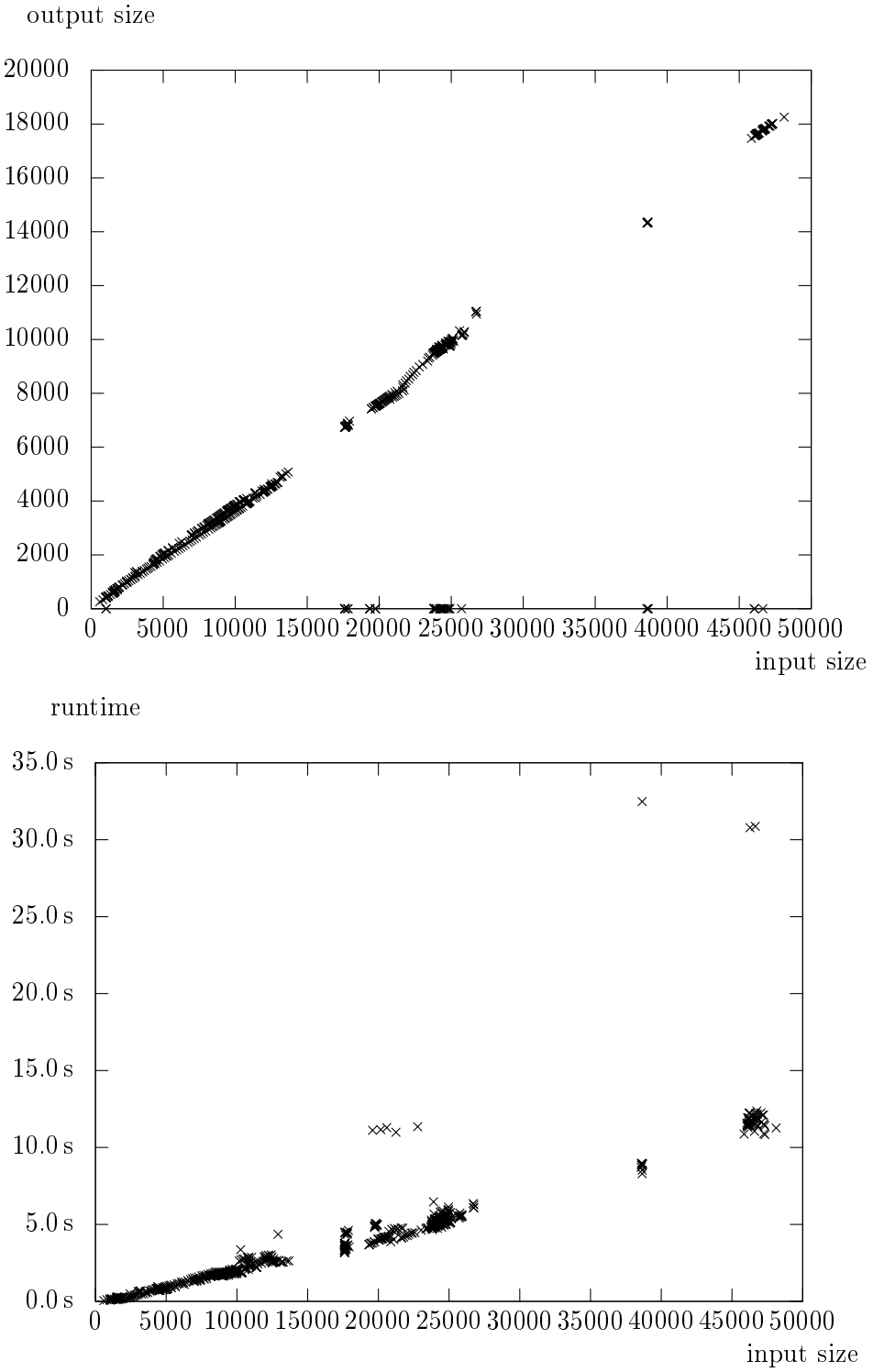


Figure 5.4.: Performance of unlifting in the BepiColombo benchmark

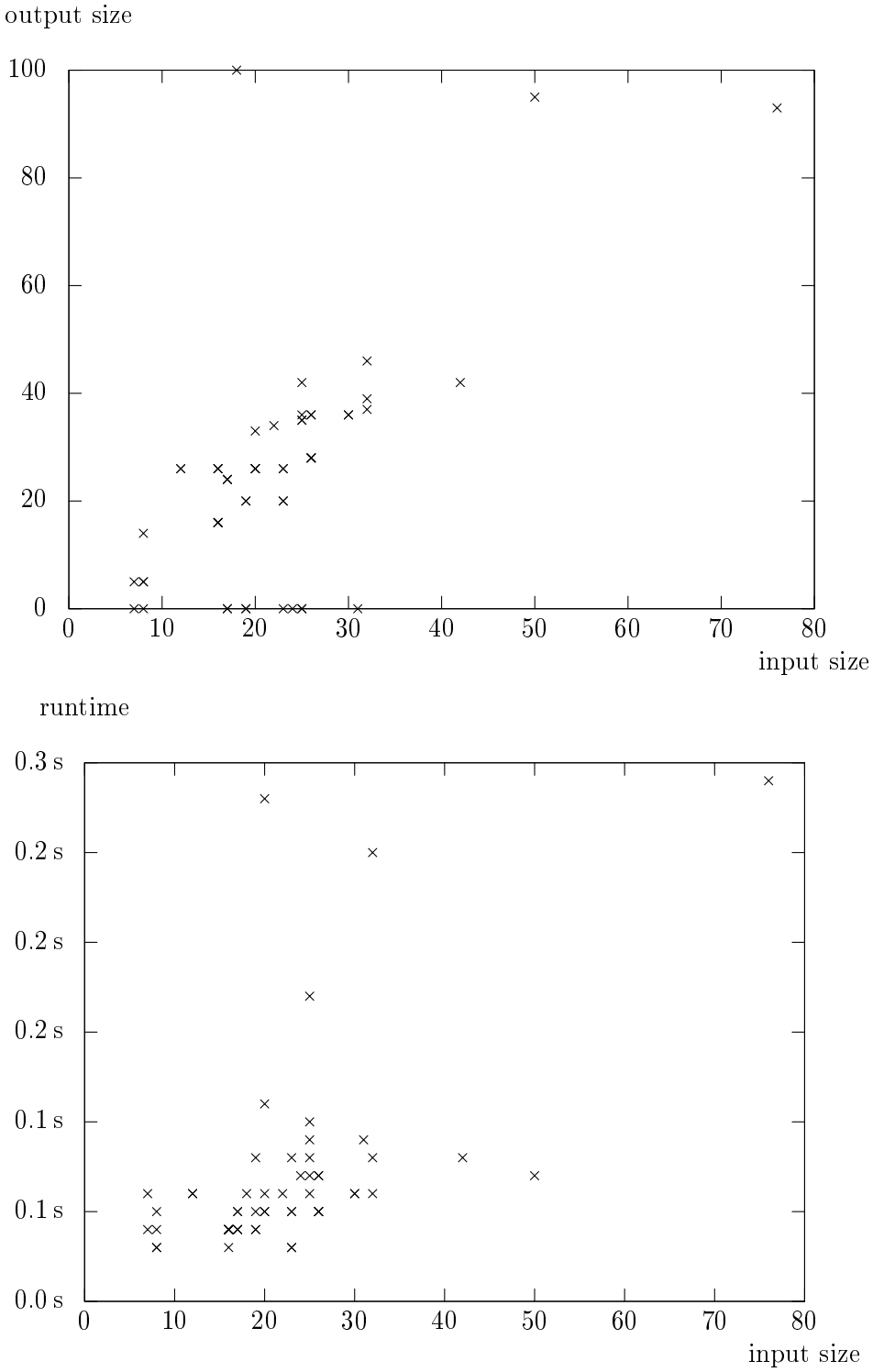


Figure 5.5.: Performance of unlifting in the Rules benchmark

5.3. Theorem Proving in Isabelle/HOL

After unlifting the proof obligation of interest it remains to prove it. One may hope that Isabelle's predefined proof methods are effective in this respect, but, at least if default configurations are used, the performance results are rather disappointing. In this section, I report on the process of developing a method whose performance is at least similar to and sometimes even better than the performance of Rodin's auto-tactic. The implementation is freely available [Schmalz].

5.3.1. Abrial Benchmark

5.3.1.1. Using Predefined Proof Methods

In the beginning, I tried to discharge the proof obligations of the Abrial benchmark by using one of Isabelle's predefined automated methods: `auto`, `blast`, `clarsimp`, `force`, `metis`, and `smt`. Further information on these methods can be obtained from [Paulson, 2011, Wenzel, 2011]. The `smt` method invokes the Z3 SMT solver [de Moura and Bjørner, 2008]; Isabelle accepts a proof generated by Z3 only if it can be reconstructed in Isabelle [Böhme and Weber, 2010].

The `smt` and `metis` methods support only a limited fragment of HOL's main theory and treat unsupported constants as uninterpreted. To improve the performance, I have introduced a preprocessing phase that tries to eliminate unsupported constants by unfolding their definitions. In the following, the terms `smt` and `metis` refer to Isabelle/HOL's `smt` and `metis` methods including this preprocessing phase. In some cases, a constant cannot be reasonably eliminated; examples include set comprehension and definite description (i.e., HOL's counterpart of funimg). In such cases, the constant remains unchanged. Although this procedure is quite ad hoc, the resulting version of `smt` is quite powerful (cf. Section 5.3.1.4). That said, it would be helpful to develop a more systematic approach to translate HOL formulae to the restricted theories supported by `smt` and `metis`.

Apart from the choice of method, the performance can also be improved by decreasing the *simplification depth limit*. Isabelle tries to solve preconditions of conditional rewrite rules by applying the simplifier recursively. Conditional rewriting may therefore lead to an infinite chain of recursive simplifier invocations. To prevent nontermination, the recursion depth of the simplifier is bounded by the simplification depth limit.

I also spent a considerable amount of time in tuning Isabelle's *clasimpset*, i.e., the set of available simplification, introduction, and elimination rules. During this process I recorded 4 versions of the clasimpset:

v0: Version 0 of the clasimpset is like Isabelle/HOL's default clasimpset, except that simplification rules are added to unfold the definitions of those constants that I have introduced during the development of EventB's standard model.

v1: Version 1 is the outcome of investigating Abrial's model of a file transfer protocol [Ch. 4 Abrial, 2010].

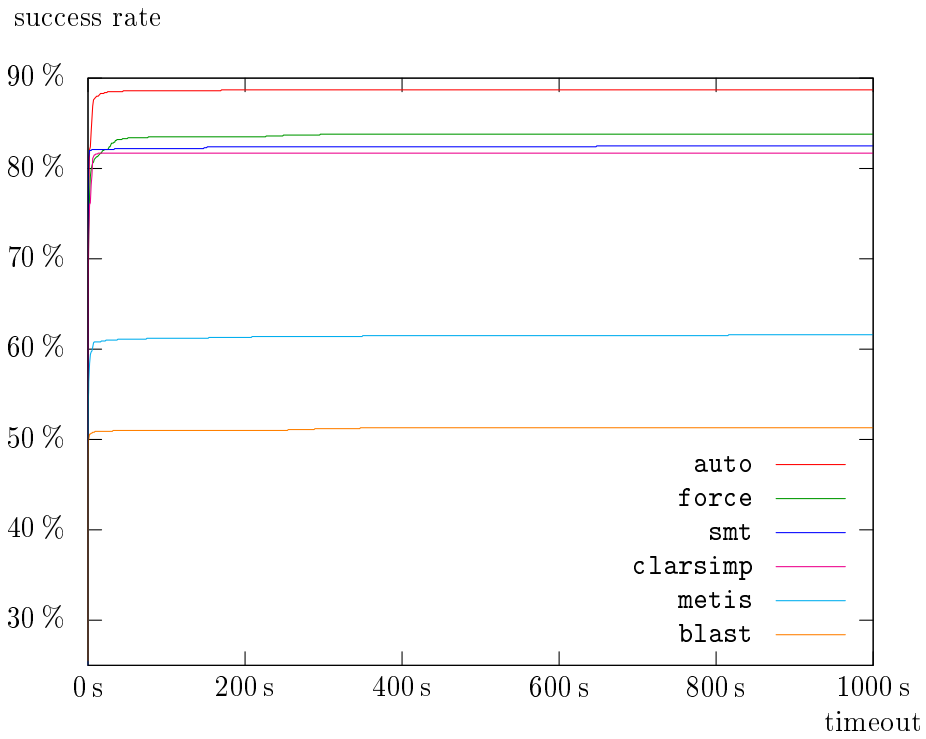


Figure 5.6.: Comparison between different methods with simplification depth limit 1 and Version 3 of the clasimpset on the Abrial benchmark

v2: Version 2 is the outcome of additionally investigating Abrial's model of a concurrent program [Ch. 7 Abrial, 2010].

v3: Version 2 covers only a small fraction of the constants that may arise during proof attempts. During the development of Version 3, I tried to define smart simplification, introduction, and elimination rules for all constants that may arise.

Based on extensive experimentation, I recommend to use `auto` with simplification depth limit 1 and Version 3 of the clasimpset: with this configuration, about 88% of the proof obligations are discharged automatically if the invocation of `auto` is aborted after a timeout of 10 seconds. To substantiate my recommendation, I present the performance of those configurations that result from changing one of the three parameters (method, simplification depth limit, version of clasimpset) while leaving the other two parameters unchanged. The results of these experiments can be found in Figures 5.6, 5.7, and 5.8. Of course, one can always try to find a better recommendation or justification by testing more configurations.

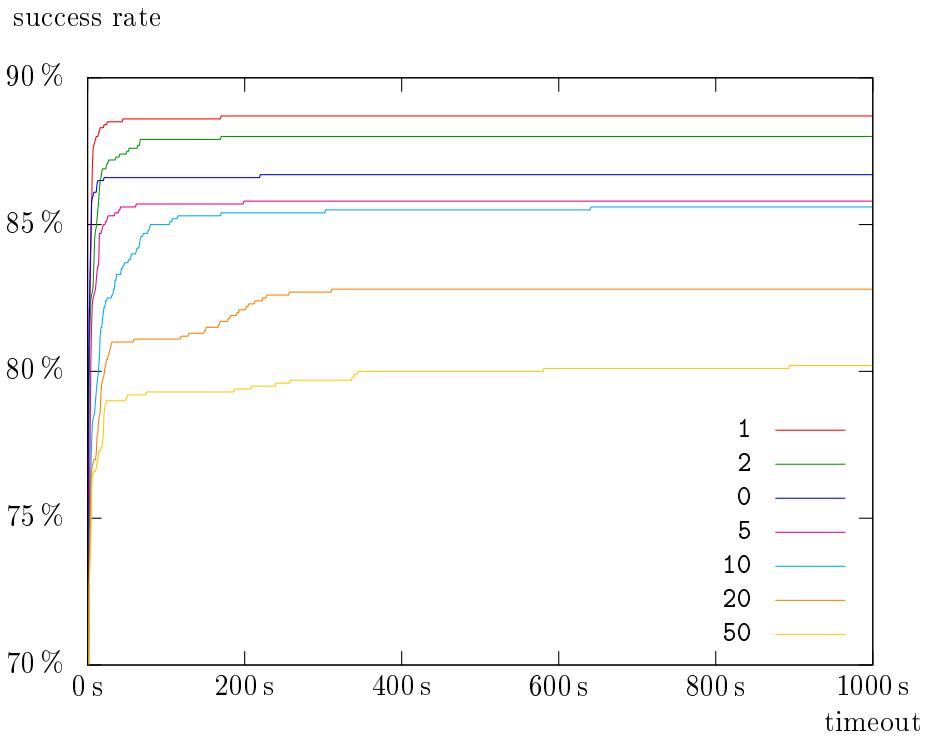


Figure 5.7.: Comparison between different simplification depth limits with `auto` and Version 3 of the clasimpset on the Abrial benchmark

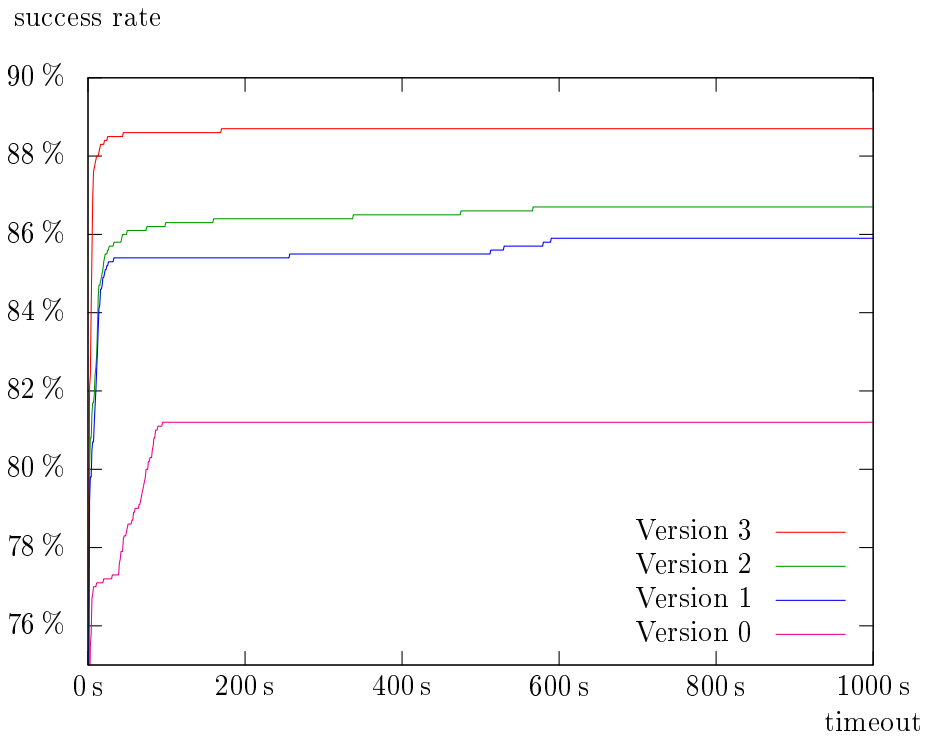


Figure 5.8.: Comparison between different clasimpsets with `auto` and simplification depth limit 1 on the Abrial benchmark

The development of Version 3 of the clasimpset does not involve a major secret. I have spent several weeks inspecting failed proof attempts and the trace information generated by automated proof methods. I identified proof steps that a human proof engineer would naturally apply, and translated these proof steps into rule declarations. I tried to avoid rules that increase the size of the proof state; in my view, such rules are useful in special situations, but often introduce new problems in the long run. Most of the enhancements could be encoded as rule declarations and only in a few cases I had to write ML code.

It was important to precisely understand the behavior of Isabelle's predefined proof methods. The manuals were a good starting point [Paulson, 2011, Wenzel, 2011], but sometimes it was necessary to study Isabelle's source code. Overall, tuning Isabelle's automated methods is slightly more difficult than performing manual proofs; but it is a skill that can be acquired within a couple of months.

A natural question is whether `auto` with simplification depth limit 1 and Version 3 of the clasimpset is a good choice in other benchmarks as well. To shed some light on this issue, I have tested several configurations on the BepiColombo benchmark. The main observations are depicted in Figures 5.9, 5.10, and 5.11. Note that the BepiColombo benchmark has not influenced the development of Versions 1–3 of the clasimpset.

Figure 5.9 indicates that a configuration with simplification depth limit 0 is most successful. This is not surprising, as the proof obligations of the BepiColombo benchmark are much bigger than the proof obligations of the Abrial benchmark. As in the Abrial benchmark, `auto` is the most powerful of the considered methods (cf. Figure 5.10), at least for timeouts of more than 200 seconds. According to Figure 5.11, Version 3 of the clasimpset is a clear improvement over earlier versions. But the differences between Versions 1 and 2 are hardly visible; this is presumably a consequence of the fact that Version 2 is an ad-hoc improvement over Version 1 that is inspired by the Abrial benchmark. At least, the changes in the clasimpset have never significantly corrupted `auto`'s performance.

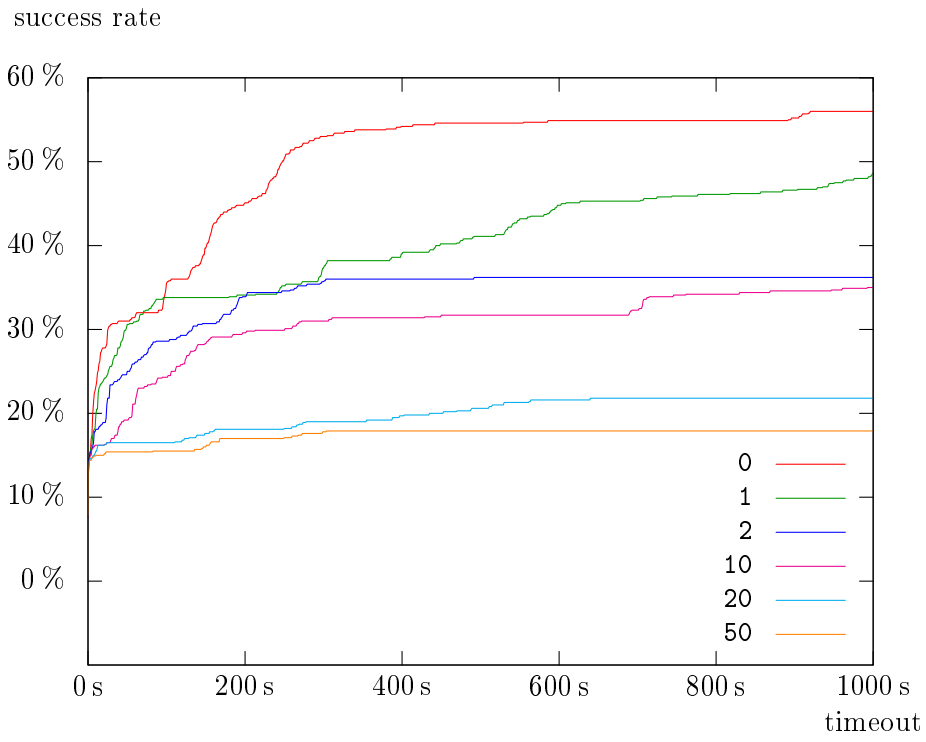


Figure 5.9.: Comparison between different simplification depth limits with `auto` and Version 3 of the clasimpset on the BepiColombo benchmark

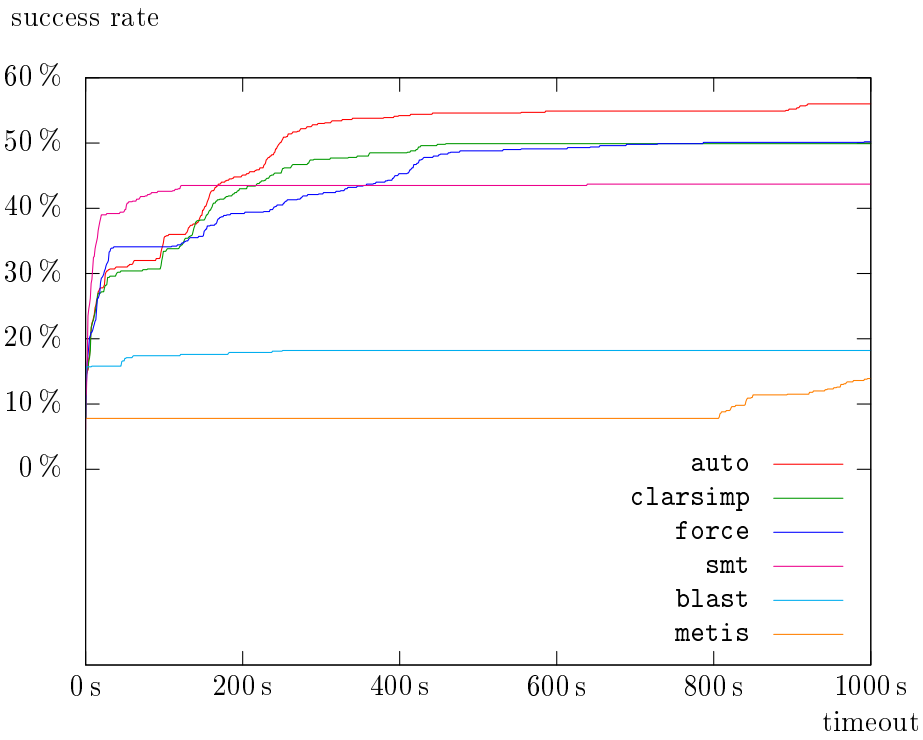


Figure 5.10.: Comparison between different methods with simplification depth limit 0 and Version 3 of the clasimpset on the BepiColombo benchmark

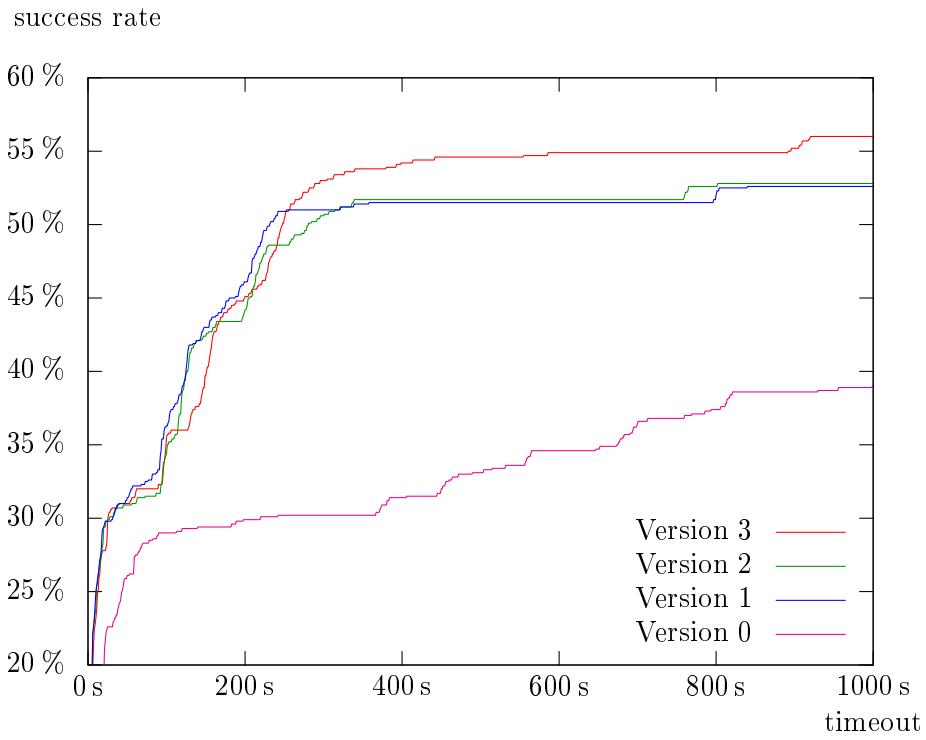


Figure 5.11.: Comparison between different clasimpsets with `auto` and simplification depth limit 0 on the BepiColombo benchmark

5.3.1.2. Design of Axe

During my experiments with Isabelle's predefined proof methods, I experienced the following problems and inconveniences:

1. It is tricky to find a "good" simplification depth limit. The recommendation to set the simplification depth limit to 1 (cf. Section 5.3.1.1) in the Abrial benchmark is the result of experiments that consumed vast amounts of processing time. In other benchmarks (such as BepiColombo), different simplification depth limits lead to a better performance. Common users do not have access to a high-performance computing cluster. Therefore, a heuristic that adapts the simplification depth limit automatically would be very useful.

2. Even with low simplification depth limits, the simplifier sometimes fails to terminate or is unreasonably slow. Since `auto` invokes the simplifier several times, a slow or non-terminating simplifier lets `auto` fail, even if a version of `auto` without simplification would discharge the given subgoal.

3. Some lemmas can only be proved by a combination of `auto`, `metis`, and `smt`.

4. The `auto` method applies *splitting rules*, i.e., rules that increase the number of subgoals, in a rather aggressive manner. An example is disjunction elimination, i.e., HOL's counterpart to disj_L. *Backjumping* [see e.g. Harrison, 2009b, p. 88] seems a promising solution, but I have refrained from implementing backjumping as it is a major undertaking. A more modest countermeasure is to restrict the set of splitting rules that are applied automatically and to delay the application of the remaining splitting rules as much as possible.

To tackle these problems, I have developed a new proof method: `axe`. Overall, `axe` executes the following phases:

1. simplification (`axe_simp`)

2. simplification and classical reasoning without splitting rules (`axe_clarsimp`)

3. simplification and classical reasoning with splitting rules (`axe_safesimp`)

4. Try to solve the remaining subgoals by applying a set of predefined tactics (`axe_solve`).

Simplification. The simplification phase of `axe` behaves quite similar to Isabelle's simplifier. One difference is that the execution of `axe_simp` is limited by a timeout. Another difference is that `axe_simp` repeatedly invokes Isabelle's simplifier with increasing simplification depth limits: the first invocation of the simplifier uses the

simplification depth limit zero[4], and in every subsequent invocation, the simplification depth limit is increased by one. If the timeout elapses while `axe_simp` is running the simplifier with simplification depth limit n, `axe_simp` aborts and outputs the result of invoking the simplifier with simplification depth limit $n-1$. (If $n = 0$, `axe_simp` fails.)

This approach has the advantage that a slow or non-terminating simplifier does not inhibit subsequent computations. If the simplifier terminates quickly with simplification depth limit 1 and fails to terminate with simplification depth limit 2, `axe_simp` still outputs the result of simplifying with simplification depth limit 1. Moreover, the user decides on the amount of time used for simplification instead of a simplification depth limit, which is a more intuitive decision.

For the sake of completeness, I want to mention that `axe_simp` also aborts if the simplification depth limit exceeds the so-called *maximum simplification depth limit* or if a certain number of subsequent simplifier invocations leave the proof state unchanged.

Classical Reasoning. The `axe_clarsimp` method is quite similar to the predefined `clarsimp` method. First, `axe_clarsimp` repeatedly applies non-splitting introduction and elimination rules (i.e., like `clarify`). To be precise, only rules that have been declared as *safe* are applied; there is an agreement that only rules that are safe in the sense of Section 4.1 should be declared as safe.

As soon as none of these introduction and elimination rules can be applied anymore, `axe_clarsimp` applies Isabelle's simplifier with simplification depth limit zero[5]. On failure, the simplification depth limit is repeatedly increased by one. On success, the entire process is restarted, i.e., `axe_clarsimp` passes on to applying introduction and elimination rules.

Like `axe_simp`, each execution of `axe_clarsimp` is limited by a maximum simplification depth limit and a timeout. If `axe_clarsimp` is aborted because the timeout has elapsed, `axe_clarsimp` outputs the intermediate proof state.

The `axe_safesimp` method behaves like `axe_clarsimp` with the exception that splitting rules are applied as well. Unlike `auto` or `safe`, `axe_safesimp` processes only the first available subgoal.

As for `axe_simp`, if an invocation of the simplifier fails to terminate, `axe_clarsimp` and `axe_safesimp` still terminate after the timeout and output an intermediate result. This limits the damage caused by a looping simplifier. I have decided to invoke `axe_clarsimp` before `axe_safesimp` to delay the application of splitting rules.

[4] An implementation note: if the simplification depth limit is zero, the simplifier is not invoked by "`simp?`", but instead by "`((simp (no_asm))?, (simp (no_asm_use))?, (simp (asm_lr))?, (simp (no_asm_use))?))`". This is admittedly a historical accident; I have rectified the situation during my subsequent work on the BepiColombo benchmark (cf. Section 5.3.2).

[5] If the simplification depth limit is zero, the simplifier is actually invoked by "`(simp (no_asm) | simp (no_asm_use) | simp (asm_lr) | simp)`".

Applying Solvers. The method `axe_solve` applies a list of tactics, called *solvers*, to solve the given subgoal. The executions of the solvers are limited by timeouts. If a solver fails to solve the subgoal, `axe_solve` discards its output (if there is any) and applies the next solver. If none of the solvers solves the subgoal, `axe_solve` terminates in failure. In the Abrial benchmark, `axe_solve` applies variants of `smt`, `metis`, `auto`, `force`, and `blast` (in this order).

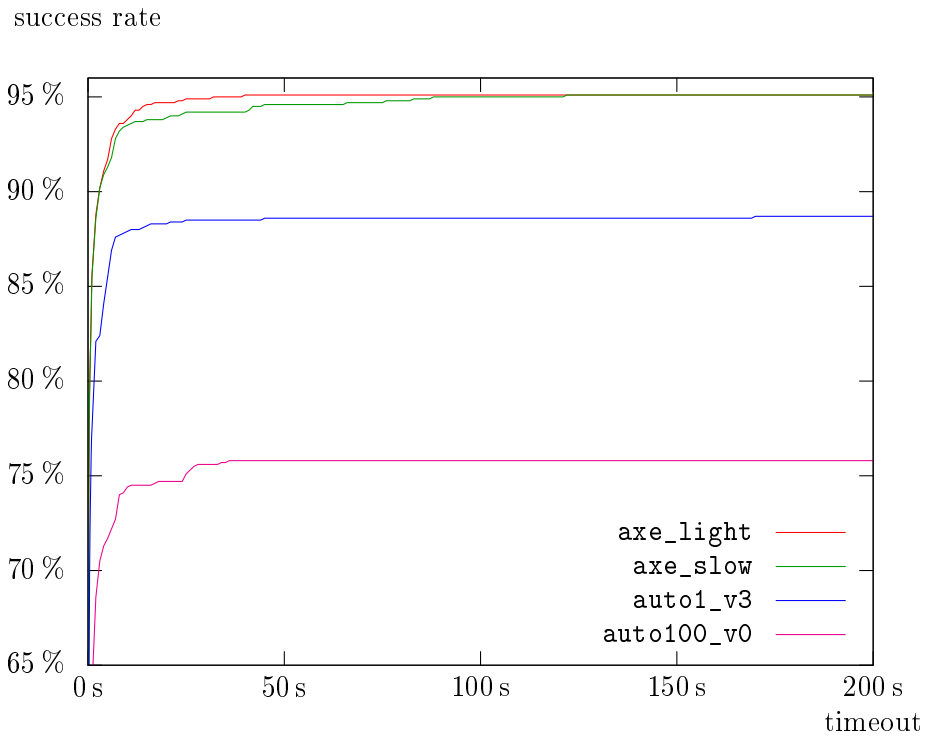
The interaction between `axe_safesimp` and `axe_solve` is as follows. The `axe` method applies `axe_solve` to the first subgoal left by `axe_safesimp`. If `axe_solve` solves the first subgoal, `axe_safesimp` is reinvoked on the new first subgoal (if there is any). The `axe` method terminates if no subgoal remains to be proved or `axe_solve` fails to prove the first subgoal, and outputs the (possibly empty) stack of remaining subgoals.

In contrast to `auto` or `safe`, `axe` focuses on the first subgoal. This does not affect the runtime on provable conjectures, but it significantly improves the runtime on unprovable conjectures: if the first subgoal cannot be solved, there is no point in trying to solve other subgoals.

5.3.1.3. Performance of Axe

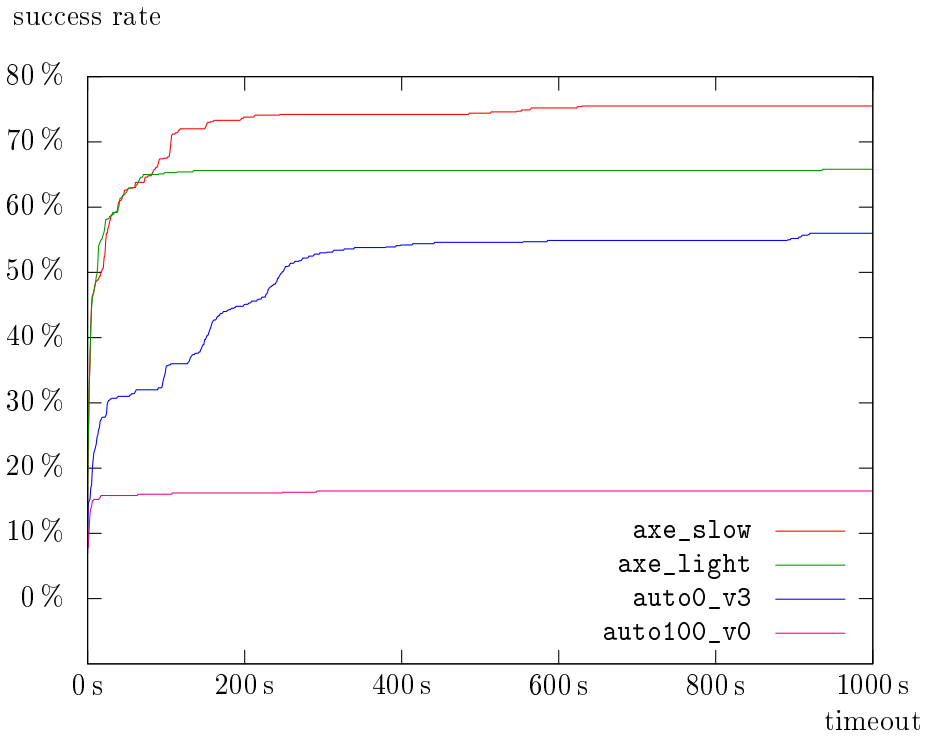
Figure 5.12 measures the impact of the main improvements driven by the Abrial benchmark. I consider two versions of `auto` and two versions of `axe`. The method `auto100_v0` is Isabelle's default configuration of `auto`, and `auto1_v3` is the recommendation of Section 5.3.1.1. The method `axe_slow` is a version of `axe` with generous timeouts, and `axe_light` is the result of optimizing timeouts. The figure shows that the recommendation of Section 5.3.1.1 are clear improvements over Isabelle's default configuration of `auto` and that `axe` is a clear improvement over `auto`: `axe_slow` proves 95% of the proof obligations with a timeout of 90 seconds, and with `axe_light`, this timeout can be reduced to 40 seconds.

Figure 5.13 compares `auto` with `axe` on the BepiColombo benchmark; the corresponding experiments have not influenced the development of `axe`. The figure confirms the results on the Abrial benchmark: `auto0_v3` is a clear improvement over `auto100_v0`, and `axe` is a clear improvement over `auto`. Since the proof obligations are much bigger than in the Abrial benchmark, the timeouts of `axe_light` are however too short.



- The `auto1_v3` method invokes `auto` with Version 3 of the clasimpset and simplification depth limit 1, i.e., with the simplification depth limit recommended in Section 5.3.1.1. `auto100_v0` invokes the default configuration of `auto`, i.e., with Version 0 of the clasimpset and simplification depth limit 100.
- The `axe_slow` method runs `axe` with Version 3 of the clasimpset. It executes `axe_simp` for up to 20 seconds, and `axe_clarsimp` and `axe_safesimp` for up to 40 seconds, respectively. The simplification depth limit is increased up to 5, and `axe_simp` aborts after 5 subsequent failures. The list of solvers is `smt`, `metis`, `auto`, `force`, `blast` (in this order), and each solver is invoked with a timeout of 40 seconds.
- The `axe_light` method differs from `axe_slow` by running `axe_simp` for up to 10 seconds, `axe_clarsimp` and `axe_safesimp` for up to 5 seconds, and each solver for up to 10 seconds. Moreover, `axe_simp` is already aborted after 2 subsequent failures.

Figure 5.12.: Comparison of `axe` with `auto` on the Abrial benchmark



The `auto0_v3` method invokes `auto` with Version 3 of the clasimpset and simplification depth limit 0, i.e., with the simplification depth limit recommended in Section 5.3.1.1. The other methods are defined in Figure 5.12.

Figure 5.13.: Comparison of `axe` with `auto` on the BepiColombo benchmark

5.3.1.4. Impact of Design Decisions

At this point, I analyze the impact of various decisions underlying the design of `axe`. The impact is measured by comparing modified versions of `axe` (or the used clasimpset) with the configuration of `axe` recommended in the previous section. In the case of the Abrial benchmark, modified versions of `axe` are compared to `axe_light`, and in the case of the BepiColombo benchmark, to `axe_slow`; both are defined in Figure 5.12.

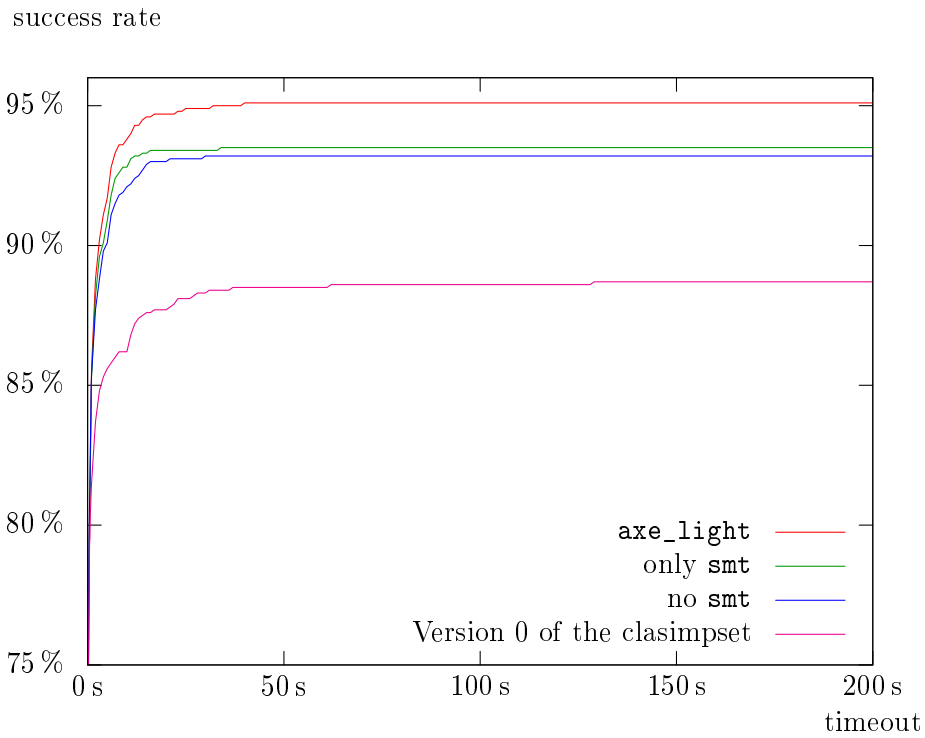
It was important to optimize the clasimpset: if `axe` is invoked with Version 0 of the clasimpset, the number of automatically proved conjectures drops by 7 percentage points in the Abrial benchmark (cf. Figure 5.14) and by 26 percentage points in the BepiColombo benchmark (cf. Figure 5.15). So unfolding definitions, the strategy underlying Version 0, is far from optimal. Although the BepiColombo benchmark has not influenced the development of Version 3, the benefits of Version 3 generalize well to the BepiColombo benchmark.

Another question concerns the impact of `smt`. In both benchmarks (cf. Figures 5.14 and 5.15), the impact of the solvers other than `smt` is quite small: if `axe_solve` invokes only `smt`, the performance drops by at most 2 percentage points. In the Abrial benchmark, a configuration of `axe_solve` that invokes every solver except `smt` (i.e., `metis`, `auto`, `force`, and `blast`) is almost as powerful as the configuration with only `smt`. But in the BepiColombo benchmark, the configuration of `axe_solve` without `smt` is significantly less powerful than the configuration with only `smt`: the difference is about 8 percentage points.

In Figures 5.16 and 5.17, I compare the impact of various strategies for applying splitting rules. *Eager splitting* means that `axe` skips the invocation of `axe_clarsimp`; splitting rules are consequently applied more eagerly than in the default version of `axe`. In Version 3 of the clasimpset, splitting rules that are triggered by a hypothesis (such as disjunction elimination, i.e., HOL's counterpart to `disj_L`) have been disabled in `axe_safesimp`; such rules are still used in some way by the various solvers[6]. *Splitting on hypotheses* means that `axe` uses Version 3 of the clasimpset except that `axe_safesimp` also applies the splitting rules of Isabelle/HOL's default clasimpset that are triggered by a hypothesis.

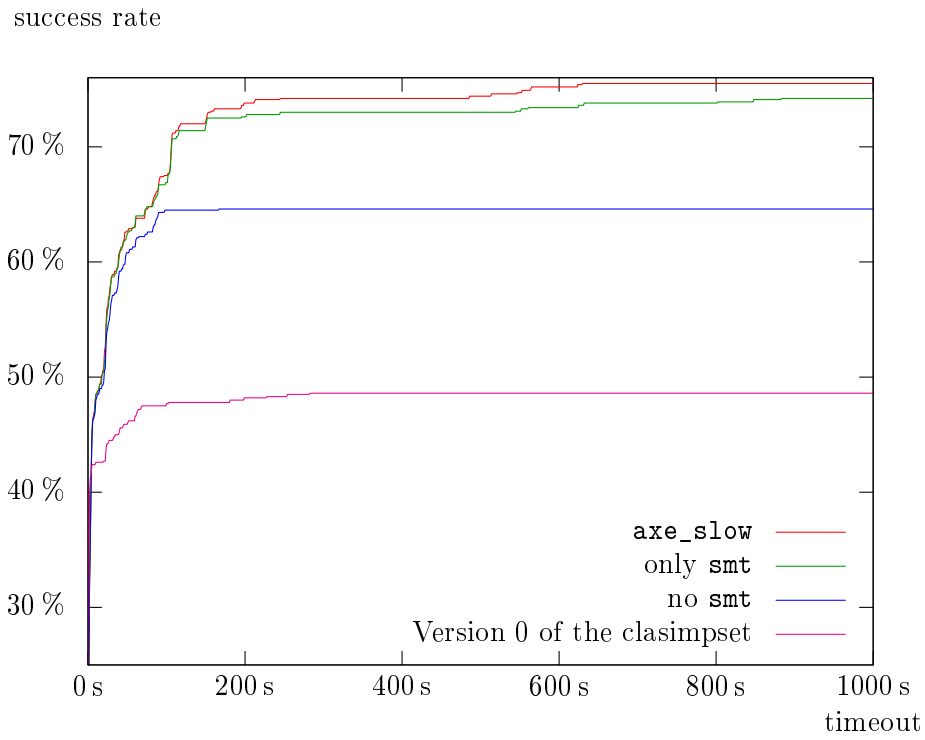
In the Abrial benchmark (cf. Figure 5.16), the choice of splitting strategy affects one percentage point of the proof obligations. Stated in absolute numbers, I could reduce the number of unproved proof obligations from about 120 to about 100 by delaying or avoiding the application of splitting rules. The effect is visible but hardly worth the effort. The improvement is much more evident in the BepiColombo benchmark (cf. Figure 5.17): the difference between the most aggressive strategy and the most conservative strategy is up to 12 percentage points. Interestingly, the most conservative strategy is most efficient in both benchmarks.

[6] The `smt` solver applies disjunction elimination as part of the underlying proof calculus. The `metis` solver does not apply disjunction elimination literally, but the resolution calculus has a similar effect. The `auto`, `force`, and `blast` solver apply splitting rules that are triggered by a hypothesis as unsafe elimination rules.



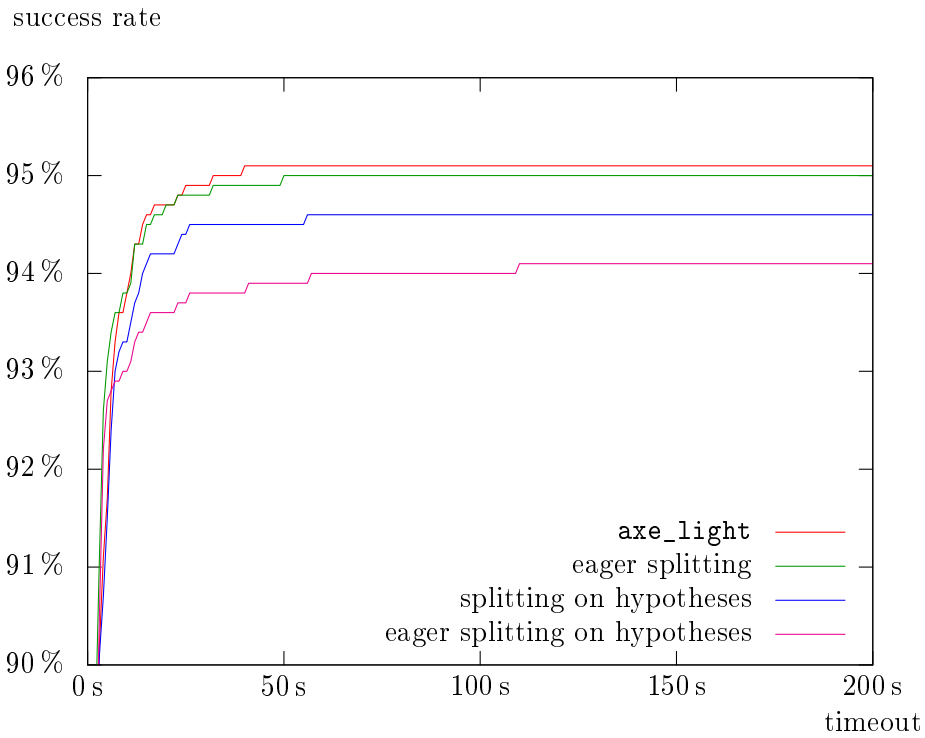
The `axe_light` method is the version of `axe` defined in Figure 5.12. The other methods differ from `axe_light` as indicated by their names.

Figure 5.14.: Variants of `axe` on the Abrial benchmark



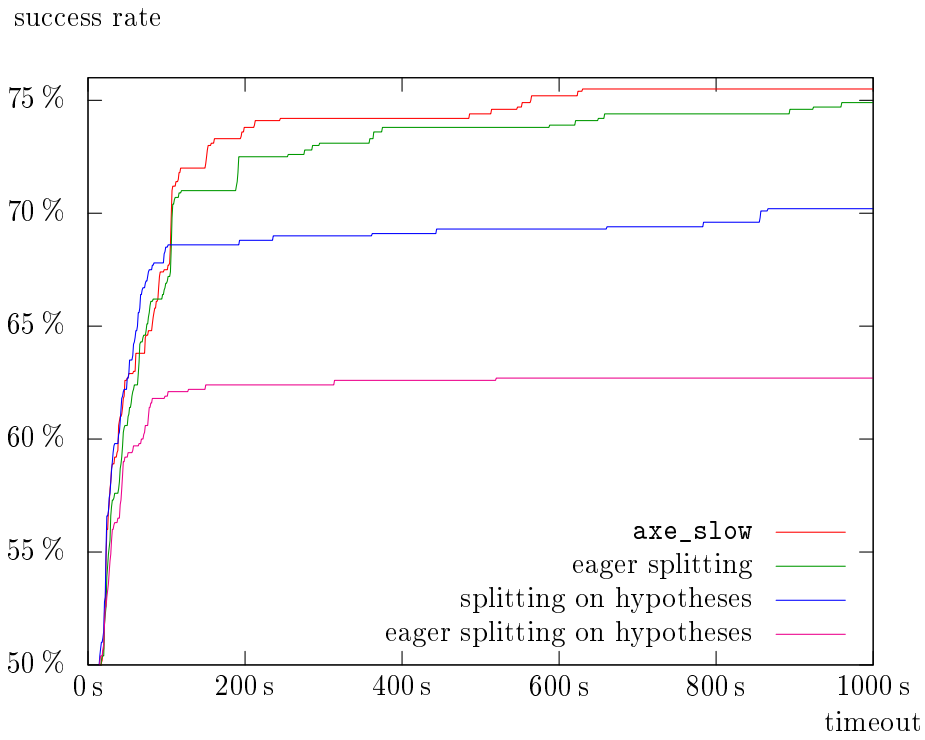
The `axe_slow` method is the version of `axe` defined in Figure 5.12. The other configurations differ from `axe_slow` as indicated by their names.

Figure 5.15.: Variants of `axe` on the BepiColombo benchmark



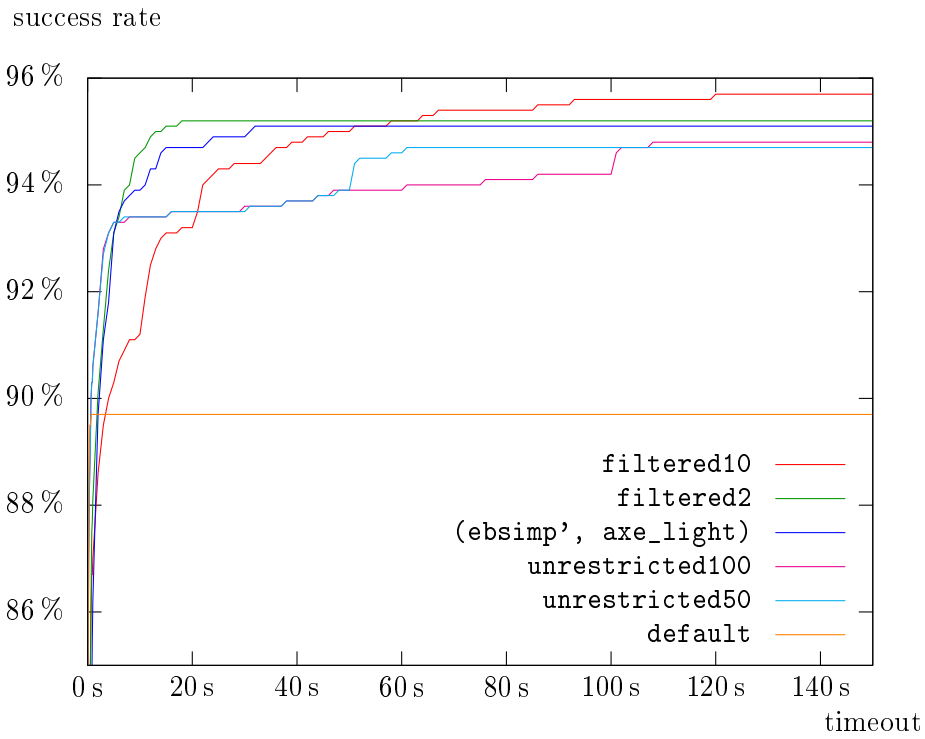
The `axe_light` method is the version of `axe` defined in Figure 5.12. The other configurations differ from `axe_light` as indicated by their names.

Figure 5.16.: Different splitting strategies on the Abrial benchmark



The `axe_slow` method is the version of `axe` defined in Figure 5.12. The other configurations differ from `axe_slow` as indicated by their names.

Figure 5.17.: Different splitting strategies on the BepiColombo benchmark



The method `(ebsimp', axe_light)` refers to the sequential composition of `ebsimp'` and `axe` with the configuration `axe_light`.

Figure 5.18.: Comparison of `axe` with Rodin's auto-tactic on the Abrial benchmark

5.3.1.5. Comparison to Rodin

Finally, I compare the performance of `axe_light` with the performance of Rodin's auto-tactic. For the reasons explained in Section 5.1.2, I have performed the corresponding experiments on a HP Compaq 6910p laptop. Since unlifting is a non-negligible part of the proof task, I compare the performance of Rodin's auto-tactic with the performance of applying the sequential composition of `ebsimp'` and `axe`. Figure 5.18 displays the main results.

The term `default` refers to the default configuration of Rodin's auto-tactic. The `default` configuration discharges less than 90% of the available proof obligations, but it is extremely fast.

A comparison to `default` is not very informative because it is easy to find configurations that are stronger than `default`. The configuration `unrestricted50` is an example. It uses most of the available features and invokes Rodin's external provers (i.e., newPP, PP, and ML) with the rather generous timeout of 50 seconds. Further information on the used configurations of Rodin's auto-tactic can be found in Appendix C. The configuration `unrestricted50` discharges almost 95% of the proof

obligations within 60 seconds per proof obligation. By increasing the timeouts of external provers to 100 seconds (cf. `unrestricted100`), the number of automatically discharged proof obligations can be only slightly increased. Both `unrestricted50` and `unrestricted100` are weaker than `axe_light`.

The configuration `filtered2` is a further improvement; it applies relevance filters [Röder, 2010], i.e., heuristics that remove hypotheses from the sequent that are assumed to be unhelpful during the proof search. Rodin's external provers are invoked with a timeout of 2 seconds. If the timeouts of Rodin's external provers are increased to 10 seconds (cf. `filtered10`), more proof obligations are discharged automatically, but `filtered10` is slower than `filtered2`. The configuration `filtered2` is slightly more powerful than `axe_light`, and `filtered10` is more powerful than `axe_light` for timeouts of more than 60 seconds.

There are many more ways of configuring Rodin's auto-tactic – too many to test them all. The configurations `filtered2` and `filtered10` are reasonable but maybe not optimal. If there exist more powerful configurations, they are not well-known and hard to find. I view this as a problem of Rodin, not of the comparison.

My hope was of course that `axe` outperforms even the best of Rodin's auto-tactic configurations. This is the case for configurations that do not use relevance filtering. (Note that `axe` does not use relevance filtering either.) If relevance filtering is used, Rodin's auto-tactic is slightly more powerful than `axe`. Since the differences are rather small, I expect that `axe` can be further improved to outperform `filtered10`.

5.3.2. BepiColombo Benchmark

The `axe` method discharges almost 76% of the proof obligations in the BepiColombo benchmark (cf. Figure 5.13). But it would be surprising if there were no room for further improvements. In this section, I report on the process of optimizing `axe` for BepiColombo.

5.3.2.1. Further Improvements of Axe

To maximize the impact of optimizations, I have focused on those proof obligations of BepiColombo that are not discharged automatically by some version of Rodin's auto-tactic; there are 132 such proof obligations, which I refer to as *difficult* proof obligations[7]. The set of difficult proof obligations is randomly partitioned into two sets: a *training* set and a *validation* set. The elements of the training set are used to discover undesired behavior or missing simplification, introduction, and elimination rules. To rule out overfitting, the performance of the various improved versions of `axe` is measured on the validation set after finishing the process of optimizing `axe`. The training set contains 33 proof obligations; the size of training set has mainly been

[7] The choice of auto-tactic configuration used to determine difficult proof obligations is somewhat ad-hoc. When I carried out the experiments, it was the best known configuration. Later, I discovered a slightly more powerful configuration, according to which only 123 proof obligations are difficult.

influenced by the available time for carrying out the experiments (i.e., 100 working hours).

During the process of optimization, I implemented 4 additional versions of the clasimpset, i.e., Versions 4–7, and an enhanced version of `axe`, called `axe'`[8]. Figures 5.19 and 5.20 show that newer versions of the clasimpset or `axe` are improvements over older versions, both on the training and the validation set. Of course, that does not show that each implemented change is indeed an enhancement, but the combination of all changes has a clear positive effect. To give the reader a feeling of how I have achieved this positive effect, I summarize the main changes.

Version 4. In Version 4 of the clasimpset, the distributivity law

$$(\exists x.\ \varphi\ x \lor \psi\ x) \longleftrightarrow (\exists x.\ \varphi\ x) \lor (\exists x.\ \psi\ x)$$

for $\exists$ and $\lor$ is used to postpone non-trivial instantiations of quantifiers or case splits: a goal that matches the left-hand side of the rule is rewritten to the right-hand side, and a hypothesis that matches the right-hand side is rewritten to the left-hand side. The distributivity law for $\forall$ and $\land$ is used in a similar way.

The simplifier tries to minimize the scope of quantifiers by *mini-scoping* rules such as

$$(\forall x.\ \varphi \lor \psi\ x) \longleftrightarrow \varphi \lor (\forall x.\ \psi\ x).$$

Version 4 introduces mini-scoping rules for conditionals such as

$$(\forall x.\ (\mathsf{if}\ \varphi\ \mathsf{then}\ \psi_1\ x\ \mathsf{else}\ \psi_2\ x)) \longleftrightarrow \mathsf{if}\ \varphi\ \mathsf{then}\ (\forall x.\ \psi_1\ x)\ \mathsf{else}\ (\forall x.\ \psi_2\ x).$$

Since Rodin does not support conditionals, $\mathsf{if}\ \varphi\ \mathsf{then}\ \psi_1\ \mathsf{else}\ \psi_2$ is usually expressed by $(\varphi \longrightarrow \psi_1) \land (\neg\varphi \longrightarrow \psi_2)$ or $(\varphi \land \psi_1) \lor (\neg\varphi \land \psi_2)$. Version 4 of the clasimpset also provides mini-scoping rules for these rephrased versions of conditionals.

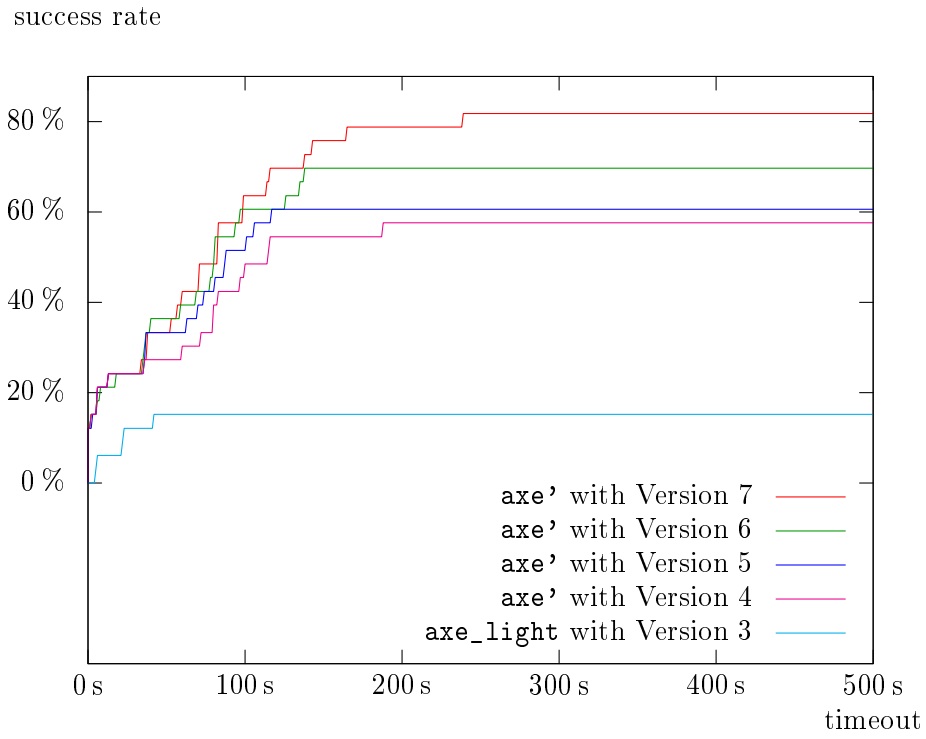
I also observed proof attempts in which the simplifier failed to terminate. Non-termination of the simplifier is often caused by a combination of factors. An important factor was that hypotheses of the form

$$\forall x.\ \varphi\ x \longrightarrow t\ x = u\ x \tag{5.8}$$

were automatically converted into conditional rewrite rules. The problem was that the left-hand side of the rewrite rule matched a subterm of the right-hand side. Version 4 of the clasimpset prevents this kind of non-termination by enforcing that the left-hand side of an automatically generated rewrite rule does not match a subterm of the right-hand side; this is achieved by exchanging $t\ x$ and $u\ x$ or replacing $t\ x = u\ x$ by $(t\ x = u\ x) = \mathsf{True}$.

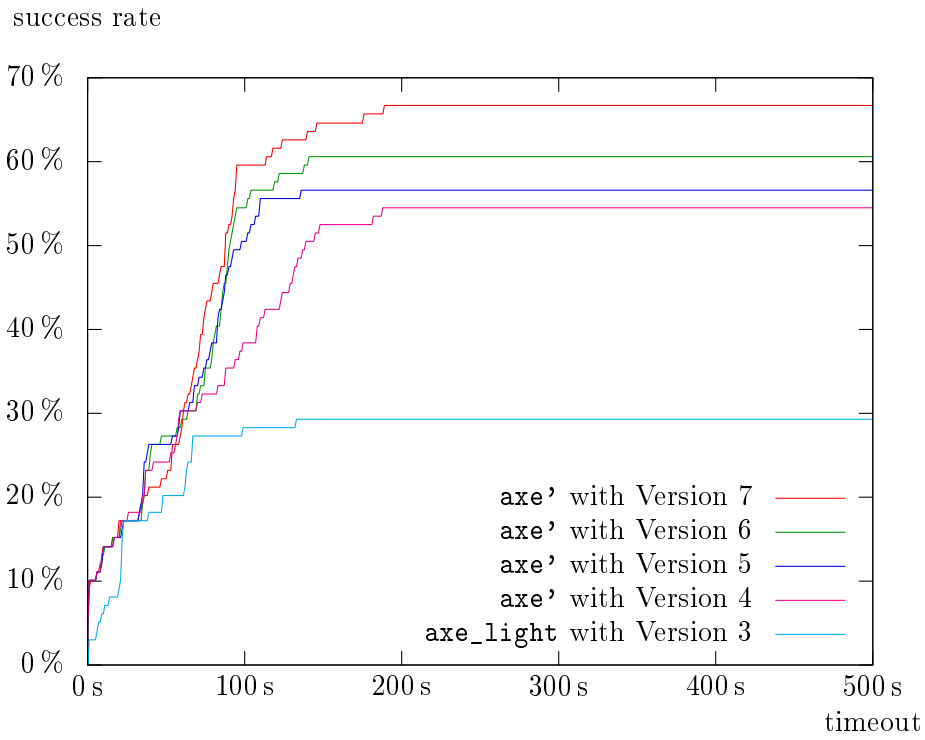
Together with the development of Version 4 of the clasimpset, I also changed the preprocessing of the input of `smt` and `metis`. Before, the extensionality axiom was applied aggressively to support the elimination of unsupported definitions. For

[8] In future releases, `axe` will be discontinued, and `axe'` will be renamed to `axe`.



- The `axe'` method invokes `axe_simp` with a maximum simplification depth limit of 5 and a timeout of 10 seconds, and aborts `axe_simp` after 5 subsequent failures. It invokes `axe_clarsimp` and `axe_safesimp` with timeouts of 180 seconds. It aborts `axe_clarsimp` if it fails to progress for 20 seconds, and `axe_safesimp` if it fails to progress for 5 seconds. The `axe_solve` method runs `smt`, `metis`, `auto`, `force`, and `blast` with timeouts of 40 seconds.
- The `axe_light` method is defined in Figure 5.12.

Figure 5.19.: Evolution of `axe` and the clasimpset on the training set



The parameters of `axe` and `axe'` are chosen as in Figure 5.19.

Figure 5.20.: Evolution of `axe` and the clasimpset on the validation set

example, the hypothesis $R = S$, where R, S are variables of set type, was rewritten to $\forall x.\ x \in R \longleftrightarrow x \in S$. But since `smt` and `metis` are unaware of extensionality, they failed to recognize that R and S were equal, which was sometimes harmful. Unfortunately, it is also harmful not to apply the extensionality axiom at all. As a compromise, I disabled application of extensionality to equalities $R = S$ if R and S are variables. This solution is helpful for BepiColombo, but there presumably remains room for improvements.

Another observation is that hypotheses $\mathsf{functional}\ r$ or $\mathsf{injective}\ r$ can be quite harmful if supplied to `smt` or `metis`. As a way out, I simply remove such hypotheses from the input of `smt` and `metis`. This is of course sometimes counterproductive, but at least in the training set of BepiColombo the benefits outweigh the drawbacks.

Since datatypes have not been available during the development of the BepiColombo model, a type PERSON with two inhabitants adam and eve is axiomatized (in HOL) by

$$\begin{aligned} &\mathsf{UNIV} = \{\mathsf{adam}, \mathsf{eve}\}, \\ &\mathsf{adam} \neq \mathsf{eve}. \end{aligned}$$

The Z3 solver behind `smt` has difficulties to cope with such a representation of finite types[9]. As a quick solution, the preprocessing phase of `smt` adds a hypothesis $t = \mathsf{adam} \vee t = \mathsf{eve}$ for every subterm of the given subgoal that does not contain bound variables. In the long run, it seems promising to solve this problem by using Isabelle/HOL's and Z3's support for datatypes.

Apart from that, Version 4 of the clasimpset introduces rules for reasoning about functional images and the converse of injective functions.

Enhancements of `axe`. During the development of Version 4 of the clasimpset, it became clear that some problems could be solved by improving `axe`; the enhanced version of `axe` is invoked with `axe'`. One observation was that *mutual* simplification [see Wenzel, 2011, p. 137] is quite expensive, even with a simplification depth limit of zero. *Mutual* simplification proceeds as follows: Simplify each hypothesis until no hypothesis can be further simplified; then simplify the goal. When a hypothesis is simplified, every other hypothesis is used as a rewrite rule. When the goal is simplified, every hypothesis is used as a rewrite rule.

The `axe` method already invokes the simplifier repeatedly with an increasing simplification depth limit. The `axe'` method further divides every invocation of the simplifier into the following four phases:

1. simplification of the goal `(simp (no_asm))`
2. simplification of all hypotheses without using other hypotheses as rewrite rules `(simp (no_asm_use))`

[9] For example, `smt` fails to prove $x = \mathsf{adam} \vee x = \mathsf{eve}$ from $\forall x.\ x = \mathsf{adam} \vee x = \mathsf{eve}$ in its default configuration.

3. left-to-right simplification: when a hypothesis is simplified, every preceding hypothesis is used as a rewrite rule; when the goal is simplified, every hypothesis is used as a rewrite rule. `(simp (asm_lr))`

4. mutual simplification `(simp)`

Interestingly, execution of the four phases is sometimes faster than a direct invocation of mutual simplification.

The `axe` method already limits the execution of `axe_clarsimp` and `axe_safesimp` by a timeout. The `axe'` method additionally aborts the execution of `axe_clarsimp` (or `axe_safesimp`) if it fails to change the proof state for a given amount of time. This sometimes allows `axe'` to abort the execution of `axe_clarsimp` or `axe_safesimp` earlier than `axe` without preventing useful proof steps.

While developing `axe'`, I discovered that `axe'` may fail to prove conjectures having a goal that is identical to one of the hypotheses. This problem arises if the first invocation of the simplifier changes the goal, subsequent invocations of the simplifier fail to terminate, and other sub-tactics of `axe'` fail to recognize that the goal equals one of the hypotheses. I am not sure whether this problem occurs frequently, but if it occurs it is rather embarrassing. To prevent this problem, I have introduced an additional phase `axe_triv` that is invoked right before `axe_simp`; `axe_triv` tries to apply a limited set of rules that prove the given subgoal in one or two steps.

Version 5. Version 5 of the clasimpset provides some rules that eliminate duplicated subterms. An example is

$$\varphi \vee (\varphi \wedge \psi) \longleftrightarrow \varphi.$$

It is unclear to me whether these rules increase the number of automatically proved conjectures, but they greatly improve readability of intermediate proof states.

Version 5 moreover improves the facilities of Version 3 for simplifying functional images whose first argument is given by an enumerated set. Using the notation of Event-B, an example is $\{1 \mapsto 2, 2 \mapsto 3\}(2)$, which is simplified to 3.

Version 6. Version 6 of the clasimpset provides a rule that rewrites $(\varphi \wedge \psi_1) \vee (\neg\varphi \wedge \psi_2)$ to $\mathsf{if}\ \varphi\ \mathsf{then}\ \psi_1\ \mathsf{else}\ \psi_2$ and a rule that replaces the two hypotheses $\varphi \longrightarrow \psi_1$ and $\neg\varphi \longrightarrow \psi_2$ by the hypothesis $\mathsf{if}\ \varphi\ \mathsf{then}\ \psi_1\ \mathsf{else}\ \psi_2$[10]. The details are rather delicate because the simplifier sometimes rewrites $\mathsf{if}\ \varphi\ \mathsf{then}\ \psi_1\ \mathsf{else}\ \psi_2$ to $(\varphi \longrightarrow \psi_1) \wedge (\neg\varphi \longrightarrow \psi_2)$. The advantage of such a transformation is that HOL's predefined rules for reasoning about conditionals can be applied to formulae that result from encoding conditionals with boolean connectives.

Version 7. In some cases, the given subgoal contained a hypothesis of the form $\mathsf{if}\ \varphi\ \mathsf{then}\ \psi_1\ \mathsf{else}\ \psi_2$ and it was helpful to perform a case split on φ. I already had the

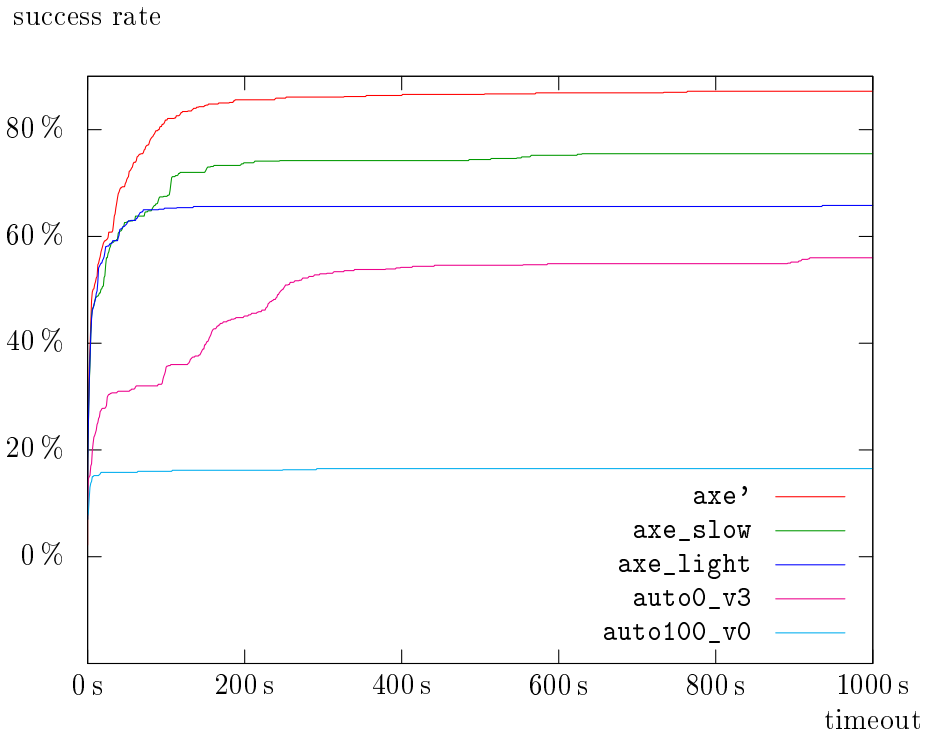
[10] Implementation note: introduction, elimination, and destruction rules that are triggered by more than one formula are implemented as classical wrappers.

intuition that case splits need to be applied with care; this was later on confirmed by the measurements of Section 5.3.1.4. To limit the negative impact of case splits, I introduced an additional phase into `axe'`, namely `axe_split`. If `axe_safesimp` fails to solve the given subgoal, `axe_split` tries to apply an elimination rule that increases the number of subgoals. In Version 7 of the clasimpset, `axe_split` tries to find a hypothesis of the form $\mathsf{if}\ \varphi\ \mathsf{then}\ \psi_1\ \mathsf{else}\ \psi_2$ and performs a case split on φ. If there is such a hypothesis, `axe'` continues with `axe_safesimp`. Otherwise, `axe'` tries to solve the subgoal with `axe_solve`. The `axe_split` method was initially an experimental feature; given that it is quite successful, I have decided to integrate it into `axe'`.

Validation. The improvements of the clasimpset and the development of `axe'` have been quite successful on the difficult proof obligations of the BepiColombo benchmark. The improvements remain successful on the validation set, which has not influenced the development of the improvements. Figure 5.21 compares the performance of `axe'` with earlier milestones on the full BepiColombo benchmark; it confirms that `axe'` with Version 7 of the clasimpset remains a clear improvement over `axe` with Version 3 of the clasimpset.

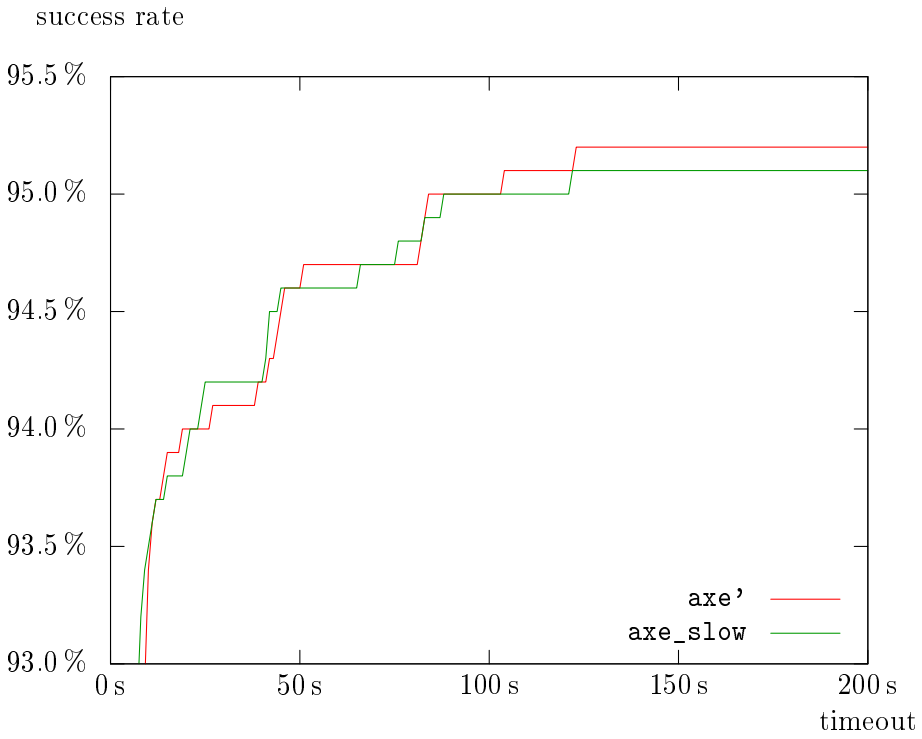
Concerning the Abrial benchmark, Figure 5.22 shows that `axe'` with Version 7 of the clasimpset is only a minor improvement over `axe` with Version 3 of the clasimpset. I compare `axe'` with `axe_slow` instead of `axe_light` (cf. Figure 5.12), because the timeouts of `axe'` have not been optimized for the Abrial benchmark and are therefore closer to the timeouts of `axe_slow`.

Apparently, the changes developed in this section are useful for the BepiColombo benchmark, but they are not very useful to find more proofs in the Abrial benchmark. This is not surprising to me. The performance of `axe` on the Abrial benchmark has already been quite good. It therefore requires very specific measure to improve the performance further; it is unlikely that these measures are discovered by investigating a different benchmark. I still view it as a success that the improvements on the BepiColombo benchmark do not degrade the performance on the Abrial benchmark.



The `axe'` method uses Version 7 of the clasimpset and the parameters given in Figure 5.19. The methods `axe_slow` and `axe_light` are the outcomes of my work on the Abrial benchmark. Their definitions can be found in Figure 5.12. The `auto0_v3` method is the preliminary recommendation of Section 5.3.1.1: `auto` with simplification depth limit 0 and Version 3 of the clasimpset. The `auto100_v0` method is Isabelle's default configuration: `auto` with simplification depth limit 100 and Version 0 of the clasimpset.

Figure 5.21.: Performance of `axe'` and Version 7 of the clasimpset on the full BepiColombo benchmark



The `axe'` method uses Version 7 of the clasimpset and the parameters given in Figure 5.19. The `axe_slow` method is defined in Figure 5.12.

Figure 5.22.: Performance of `axe'` and Version 7 of the clasimpset on the Abrial benchmark

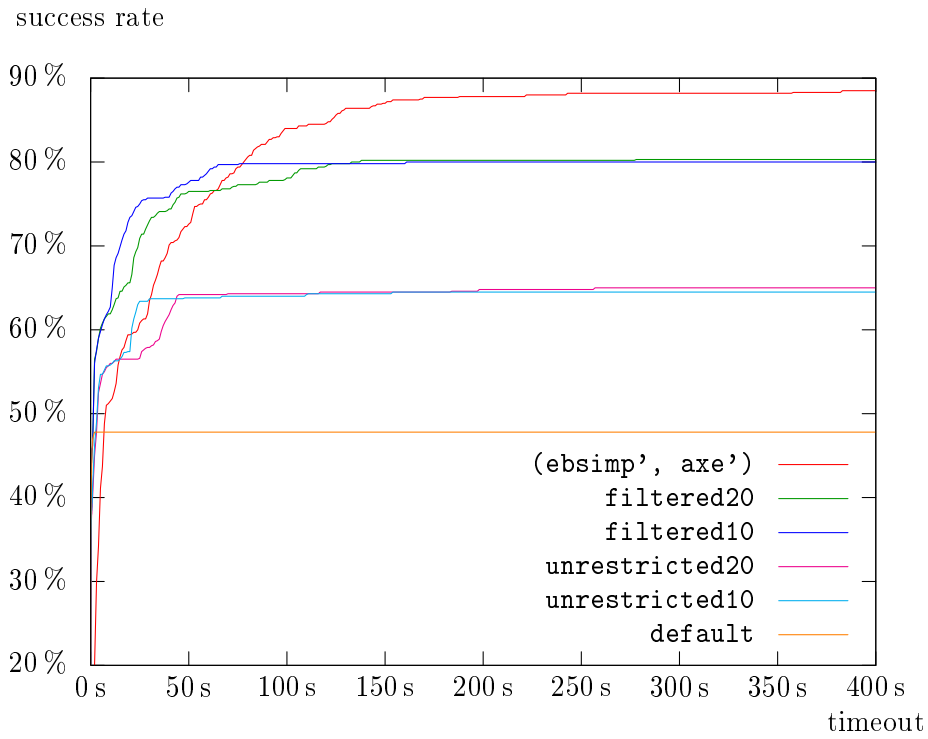


Figure 5.23.: Comparison of `axe'` with Rodin's auto-tactic on the BepiColombo benchmark

5.3.2.2. Comparison to Rodin

Similarly as in Section 5.3.1.5, I compare `axe'` with Rodin's default auto-tactic configuration (`default`), and improved tactics that do (`filtered10` and `filtered20`) and do not (`unrestricted10` and `unrestricted20`) use relevance filtering. The details on auto-tactic configurations are explained in Appendix C. As in Section 5.3.1.5, I have performed the experiments on a HP Compaq 6910p laptop (cf. Section 5.1.2). Figure 5.23 shows that `axe'` with Version 7 of the clasimpset (preceded by `ebsimp'`) discharges more proof obligations than the best known configurations of Rodin's auto-tactic. The `axe'` method is however sometimes slower than Rodin's auto-tactic, in particular if relevance filtering is used.

5.3.3. Limitations

Several problems have not been solved because of lack of time or suitable ideas; some of these problems seem not to be relevant to the considered benchmarks, but they may be relevant to others.

A technical problem stems from the cost of parsing formulae. The current toolchain uses Isabelle's generic parser, which sometimes takes several minutes to parse a proof obligation. This is clearly unacceptable. After finishing my experiments, a more efficient method for parsing formulae has been implemented[11]. By using this method, I expect that the parsing time can be reduced by several orders of magnitude.

Another problem is the number of irrelevant hypotheses: `axe` processes all available hypotheses, although usually only a fraction is required to find a proof. I expect that the performance of `axe` can be improved by heuristically removing some of the available hypotheses before starting the proof attempt or by delaying the processing of hypotheses that seem unlikely to be useful for a proof. Such techniques have been implemented elsewhere [such as in Hoder and Voronkov, 2011, Meng and Paulson, 2009, Röder, 2010]; it remains to implement them and to find suitable parameters.

Another problem concerns the application of splitting rules. The current version of `axe` prevents or delays the application of splitting rules such as disjunction elimination. It is difficult to construct a simple example where this behavior lets a proof attempt fail, because several heuristics reduce the need for applying splitting rules. Nevertheless, it seems plausible that `axe` is unable to find proofs that require excessively many applications of splitting rules. A way out is to implement a more sophisticated splitting strategy such as backjumping [see Harrison, 2009a], which is of course a major undertaking.

Before invoking the `metis` or `smt` method, I apply a preprocessing phase that unfolds the definitions of most constants that are not supported by `smt` or `metis`. Although this kind of preprocessing is effective, the approach is quite ad hoc and leaves room for improvements.

5.3.4. Conclusions

Out of the box, Isabelle/HOL has not been useful as an automated theorem prover for Rodin: the `auto` method discharges 76% of the proof obligations in the Abrial benchmark and 17% in the BepiColombo benchmark. But Isabelle/HOL provides a great support for improving existing and developing new proof methods. In a first step, I have developed the `axe` method, which is essentially a smart combination of predefined tactics, and I have defined several introduction, elimination, and simplification rules tailored to the HOL theory underlying Event-B's standard model. To be precise, I have also developed a few so-called classical wrappers [Paulson, 2011]. The development has been driven by the Abrial benchmark. After these improvements, `axe` discharges 95% of the proof obligations in the Abrial benchmark and almost 76% in the BepiColombo benchmark. The performance improvements on the Abrial

[11] I am grateful to Makarius Wenzel for his quick reaction on my feature request.

benchmark generalize well to the BepiColombo benchmark, and the overall performance (on the Abrial benchmark) is almost as good as the performance of Rodin's auto-tactic.

Next, I have further improved the performance of `axe`; the process was driven by a training set, i.e., a small fragment of the BepiColombo benchmark that consists of 33 proof obligations. Within about 100 working hours, I could increase the number of automatically discharged proof obligations from 5 to 27 on the training set, and from 411 to 545 on the entire BepiColombo benchmark. This means, whenever I turned a failed proof attempt of the training set into an automatic proof, on average about 5 failed proof attempts of the remaining BepiColombo benchmark became automatic as well: "buy one, get five for free". This calculation still gives an underapproximation, because it assumes that I investigated 22 proof obligations of the training set; in fact, it was less than 22, because of generalization effects within the training set. In the end, Rodin's auto-tactic still performs better than `axe` for timeouts of up to 75 seconds, but `axe` outperforms Rodin for timeouts of more than 80 seconds.

The main arguments in favor of Isabelle/HOL and `axe` are strong guarantees that proofs are correct and the great support for extending and configuring automated methods. One could try to carry the performance improvements inspired by the BepiColombo benchmark over to Rodin's auto-tactic; but that would take significantly more time than 100 working hours. The various improvements often generalize well to proof obligations that have not influenced their development; I encountered the situation that a later version of a tactic or configuration was not an improvement over an earlier version, but I never experienced that a later version performed significantly worse than an earlier version. Another argument in favor of `axe` is that its performance is not bad (in comparison to Rodin's auto-tactic) and sometimes even clearly better.

The main argument against Isabelle/HOL and `axe` is that its performance was always worse than the performance of Rodin's auto-tactic before domain specific improvements had been developed. This is not surprising, since Rodin's auto-tactic has been optimized during several years and `axe` only during months. The price of improving the performance of `axe` is that a developer has to become familiar with Isabelle/HOL. Overall, users should consider using Isabelle/HOL and `axe` if soundness is of utmost importance or when Rodin's auto-tactic performs poorly.

Coming back to the problems mentioned at the beginning of the chapter, the integration of Isabelle/HOL gives Rodin a theorem prover that is sound by construction and has a complete proof calculus (in the general sense). On the considered benchmarks, Isabelle/HOL discharges either roughly as many proof obligation as Rodin's auto-tactic or sometimes even significantly more; the price is that the proof search by Isabelle/HOL sometimes takes more runtime than the proof search by Rodin. Adaptability is a major strength of Isabelle/HOL.

6. Conclusions

6.1. Summary of Contributions

I have achieved the goals stated in the problem statement of Section 1.3 in the following way. In Sections 2.1 and 2.2, I develop definitions of syntax and semantics of Event-B's logic, viewing the underlying signature and structure as unknown. In Sections 3.1 and 3.3, I instantiate these generic definitions by giving a signature and structure covering the symbols that are by default available in Rodin. The impact of non-trivial design decisions underlying Event-B's logic is analyzed in Chapter 4.

The definition of Event-B's logic is close to its implementation in Rodin; exceptions are described and justified in Sections 2.3.2.4, 2.4, 3.4, and 4.3. Readers may validate the given semantics by inspecting its definition (Section 2.2) and the corresponding proof calculus in Sections 3.1 and 3.3.

I have demonstrated with various examples that my specification of syntax and semantics is suitable for developing and understanding other features of Event-B's logic:

- a definition of substitution with a semantic justification (Section 2.2.7),
- a rigorous account on proof calculi in Event-B, explaining what the different kinds of rules are, how they may be applied during proofs, and when their application is sound (Section 2.3),
- a soundness proof of a concrete proof calculus, verified to a large extent by Isabelle/HOL (Chapter 3),
- a development of methods for extending theories and a proof that these methods are definitional (Section 3.2),
- an embedding into the monotonic fragment of Event-B's logic (Section 3.2.4).

I have improved Rodin's theorem proving capabilities by integrating Isabelle/HOL as an automated theorem prover and by developing the `axe` method (Chapter 5). Concerning soundness, Isabelle/HOL meets high standards because it implements the LCF approach, it is committed to definitional theory extensions, and it has matured during several decades. My integration of Isabelle/HOL into Rodin adopts this trustworthiness to a large extent because the actual embedding of Event-B's logic into HOL is quite simple; the complicated parts of the embedding have been implemented as Isabelle tactics and are therefore correct by construction. Since there

is no need to constantly change the implementation of the embedding, it is unlikely that new soundness bugs will be introduced in the future.

I have evaluated my Isabelle/HOL based tactic on proof obligations from various domains, in particular on the proof obligations of BepiColombo, which is one of the most challenging industrial case studies to which Rodin has been applied. The evaluation shows that Isabelle/HOL and `axe` are an improvement over Rodin's auto-tactic in terms of adaptability and – in the case of BepiColombo – even in the number of automatically discharged proof obligations.

Impact on Event-B and Rodin. The results of this thesis and the process of doing this research have impacted Event-B and Rodin in several ways. Several consistency bugs have been discovered and corrected. The discovery of such bugs and the parallel development of theoretical foundations has initiated a learning process: over the years, the consistency bugs have become much more subtle.

By studying the foundations of term rewriting, it has become clear how to apply conditional rewrite rules to the arguments of binders, a feature that is so far not supported by Rodin. The study of proof calculi has also helped to eliminate dispensable preconditions of rules; a few examples can be found in Section A.3.

The basic extension methods implemented in the theory plug-in [Butler and Maamria, 2010] have now got a theoretical foundation. My work also shows how to overcome various restrictions when introducing new rules, most notably the restriction that $\wedge, \vee, \Rightarrow, \forall, \exists$ may not occur in the left-hand sides of rewrite rules.

The integration of Isabelle/HOL is not only useful for discharging proof obligations automatically, it can also be used to formally reason about the soundness of new rules. A better approach to soundness is of course to change Rodin's prover architecture to prevent application of unsound rules; but it is unclear whether and when such a change of architecture will take place. In the meantime, formal soundness proofs in HOL are a beneficial complement to the current review procedure that is based on human inspection.

Contributions to the Field of Theorem Proving. Although this research has been driven by the aim to improve Event-B and Rodin, it has led to results of a more general interest. One of these results is directed rewriting. Although this term rewriting technique has already been implemented in Rodin and PVS, it has been poorly understood. With my research, I point out under which conditions directed rewriting is sound (i.e., monotonicity and SW-semantics), I give empirical evidence that it improves the performance of term rewriting, and I show that it is safe (under weak preconditions), which is not obvious from the definition of directed rewriting.

Another result concerns unlifting. A common argument against Kleene semantics of first-order connectives and quantifiers is an exponential overhead of unlifting. With my parametric complexity analysis I point out that this exponential overhead occurs only in corner cases, and with my empirical evaluation I show that these corner cases do not arise in a wide variety of practical problems. Unlike other unlifting algorithms,

my algorithm is able to cope with functions whose domains are unknown.

Although the `axe` method has been designed to solve conjectures originating from Event-B, it is as generic as other automated methods in Isabelle. It has proved to be effective on big and somewhat shallow problems without guidance from the user. I expect that it is useful in application domains beyond Event-B.

6.2. Future Work

There are several ways of improving or extending the results obtained in this thesis. The various restrictions on definitional theory extension methods in Rodin need to be overcome; the foundations of more extension methods than given in this document can and should be developed, e.g., methods for defining new types and methods for recursive function definitions. The theory plug-in already allows users to define datatypes [Butler and Maamria, 2010], but the theoretical foundations have not yet been worked out; in fact, certain datatype "definitions" make the resulting theory inconsistent [Schmalz, 2012].

Another important problem is to make the integration of Isabelle compatible with definitional extensions. Most of Rodin's automated tactics, in particular the external provers, have difficulties coping with definitional extensions. A trivial approach to achieve at least some degree of compatibility is to eliminate user-defined operators by unfolding definitions and to specify user-defined types by adding appropriate axioms. But the results of Section 5.3.1.4 show that unfolding definitions often leads to a poor performance. It therefore seems beneficial to investigate alternative approaches that lead to a better performance.

Before this research has been carried out, there was not much that could be done to influence the behavior of Rodin's automated tactics. The `axe` method is easier to adapt than other automated tactics of Rodin, but adapting `axe` still requires experience with Isabelle/HOL and the embedding of Event-B's logic into HOL; it is therefore unlikely that common Rodin users will be able to configure `axe`. A natural next step is to investigate more user-friendly mechanisms for influencing the behavior of automated tactics. This could include annotations for indicating which hypotheses are relevant for proofs, which disjunctions should be used for case splits, which witnesses should be considered for quantifier instantiation, and which hypotheses should be used as inference or rewrite rules. One challenge is to keep the effort of writing annotations low; another challenge is the automatic conversion of hypotheses to rules, which is more challenging than in higher-order logic because of Event-B's partial function semantics.

In the current setting, the new tactic based on Isabelle/HOL outputs only whether the given proof obligation could be proved. If a proof cannot be found, `axe` outputs the subgoals that remain to be proved; this output could help users to understand why a proof has not been found. It would therefore be useful to translate the output of `axe` back to Event-B's logic and display it in Rodin. As for unlifting, the challenge is to define a translation to Event-B's logic that produces a concise and readable

output.

My integration uses Isabelle/HOL as an automated theorem prover, but Isabelle has more to offer. For example, it supports a readable and robust notation for proofs [Wenzel, 2001] and provides a sophisticated user interface [Wenzel, 2010] that goes far beyond the common read-eval-print loop of terminals. If Isabelle/HOL were integrated as an interactive theorem prover, Rodin users would benefit from these and other features. In a possible approach to such an integration, the user specifies models in Rodin, Rodin generates proof obligations, translates them to HOL, and invokes the unlifting method. The user subsequently uses Isabelle as an interactive prover to discharge the remaining subgoals. A minor difficulty with this approach is that Event-B and HOL have a different syntax; this can be overcome by configuring Isabelle's parser and pretty-printer. Another difficulty is that corresponding functions (such as Event-B's and HOL's division) have slightly different semantics; I expect that this difficulty can be overcome by appropriate documentation, because Rodin does already implement different semantics during modeling and proving (cf. Section 4.3) and users (as opposed to developers) hardly notice the difference.

It is also natural to invest into building more powerful and generic proof tactics for Rodin; some progress is reported in [Butler and Maamria, 2010, Maamria and Butler, 2010]. On the one hand, this approach is not burdened with the engineering challenges of a tool integration. It is also a great chance to develop novel proof tactics for reasoning about partial functions, such as directed rewriting. On the other hand, apart form soundness concerns, non-trivial techniques like term rewriting, higher-order unification [Huet, 1975], and tableau reasoning need to be implemented from scratch, which is a major undertaking. So this approach has the best chances to lead to an optimal solution, but the effort may be unreasonably high.

6.3. Personal Remarks

I conclude my thesis with three personal remarks. First, it became clear in a very early stage that any serious work on theorem proving in Event-B would require a major overhaul of the foundations of its logic. Initially, I was not too happy about it, because I viewed this task as tedious routine work that would not reveal valuable insights. But then, it turned out to be surprisingly challenging to get all the details right and to make the presentation readable; that was already an exciting task per se. Even better, inspired by discussions (most notably with Issam Maamria) and by carefully examining Rodin's proof calculus, I discovered that Rodin implemented a smart, powerful, and hitherto poorly understood rewriting technique: directed rewriting.

Second, my attitude towards partial functions has changed over the years. Initially, I had the impression that logics of partial functions were unreasonably complicated and therefore a pointless academic exercise. I still view logics of partial functions as more complicated than classical logics. However, this complexity stems from partial functions themselves and not from their representation in logics of partial functions.

Reasoning about total functions in a logic of partial functions does not have to be more difficult than in a classical logic, because the corresponding well-definedness conditions can be solved by a trivial tactic (i.e., bottom-up unlifting). Likewise, reasoning about partial functions in a classical logic is not simpler than in a logic of partial functions: this conviction is the result of doing numerous painful proofs about definite descriptions (the HOL counterparts of functional images) with Isabelle/HOL. I even had the impression that these proofs would have been less painful in Rodin with appropriate directed rewrite rules. In practice, it may still be worth expressing partial functions in a classical logic, because classical logics often have a better tool support. For the time being, my advice is this: if partial functions can be avoided, avoid them and use a classical logic; if they are inherent to the problem, try to make the decision by the quality of tool support; if tools for classical logics do not bring clear benefits, choose a tool supporting a logic of partial functions.

Third, one of the most critical decision during this research was whether to develop my own theorem prover and, if not, which theorem prover to build on. The advice I received from other people could not have been more contradictory; this is not a complaint – it was a difficult decision. After frustrating experiences with other theorem provers, I decided to build on Isabelle/HOL. Of course, it was difficult to become acquainted with the tool. But then its capabilities far surpassed my expectations. I progressed very quickly mainly for two reasons: Isabelle's automated tactics can be configured on a high level of abstraction, which relieved me to a large extent of implementing low-level term manipulations. And thanks to the LCF architecture, I never had to bother about soundness, which decreased the effort of testing significantly. Without these two capabilities of someone else's theorem prover I would have achieved much less. So with hindsight, Paulson's advice not to write a theorem prover was good for me.

List of Figures

2.1. Naming conventions 29
2.2. Unsound inference rules for the various sequent semantics 41

3.1. Event-B's theories 56
3.2. Definitional extensions 68

4.1. Dependencies between features of Event-B's logic 95

5.1. A trivial unlifting algorithm 110
5.2. Additional rules of the efficient unlifting algorithm 112
5.3. Performance of unlifting in the Abrial benchmark 123
5.4. Performance of unlifting in the BepiColombo benchmark 124
5.5. Performance of unlifting in the Rules benchmark 125
5.6. Comparison between different methods with simplification depth limit 1 and Version 3 of the clasimpset on the Abrial benchmark 127
5.7. Comparison between different simplification depth limits with `auto` and Version 3 of the clasimpset on the Abrial benchmark 128
5.8. Comparison between different clasimpsets with `auto` and simplification depth limit 1 on the Abrial benchmark 129
5.9. Comparison between different simplification depth limits with `auto` and Version 3 of the clasimpset on the BepiColombo benchmark . . . 131
5.10. Comparison between different methods with simplification depth limit 0 and Version 3 of the clasimpset on the BepiColombo benchmark . 132
5.11. Comparison between different clasimpsets with `auto` and simplification depth limit 0 on the BepiColombo benchmark 133
5.12. Comparison of `axe` with `auto` on the Abrial benchmark 137
5.13. Comparison of `axe` with `auto` on the BepiColombo benchmark . . . 138
5.14. Variants of `axe` on the Abrial benchmark 140
5.15. Variants of `axe` on the BepiColombo benchmark 141
5.16. Different splitting strategies on the Abrial benchmark 142
5.17. Different splitting strategies on the BepiColombo benchmark 143
5.18. Comparison of `axe` with Rodin's auto-tactic on the Abrial benchmark 144
5.19. Evolution of `axe` and the clasimpset on the training set 147
5.20. Evolution of `axe` and the clasimpset on the validation set 148
5.21. Performance of `axe'` and Version 7 of the clasimpset on the full BepiColombo benchmark 152

5.22. Performance of `axe'` and Version 7 of the clasimpset on the Abrial benchmark 153
5.23. Comparison of `axe'` with Rodin's auto-tactic on the BepiColombo benchmark 154

Bibliography

J.-R. Abrial. Models accompanying Abrial [2010]. `http://wiki.event-b.org/index.php/Event-B_Language`.

J.-R. Abrial. *The B-book – assigning programs to meanings*. Cambridge University Press, 2005.

J.-R. Abrial. *Modeling in Event-B*. Cambridge University Press, 2010.

J.-R. Abrial and L. Mussat. On using conditional definitions in formal theories. In *ZB*, volume 2272 of *Lecture Notes in Computer Science*, pages 242–269. Springer, 2002.

J.-R. Abrial, M. J. Butler, S. Hallerstede, T. S. Hoang, F. Mehta, and L. Voisin. Rodin: an open toolset for modelling and reasoning in Event-B. *STTT*, 12(6): 447–466, 2010.

P. B. Andrews. *An Introduction to Mathematical Logic and Type Theory*. Springer, 2002.

R. D. Arthan. Undefinedness in Z: Issues for specification and proof. In *Proceedings of the Workshop on the Mechanization of Partial Functions*. 1996.

F. Badeau and A. Amelot. Using B as a high level programming language in an industrial project: Roissy VAL. In *ZB*, volume 3455 of *Lecture Notes in Computer Science*, pages 334–354. Springer, 2005.

C. Baier and J.-P. Katoen. *Principles of Model Checking*. MIT Press, 2008.

M. Barnett, M. Fähndrich, K. R. M. Leino, P. Müller, W. Schulte, and H. Venter. Specification and verification: the Spec# experience. *Communications of the ACM*, 54(6):81–91, 2011.

H. Barringer, J. H. Cheng, and C. B. Jones. A logic covering undefinedness in program proofs. *Acta Informatica*, 21:251–269, 1984.

D. Basin and S. Matthews. Logical frameworks. In *Handbook of Philosophical Logic*, volume 9, pages 89–163. Kluwer Academic Publishers, 2002.

M. J. Beeson. *Foundations of Constructive Mathematics*. Springer, 1985.

P. Behm, P. Benoit, A. Faivre, and J.-M. Meynadier. Météor: A successful application of B in a large project. In *World Congress on Formal Methods*, volume 1708 of *Lecture Notes in Computer Science*, pages 369–387. Springer, 1999.

S. Berezin, C. Barrett, I. Shikanian, M. Chechik, A. Gurfinkel, and D. L. Dill. A practical approach to partial functions in CVC Lite. *Electronic Notes in Theoretical Computer Science*, 125(3):13–23, 2005.

S. Berghofer, T. Nipkow, C. Urban, and M. Wenzel, editors. *22nd International Conference on Theorem Proving in Higher Order Logics*, volume 5674 of *Lecture Notes in Computer Science*, 2009. Springer.

S. Böhme and T. Weber. Fast LCF-style proof reconstruction for Z3. In *ITP*, volume 6172 of *Lecture Notes in Computer Science*, pages 179–194. Springer, 2010.

S. Böhme, M. Moskal, W. Schulte, and B. Wolff. HOL-Boogie - an interactive prover-backend for the verifying C compiler. *Journal of Automated Reasoning*, 44(1-2): 111–144, 2010.

A. D. Brucker, F. Rittinger, and B. Wolff. HOL-Z 2.0: A proof environment for Z-specifications. *Journal of Universal Computer Science*, 9(2):152–172, 2003.

T. Burge. Truth and singular terms. *Noûs*, 8(4):309–325, 1974.

M. Butler and I. Maamria. Mathematical extension in Event-B through the Rodin theory component, 2010. `http://deploy-eprints.ecs.soton.ac.uk/251`.

CADiZ/Ref. CADiZ reference manual. `http://www.cs.york.ac.uk/hise/cadiz/looseness.html`.

J. H. Cheng and C. B. Jones. On the usability of logics which handle partial functions. In *3rd BCS-FACS Refinement Workshop*, pages 51–69. Springer, 1990.

E. M. Clark, E. M. J. Clark, and O. Grumberg. *Model Checking*. MIT Press, 2000.

ClearSy. Atelier B. `http://www.atelierb.eu`.

D. Cohen and P. Watson. An efficient representation of arithmetic for term rewriting. In *RTA*, volume 488 of *Lecture Notes in Computer Science*, pages 240–251. Springer, 1991.

E. Cohen, M. Dahlweid, M. A. Hillebrand, D. Leinenbach, M. Moskal, T. Santen, W. Schulte, and S. Tobies. VCC: A practical system for verifying concurrent C. In Berghofer et al. [2009], pages 23–42.

D. R. Cok and J. Kiniry. ESC/Java2: Uniting ESC/Java and JML. In *CASSIS*, volume 3362 of *Lecture Notes in Computer Science*, pages 108–128. Springer, 2004.

Á. Darvas, F. Mehta, and A. Rudich. Efficient well-definedness checking. In *IJCAR*, volume 5195 of *Lecture Notes in Computer Science*, pages 100–115. Springer, 2008.

J. E. Dawson. Simulating term-rewriting in LPF and in display logic. In *Supplementary Proceedings of TPHOLs*, pages 47–62. Australian National University, 1998.

L. M. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *TACAS*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008.

E. W. Dijkstra. Guarded commands, non-determinancy and a calculus for the derivation of programs. In *Language Hierarchies and Interfaces*, volume 46 of *Lecture Notes in Computer Science*, pages 111–124. Springer, 1975.

E. W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.

W. M. Farmer. A partial functions version of Church's simple theory of types. *Journal of Symbolic Logic*, 55(3):1269–1291, 1990.

W. M. Farmer. A simple type theory with partial functions and subtypes. *Annals of Pure and Applied Logic*, 64(3):211–240, 1993.

W. M. Farmer, J. D. Guttman, and F. J. Thayer. IMPS: An interactive mathematical proof system. *Journal of Automated Reasoning*, 11(2):213–248, 1993.

W. M. Farmer, J. D. Guttman, and F. J. Thayer. IMPS: An updated system description. In *CADE*, volume 1104 of *Lecture Notes in Computer Science*, pages 298–302. Springer, 1996.

S. Feferman. Polymorphic typed lambda-calculi in a type-free axiomatic framework. *Contemporary Mathematics*, 106:101–136, 1990.

C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for Java. In *PLDI*, pages 234–245, 2002.

G. Gentzen. Untersuchungen über das logische Schließen. I. *Mathematische Zeitschrift*, 39(1):176–210, 1935.

R. Gmehlich, K. Grau, S. Hallerstede, M. Leuschel, F. Lösch, and D. Plagge. On fitting a formal method into practice. In Qin and Qiu [2011], pages 195–210.

M. J. C. Gordon and T. F. Melham. *Introduction to HOL*. Cambridge University Press, 1993.

M. J. C. Gordon and A. M. Pitts. *The HOL Logic*, pages 191–232. In Gordon and Melham [1993], 1993.

M. J. C. Gordon, R. Milner, and C. P. Wadsworth. *Edinburgh LCF*, volume 78 of *Lecture Notes in Computer Science*. Springer, 1979.

D. Gries and F. B. Schneider. Avoiding the undefined by underspecification. In *Computer Science Today*, volume 1000 of *Lecture Notes in Computer Science*, pages 366–373. Springer, 1995.

S. Hallerstede and M. Leuschel. Constraint-based deadlock checking of high-level specifications. *TPLP*, 11(4-5):767–782, 2011.

J. Harrison. HOL Light: An overview. In Berghofer et al. [2009], pages 60–66.

J. Harrison. *Handbook of Practical Logic and Automated Reasoning.* Cambridge University Press, 2009b.

M. Hinchey, J. P. Bowen, and E. Vassev. Formal methods. In *Encyclopedia of Software Engineering*, pages 308–320. Taylor & Francis, 2010.

J. R. Hindley and J. P. Seldin. *Lambda-Calculus and Combinators.* Cambridge University Press, 2008.

T. S. Hoang and J.-R. Abrial. Reasoning about liveness properties in Event-B. In Qin and Qiu [2011], pages 456–471.

C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

K. Hoder and A. Voronkov. Sine qua non for large theory reasoning. In *CADE*, volume 6803 of *Lecture Notes in Computer Science*, pages 299–314. Springer, 2011.

G. P. Huet. Unification in typed lambda calculus. In *Lambda-Calculus and Computer Science Theory*, volume 37 of *Lecture Notes in Computer Science*, pages 192–212. Springer, 1975.

D. Ilić, T. Latvala, P. Väisänen, K. Varpaaniemi, L. Laibinis, and E. Troubitsyna. Deploy deliverable D20 – pilot deployment in the space sector. `http://www.deploy-project.eu/pdf/D20-pilot-deployment-in-the-space-sector-final-version.pdf`.

C. B. Jones. Partial functions and logics: A warning. *Information Processing Letters*, 54(2):65–67, 1995.

C. B. Jones. Reasoning about partial functions in the formal development of programs. *Electronic Notes in Theoretical Computer Science*, 145:3–25, 2006.

C. B. Jones and C. A. Middelburg. A typed logic of partial functions reconstructed classically. *Acta Informatica*, 31(5):399–430, 1994.

C. B. Jones, K. D. Jones, P. A. Lindsay, and R. C. Moore. *Mural - a formal development support system.* Springer, 1991.

M. Kaufmann, P. Manolios, and J. S. Moore. *Computer-Aided Reasoning: An Approach.* Kluwer Academic Publishers, 2000.

S. C. Kleene. *Introduction to Metamathematics.* North Holland, 1952.

G. Klein, J. Andronick, K. Elphinstone, G. Heiser, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. seL4: formal verification of an operating-system kernel. *Communications of the ACM*, 53(6):107–115, 2010.

Kolyang, T. Santen, and B. Wolff. A structure preserving encoding of Z in Isabelle/HOL. In *TPHOLs*, volume 1125 of *Lecture Notes in Computer Science*, pages 283–298. Springer, 1996.

M. Leuschel, J. Falampin, F. Fritz, and D. Plagge. Automated property verification for large scale B models with ProB. *Formal Aspects of Computing*, 23(6):683–709, 2011.

I. Maamria and M. Butler. Rewriting and well-definedness within a proof system. In *PAR*, volume 43 of *Electronic Proceedings in Theoretical Computer Science*, pages 49–64, 2010.

J. McCarthy. A basis for a mathematical theory of computation. *Computer Programming and Formal Systems*, pages 33–70, 1963.

F. Mehta. Supporting proof in a reactive development environment. In *SEFM*, pages 103–112. IEEE Computer Society, 2007.

F. Mehta. A practical approach to partiality - a proof based approach. In *ICFEM*, volume 5256 of *Lecture Notes in Computer Science*, pages 238–257. Springer, 2008.

E. Mendelson. *Number Systems and the Foundations of Analysis*. Academic Press, 1973.

J. Meng and L. C. Paulson. Lightweight relevance filtering for machine-generated resolution problems. *Journal of Applied Logic*, 7(1):41–57, 2009.

J. Meng, C. Quigley, and L. C. Paulson. Automation for interactive proof: First prototype. *Information and Computation*, 204(10):1575–1596, 2006.

C. Metayer and L. Voisin. The Event-B mathematical language, 2009. `http://deploy-eprints.ecs.soton.ac.uk/11`.

D. Miller. Unification under a mixed prefix. *Journal of Symbolic Computation*, 14 (4):321–358, 1992.

O. Müller and K. Slind. Treating partiality in a logic of total functions. *Computer Journal*, 40(10):640–652, 1997.

O. Müller, T. Nipkow, D. von Oheimb, and O. Slotosch. HOLCF=HOL+LCF. *Journal of Functional Programming*, 9(2):191–223, 1999.

T. Nipkow. Term rewriting and beyond - theorem proving in isabelle. *Formal Aspects of Computing*, 1(4):320–338, 1989.

T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer, 2002.

M. Ouimet and K. Lundqvist. The TASM toolset: Specification, simulation, and formal verification of real-time systems. In *CAV*, volume 4590 of *Lecture Notes in Computer Science*, pages 126–130. Springer, 2007.

O. Owe. Partial logics reconsidered: A conservative approach. *Formal Aspects of Computing*, 5(3):208–223, 1993.

S. Owre and N. Shankar. The formal semantics of PVS, 1999. `http://pvs.csl.sri.com/papers/csl-97-2/csl-97-2.ps`.

S. Owre, J. M. Rushby, and N. Shankar. PVS: A prototype verification system. In *CADE*, volume 607 of *Lecture Notes in Computer Science*, pages 748–752. Springer, 1992.

L. C. Paulson. *Logic and Computation: Interactive Proof with Cambridge LCF*. Cambridge University Press, 1987.

L. C. Paulson. The foundation of a generic theorem prover. *Journal of Automated Reasoning*, 5(3):363–397, 1989.

L. C. Paulson. Designing a theorem prover. In *Handbook of Logic in Computer Science*, volume II, pages 415–475. Oxford, 1992.

L. C. Paulson. Old Isabelle reference manual, 2011. `http://isabelle.in.tum.de/dist/Isabelle2011-1/doc/ref.pdf`.

D. Prawitz. *Natural deduction: a proof-theoretical study*. Dover Publications, 2006.

S. Qin and Z. Qiu, editors. *13th International Conference on Formal Engineering Methods*, volume 6991 of *Lecture Notes in Computer Science*, 2011. Springer.

A. Riazanov and A. Voronkov. The design and implementation of Vampire. *AI Communications*, 15(2-3):91–110, 2002.

J. Röder. Relevance filters for Event-B. Master Thesis, ETH Zürich, 2010.

J. M. Rushby, S. Owre, and N. Shankar. Subtypes for specifications: Predicate subtyping in PVS. *IEEE Transaction on Software Engineering*, 24(9):709–720, 1998.

B. Russell. On denoting. *Mind*, 14(56):479–493, 1905.

B. Russell. *Introduction to Mathematical Philosophy*. Allen & Unwin, reprinted edition, 1948.

M. Saaltink. The Z/EVES system. In *ZUM*, volume 1212 of *Lecture Notes in Computer Science*, pages 72–85. Springer, 1997.

M. Schmalz. HOL/Event-B, Version 0.5.0. `http://sourceforge.net/projects/rodin-b-sharp/files/Plugin_Isabelle`.

M. Schmalz. Term rewriting in logics of partial functions. In Qin and Qiu [2011], pages 633–650.

M. Schmalz. Rodin's soundness bugs. Technical Report 697, ETH Zurich, Switzerland, 2012. `http://www.inf.ethz.ch/research/disstechreps/techreports`.

R. Schock. *Logics without existence assumptions*. Almqvist & Wiksell, 1968.

S. Schulz. E - a brainiac theorem prover. *AI Communications*, 15(2-3):111–126, 2002.

N. Shankar and S. Owre. Principles and pragmatics of subtyping in PVS. In *WADT*, volume 1827 of *Lecture Notes in Computer Science*, pages 37–52. Springer, 1999.

N. Shankar, S. Owre, J. M. Rushby, and D. W. J. Stringer-Calvert. PVS prover guide, 2001. `http://pvs.csl.sri.com/doc/pvs-prover-guide.pdf`.

J. R. Shoenfield. *Mathematical Logic*. Addison Wesley, 1967.

K. Slind. Trusted extensions of interactive theorem provers: Workshop summary. `http://www.cs.utexas.edu/~kaufmann/itp-trusted-extensions-aug-2010/summary/summary.pdf`, 2010.

C. F. Snook and M. J. Butler. UML-B: A plug-in for the Event-B tool set. In *ABZ*, volume 5238 of *Lecture Notes in Computer Science*, page 344. Springer, 2008.

J. M. Spivey. *Understanding Z*. Cambridge University Press, 1988.

J. M. Spivey. *The Z Notation - a reference manual*. Prentice Hall International Series in Computer Science. Prentice Hall, 2nd edition, 1992.

C. Sprenger and D. A. Basin. Developing security protocols by refinement. In *ACM Conference on Computer and Communications Security*, pages 361–374. ACM, 2010.

telcoWD. Telephone conference of the Deploy project on well-definedness, December 2009. Participants: M. J. Butler, S. Hallerstede, C. B. Jones, M. Schmalz, and L. Voisin.

I. Toyn and J. A. McDermid. CADiZ: An architecture for Z tools and its implementation. *Software: Practice and Experience*, 25(3):305–330, 1995.

S. H. Valentine. Inconsistency and undefinedness in Z - a practical guide. In *ZUM*, volume 1493 of *Lecture Notes in Computer Science*, pages 233–249. Springer, 1998.

D. van Dalen. *Logic and Structure*. Springer, fourth edition, 2004.

K. Varpaaniemi. Bepicolombo models v6.4, 2010. `http://deploy-eprints.ecs.soton.ac.uk/244`.

P. A. S. Veloso and S. R. M. Veloso. On conservative and expansive extensions. O que nos faz pensar, 4, 1991. `http://www.oquenosfazpensar.com/adm/uploads/artigo/on_conservative_and_expansive_extensions/n4paulo.pdf`.

C. Weidenbach, D. Dimova, A. Fietzke, R. Kumar, M. Suda, and P. Wischnewski. Spass version 3.5. In *CADE*, volume 5663 of *Lecture Notes in Computer Science*, pages 140–145. Springer, 2009.

M. Wenzel. Type classes and overloading in higher-order logic. In *TPHOLs*, volume 1275 of *Lecture Notes in Computer Science*, pages 307–322. Springer, 1997.

M. Wenzel. Isabelle/Isar – a versatile environment for human-readable formal proof documents. Dissertation, Technische Universität München, 2001.

M. Wenzel. Asynchronous proof processing with Isabelle/Scala and Isabelle/jEdit. In *User Interfaces for Theorem Provers*, Electronic Notes in Theoretical Computer Science. Elsevier, 2010.

M. Wenzel. The Isabelle/Isar reference manual, 2011. `http://isabelle.in.tum.de/dist/Isabelle2011/doc/isar-ref.pdf`.

J. Woodcock, M. Saaltink, and L. Freitas. Unifying theories of undefinedness. In *Engineering Methods and Tools for Software Safety and Security*, pages 311–330. IOS Press, 2009.

E. Yilmaz and T. S. Hoang. Development of Rabin's choice coordination algorithm in Event-B. *ECEASST*, 35, 2010.

A. Statistics on Rules Implemented in Rodin

Rodin's inference and rewrite rules can be found in the following locations:

`http://wiki.event-b.org/index.php?title=Inference_Rules&oldid=8898`
`http://wiki.event-b.org/index.php?title=Set_Rewrite_Rules&oldid=8523`
`http://wiki.event-b.org/index.php?title=Relation_Rewrite_Rules&oldid=8891`
`http://wiki.event-b.org/index.php?title=Arithmetic_Rewrite_Rules&oldid=8892`

My analysis is based on the version of April 5, 2011; I also include rules into my analysis that had been proposed for implementation but were not yet implemented.

A.1. Conditional Rewrite Rules

The documentation clearly distinguishes inference rules from unconditional rewrite rules. Conditional rewrite rules are represented either as inference rules or as rewrite rules with side-conditions. For my analysis I therefore had to change the representation of rules as in the following examples:

fin_l_lower_bound_l: $$\frac{\mathsf{finite}(\$S)}{\exists n \cdot \forall x \cdot x \in \$S \Rightarrow n \leq x \ \sqsubseteq \ \top}$$

ov_setenum_l: $$\frac{\$G = \$E}{(\$f \lhd\!\!+ \{\$E \mapsto \$F\})(\$G) \ \sqsubseteq \ \$F}$$

and $$\frac{\neg(\$G = \$E)}{(\$f \lhd\!\!+ \{\$E \mapsto \$F\})(\$G) \ \sqsubseteq \ \{\$E\} \lhd \$f(\$G)}$$

fin_subseteq_r: $$\frac{\mathsf{D}(\$T) \land \$S \subseteq \$T \land \mathsf{finite}(\$T)}{\mathsf{finite}(\$S) \ \sqsubseteq \ \top}$$

simp_card_setminus_l: $$\frac{\mathsf{finite}(\$S)}{\mathsf{card}(\$S \setminus \$T) \ \sqsubseteq \ \mathsf{card}(\$S) - \mathsf{card}(\$S \cap \$T)}$$

Below is the list of conditional rewrite rules.

fun_goal, neg_in_l, subset_inter, in_inter, notin_inter, fin_l_lower_bound_l, fin_l_lower_bound_r, fin_l_upper_bound_l, fin_l_upper_bound_r, ov_setenum_l (2 rules), ov_l (2 rules), dis_binter_r, dis_setminus_r,

sim_rel_image_r, sim_fcomp_r, fin_subseteq_r, fin_binter_r, fin_kinter_r, fin_qinter_r, fin_setminus_r, fin_rel, fin_rel_img_r, fin_rel_ran_r, fin_rel_dom_r, fin_fun_dom, fin_run_ran, fin_fun_img_r, fin_fun_ran_r, fin_fun_dom_r, fin_lt_0, fin_ge_0, card_interv, card_empty_interv, deriv_le_card, deriv_ge_card, deriv_lt_card, deriv_gt_card, deriv_equal_card, simp_card_setminus_l, simp_card_cprod_l, one_point_l (with $\forall$), one_point_l (with $\exists$), simp_funimage_domres, simp_funimage_domsub, simp_funimage_ranres, simp_funimage_ransub, simp_funimage_setminus, deriv_dom_totalrel, deriv_ran_surjrel, simp_card_setminus, simp_card_setminus_setenum

A.2. Truly Directed Rewrite Rules in Rodin

Rodin's proof rules are organized in four categories: inference rules, set rewrite rules, relation rewrite rules, and arithmetic rewrite rules. Table A.1 summarizes my observations on unconditional rewrite rules. The column TDCTF counts rules that are truly directed and conditional if restated in a logic of total functions, assuming that this logic of total functions approximates partial functions as underspecified total functions.

To avoid biased results due to uncommon domains of operators, I assume for the TDCTF column that inter (and $\bigcap$) is definite, and mod and $\widehat{\ }$ are well-defined if their first-arguments are negative.

In fact, there are rules that are unsound in logics of total functions (with underspecification), but sound as *symmetric* rewrite rule in Event-B:

$$(2 * \$x) \div (2 * \$y) \equiv \$x \div \$y.$$

Its classical counterpart is unsound, because $2 \div 0$ and $1 \div 0$ may denote different numbers. This rule shows that some rewrite rules have less conditions if there is only one ill-defined value. The TDCTF column does not count such rules; it counts only truly directed rules because it is supposed to measure the positive impact of directed rewriting, not the positive impact of logics of partial functions in general.

In the following, I list the unconditional truly directed rewrite rules. The notation "rule_name (+x)" designates the rule with name "rule_name" and the x following rules.

Category Set Rewrite Rules.

simp_multi_and_not, simp_multi_or_not, simp_multi_imp (+1), simp_multi_eqv (+1), simp_multi_equal (+2), simp_type_subseteq, simp_special_subseteq (+7), simp_special_binter, simp_special_bunion, simp_multi_setminus, simp_special_setminus_l (+1), simp_kinter_pow, simp_special_cprod_r (+2), simp_subseteq_compset_l (+4), distri_subseteq_bunion_sing, simp_finite_setenum (+1), simp_finite_qunion,

Table A.1.: Statistics on unconditional rewrite rules

Category	Total	Truly Directed	TDCTF
Inference Rules	0	0	0
Set Rewrite Rules	166	59	0
Relation Rewrite Rules	198	65	12
Arithmetic Rewrite Rules	89	41	23
Total	453	165	35

deriv_finite_cprod, simp_finite_upto, simp_finite_lambda, simp_type_in, simp_special_subset_r, simp_multi_subset (+3), def_in_mapsto, def_in_pow1 (+10), deriv_subseteq_setminus_r (+1).

Category Relation Rewrite Rules.

simp_dom_setenum, simp_ran_setenum, simp_type_overl_cprod (+2), simp_special_ranres_r (+1), simp_special_domsub_r (+3), simp_special_ransub_l (+3), simp_special_fcomp, simp_special_bcomp, simp_special_dprod_r (+1), simp_special_pprod_r (+1), simp_special_relimage_r (+1), simp_multi_relimage_cprod_sing (+2), simp_multi_relimage_domsub, simp_special_rel_r (+5), simp_funimage_lambda (+10), simp_funimage_funimage_converse (+2), deriv_fcomp_sing, def_in_domres (+5), def_in_dprod (+1), def_in_reldom (+4), def_in_tinj (+3).

Category Arithmetic Rewrite Rules.

simp_special_mod_0 (+1), simp_min_bunion_sing (+3), simp_card_sing (+1), simp_card_lambda, simp_lit_ge_card_1 (+8), def_equal_min (+1), simp_multi_minus (+10), simp_special_div_0, simp_special_expn_1_l (+8).

In the following, I list the rules I counted in the TDCTF column.

Category Relation Rewrite Rules.

simp_funimage_lambda, simp_in_funimage (+7), simp_funimage_funimage_converse (+2).

Table A.2.: Statistics on conditional rewrite rules

Category	Total	Truly Directed
Inference Rules	44	35
Set Rewrite Rules	0	0
Relation Rewrite Rules	7	7
Arithmetic Rewrite Rules	2	0
Total	53	42

Category Arithmetic Rewrite Rules.

simp_special_mod_0, simp_min_bunion_sing (+3), simp_special_equal_card, simp_card_lambda, simp_lit_ge_card_1 (+8), def_equal_min (+1), simp_special_div_0, simp_special_expn_1_l, simp_multi_div (+2).

Table A.2 summarizes my observations on conditional rewrite rules. The inference rule category contains conditional rewrite rules that are represented as inference rules. In the following, I list the truly directed conditional rewrite rules.

Category Inference Rules.

fun_goal, neg_in_l, subset_inter (+6), ov_setenum_l (2 rules), ov_l (2 rules), fin_subseteq_r (+10), fin_funimage_r (+2), fin_lt_0 (+1), card_empty_interv (+5).

Category Relation Rewrite Rules.

simp_funimage_domres (+4), deriv_dom_totalrel (+1).

A.3. Conditional Rewrite Rules with Dispensable Conditions

For the following rules, the strictness side-condition can be dropped:

dis_binter_r, dis_binter_l, dis_setminus_r, dis_setminus_l, simp_card_setminus_l, simp_card_setminus_r, simp_card_cprod_l, simp_card_cprod_r.

In the case of ov_setenum_l, the strictness side-condition is necessary for soundness, but can be avoided if the rule is stated as proposed in Section A.1. Similar considerations apply to

ov_setenum_r, ov_l, ov_r, card_interv, and card_empty_interv.

B. Complexity of Unlifting

theory *Unlifting* **imports** *Main* **begin**

Generated with Isabelle 2011.

B.1. Definitions

The type *tt* is a datatype for (a subset of) the terms over the theory EventB. Types and binding structure is ignored. Several operators / binders are identified because the equalities used for unlifting have a similar structure. For simplicity, operators that take more than two arguments (i.e., cond, enumerated sets, and partition) or bind more than one variable (i.e., some set comprehensions) are ignored. The main result holds for such operators as well.

datatype *tt* =
  var — ordinary variable or constant

| *coll tt* — smashed and definite, one argument, such as collect and dom
| *eq tt tt* — strict and definite, two arguments, such as equality
| *ch tt* — some, min, max, inter, card
| *md tt tt* — set comprehensions that bind one variable (including $\bigcup$ and $\bigcap$), funimg, mod, div, and exponentiation

| *land tt tt* — conjunction, disjunction, implication
| *all tt* — universal and existential quantifiers

The function *size* yields a lower bound on the size of the denotation of an Event-B term. The lower bound differs from the actual size by a constant factor.

hide_const (**open**) *size*

fun *size* :: "*tt* ⇒ *int*"
where
  "*size var* = *1*"
| "*size* (*coll t*) = *1* + *size t*"
| "*size* (*eq t1 t2*) = *1* + *size t1* + *size t2*"
| "*size* (*ch t*) = *1* + *size t*"
| "*size* (*md t1 t2*) = *1* + *size t1* + *size t2*"
| "*size* (*land t1 t2*) = *1* + *size t1* + *size t2*"
| "*size* (*all t*) = *1* + *size t*"

lemma *size_pos* [*simp*]: "*0* < *size t*" "*1* ≤ *size t*" "*0* ≤ *size t*"
  by (*induct t*) *auto*

The *constant overhead* of an unlifting rule is the number of symbols in the right-hand side "between" the terms that are unlifted further. E.g., in the case of `WD (⟦mod⟧ x y) = WD x ∧ WD y ∧ x↓ ≥ 0 ∧ y↓ ≥ 0` the constant overhead is 7 (3 occurrences of `op ∧`, 2 occurrences of `op ≥`, 2 occurrences of `0`). The constant `zc` is an upper bound on the constant overhead of all unlifting rules.

definition `zc :: "int"`
where `zc_pos [simp, arith]: "zc = 20"` — underspecification would be nice, but it complicates proofs too much.

The *strip overhead* of an unlifting rule is the maximum number of occurrences of a term `x↓` in the right-hand side, where `x` ranges over variables in the left-hand side. E.g., in the case of `WD (⟦mod⟧ x y) = WD x ∧ WD y ∧ x↓ ≥ 0 ∧ y↓ ≥ 0` the strip overhead is 1 because `x↓` and `y↓` occur only once in the right-hand side. The constant `zs` is an upper bound on the strip overhead of all unlfiting rules.

definition `zs :: "int"`
where `zs_pos [simp, arith]: "zs = 3"`
— underspecification would be nice, but it complicates proofs too much.

Upper bounds on the sizes of unlifted terms:

fun `wds :: "tt ⇒ int"` — `wds t` $\geq |\mathcal{U}(\mathsf{WD}\ [\![t]\!])|$
and `ss :: "tt ⇒ int"` — `ss t` $\geq |\mathcal{U}([\![t]\!]\downarrow)|$
and `ts :: "tt ⇒ int"` — `ts t` $\geq |\mathcal{U}(\mathsf{T}\ [\![t]\!])|$
There is no need to consider WT and F because they behave like T. There is also no need to consider $=$ and $\sqsubseteq$, as they are unfolded only once.
where

```
  "wds var = zc"
| "wds (coll t) = zc + wds t"
| "wds (eq t1 t2) = zc + wds t1 + wds t2"
| "wds (ch t) = zc + wds t + zs * ss t"
| "wds (md t1 t2) = zc + wds t1 + wds t2 + zs * ss t1 + zs * ss t2"
| "wds (land t1 t2) = zc + 2 * (ts t1 + ts t2)"
| "wds (all t) = zc + 2 * ts t"

| "ss var = zc"
| "ss (coll t) = zc + ss t"
| "ss (eq t1 t2) = zc + ss t1 + ss t2"
| "ss (ch t) = zc + ss t"
| "ss (md t1 t2) = zc + ss t1 + ss t2"
| "ss (land t1 t2) = zc + ss t1 + ss t2"
| "ss (all t) = zc + ss t"

| "ts (land t1 t2) = zc + ts t1 + ts t2"
| "ts (all t) = zc + ts t"
| "ts var = zc"
| "ts t = wds t + ss t"
```

The functions `nt` and `nwd` count the nested applications of "expensive" unlifting rules like `WD (x ^∧^ y) = ...` and `WD (∀^ x. phi x) = ...`.

- *nwd*: a term of the form WD t is unlifted.
- *nt*: a term of the form T t is unlifted.

```
fun nt :: "tt ⇒ nat"
and nwd :: "tt ⇒ nat"
where
  "nwd var = 0"
| "nwd (coll t) = nwd t"
| "nwd (eq t1 t2) = max (nwd t1) (nwd t2)"
| "nwd (ch t) = nwd t"
| "nwd (md t1 t2) = max (nwd t1) (nwd t2)"
| "nwd (land t1 t2) = 1 + max (nt t1) (nt t2)"
| "nwd (all t) = 1 + nt t"

| "nt var = 0"
| "nt (coll t) = nwd t"
| "nt (eq t1 t2) = max (nwd t1) (nwd t2)"
| "nt (ch t) = nwd t"
| "nt (md t1 t2) = max (nwd t1) (nwd t2)"
| "nt (land t1 t2) = max (nt t1) (nt t2)"
| "nt (all t) = nt t"
```

The $\{\wedge, \vee, \Rightarrow, \forall, \exists\}$-alternation depth of a term. Currently, there is no difference between *alt* and *nt* or *alt'* and *nwd*; but they could differ in future versions that take more operators or binders into account.

```
fun alt :: "tt ⇒ nat"
and alt' :: "tt ⇒ nat"
where
  "alt var = 0"
| "alt (coll t) = alt' t"
| "alt (eq t1 t2) = max (alt' t1) (alt' t2)"
| "alt (ch t) = alt' t"
| "alt (md t1 t2) = max (alt' t1) (alt' t2)"
| "alt (land t1 t2) = max (alt t1) (alt t2)"
| "alt (all t) = alt t"

| "alt' var = 0"
| "alt' (coll t) = alt' t"
| "alt' (eq t1 t2) = max (alt' t1) (alt' t2)"
| "alt' (ch t) = alt' t"
| "alt' (md t1 t2) = max (alt' t1) (alt' t2)"
| "alt' (land t1 t2) = 1 + max (alt t1) (alt t2)"
| "alt' (all t) = 1 + alt t"
```

```
lemma alt_bound: "alt t ≥ nt t" "alt' t ≥ nwd t" "alt t + 1 ≥ nwd t"
  by (induct t) auto
```

Next, I define the P-depth of a term, where P contains some, min, max, cond, inter, set comprehensions (including $\bigcup$ and $\bigcap$), funimg, mod, $\div$, exponentiation, and card.

For technical reasons, `pdepth t` equals $\mathcal{DEP}_P(t) + 1$.

```
fun pdepth :: "tt ⇒ int"
where
  "pdepth var = 1"
| "pdepth (coll t) = pdepth t"
| "pdepth (eq t1 t2) = max (pdepth t1) (pdepth t2)"
| "pdepth (ch t) = 1 + pdepth t"
| "pdepth (md t1 t2) = 1 + max (pdepth t1) (pdepth t2)"
| "pdepth (land t1 t2) = max (pdepth t1) (pdepth t2)"
| "pdepth (all t) = pdepth t"

lemma pdepth_nonneg [simp]: "pdepth t ≥ 1" "pdepth t > 0" "pdepth t ≥ 0"
  by (induct t) auto
```

B.2. Automated Tool Setup

The following declarations implement an ad-hoc tactic for solving non-linear inequalities:

- apply the distributivity laws `ring_distribs`
- sort with `add_ac`
- compare first summands of both sides
  - if one of them is negative, move it to the other side
  - if they are in the right order, apply `add_mono`
  - otherwise: skip with `le_add_drop`

Products are treated similarly.

Limitations:

- Sorting with `add_ac` does not always yield the "right" order.
- Unable to solve `a - b ≤ (0::'a)`, even if `a ≤ b`.

```
declare mult_le_cancel_left_pos [simp]
thm mult_le_cancel_left_pos

lemma mult_le_cancel_left_pos' [simp]:
"(0 :: int) < a ⟹ a ≤ a * d ⟷ 0 < d"
  apply (subst mult_1_right [symmetric])
  apply (unfold mult_le_cancel_left_pos)
  by auto

declare mult_mono [intro] mult_left_mono [intro]
thm mult_mono mult_left_mono
```

```
lemma le_mult_drop [intro]:
  "⟦(a :: int) ≤ e; 0 ≤ a; 0 < d⟧ ⟹ a ≤ d * e"
  apply (subst mult_1_left [symmetric])
  apply (rule mult_mono)
  by auto

lemma sign_simps [simp]:
"⟦(0 :: int) < a; 0 < b⟧ ⟹ 0 < a * b"
"⟦0 < a; 0 < b⟧ ⟹ 1 ≤ a * b"
proof -
  assume "0 < a" "0 < b"
  thus "0 < a * b" by (auto simp add: sign_simps)
  thus "1 ≤ a * b" by arith
qed

declare mult_nonneg_nonneg [intro]
thm mult_nonneg_nonneg

lemma le_add_drop [intro]:
"⟦(a :: int) ≤ e; 0 ≤ d⟧ ⟹ a ≤ d + e"
by auto

declare add_mono [intro]
thm add_mono

lemma le_add_flip [simp]:
"b < (0 :: int) ⟹ a ≤ b + d ⟷ a - b ≤ d"
"b < 0 ⟹ a ≤ b ⟷ a - b ≤ 0"
"a < 0 ⟹ a + b ≤ d ⟷ b ≤ d - a"
"a < 0 ⟹ a ≤ b ⟷ 0 ≤ b - a"
  by auto

lemma uminus_push [simp]:
  "- ((a :: int) * number_of x) = a * - number_of x"
  unfolding minus_mult_right
  by simp

declare ring_distribs [simp]
thm ring_distribs
declare zminus_zadd_distrib [simp]
thm zminus_zadd_distrib
declare add_ac [simp] mult_ac [simp]
thm add_ac mult_ac
declare diff_int_def_symmetric [simp del] diff_int_def [simp]
thm diff_int_def_symmetric diff_int_def
declare split_max [split]
thm split_max

lemma le_push:
```

```
"⟦(a::int) ≤ b; a ≤ b ⟹ d + b ≤ e⟧ ⟹ a + d ≤ e"
"⟦a ≤ b; a ≤ b ⟹ d + b * 2 ≤ e⟧ ⟹ a * 2 + d ≤ e"
"⟦a ≤ b; a ≤ b ⟹ d + b * 3 ≤ e⟧ ⟹ a * 3 + d ≤ e"
"⟦a ≤ b; a ≤ b ⟹ d + b * 4 ≤ e⟧ ⟹ a * 4 + d ≤ e"
"⟦a ≤ b; a ≤ b ⟹ b ≤ d⟧ ⟹ a ≤ d"
"⟦a ≤ b; a ≤ b ⟹ b * 2 ≤ d⟧ ⟹ a * 2 ≤ d"
"⟦a ≤ b; a ≤ b ⟹ b * 3 ≤ d⟧ ⟹ a * 3 ≤ d"
"⟦a ≤ b; a ≤ b ⟹ b * 4 ≤ d⟧ ⟹ a * 4 ≤ d"
by auto

lemma const_commute:
"a + zc ≤ b ⟹ zc + a ≤ b"
"a + zc * 2 ≤ b ⟹ zc * 2 + a ≤ b"
"a + zc * zs ≤ b ⟹ zc * zs + a ≤ b"
  by auto

lemma push: "(a :: int) + d ≤ b ⟹ d + a ≤ b"
  by auto
```

B.3. Main Results

```
theorem unlift_bound:
"ts t ≤
  2 ^ nt t * size t * pdepth t * zc * zs * 3 -
  size t * zc * zs ∧
ss t ≤ size t * zc ∧
wds t ≤
  2 ^ nwd t * size t * pdepth t * zc * zs * 3 -
  size t * zc * zs * 2"
proof (induct t)
  case var — base case
  thus ?case by auto
next
  case (coll t)
  thus ?case
    apply (simp (no_asm_simp))
    apply safe
    apply (simp only: add_assoc | elim le_push |
      intro const_commute | elim thin_rl)+
    apply fastsimp
    apply (simp only: add_assoc | elim le_push |
      intro const_commute | elim thin_rl)+
    apply fastsimp
    apply (simp only: add_assoc | elim le_push |
      intro const_commute | elim thin_rl)+
    apply fastsimp
    done
next
  case (all t) — lazy case
```

```
  thus ?case
    apply (simp (no_asm_simp))
    apply safe
    apply (simp only: add_assoc | elim le_push |
      intro const_commute | elim thin_rl)+
    apply fastsimp
    apply (simp only: add_assoc | elim le_push |
      intro const_commute | elim thin_rl)+
    apply fastsimp
    apply (simp only: add_assoc | elim le_push |
      intro const_commute | elim thin_rl)+
    apply fastsimp
    done
next
  case (ch t) — indefinite case
  thus ?case
    apply (simp (no_asm_simp))
    apply safe
    apply (simp only: add_assoc | elim le_push |
      intro const_commute | elim thin_rl)+
    apply fastsimp
    apply (simp only: add_assoc | elim le_push |
      intro const_commute | elim thin_rl)+
    apply fastsimp
    apply (simp only: add_assoc | elim le_push |
      intro const_commute | elim thin_rl)+
    apply fastsimp
    done
```

— The remaining cases are similar except that more terms arise because the involved symbols take two arguments.

```
next
  case (eq t1 t2) thus ?case
    apply (simp (no_asm_simp))
    apply safe

    apply (simp only: add_assoc | elim le_push | intro const_commute)+
    apply fastsimp
    apply (simp only: add_assoc | elim le_push | intro const_commute)+
    apply fastsimp
    apply (simp only: add_assoc | elim le_push | intro const_commute)+
    apply fastsimp
    apply (simp only: add_assoc | elim le_push | intro const_commute)+
    apply fastsimp
    apply (simp only: add_assoc | elim le_push | intro const_commute)+
    apply fastsimp
    apply (simp only: add_assoc | elim le_push | intro const_commute)+
    apply fastsimp
    apply (simp only: add_assoc | elim le_push | intro const_commute)+
```

```
    apply fastsimp
    apply (simp only: add_assoc | elim le_push | intro const_commute)+
    apply fastsimp
    apply (simp only: add_assoc | elim le_push | intro const_commute)+
    apply fastsimp
    apply (simp only: add_assoc | elim le_push | intro const_commute)+
    apply fastsimp
    apply (simp only: add_assoc | elim le_push | intro const_commute)+
    apply fastsimp
    apply (simp only: add_assoc | elim le_push | intro const_commute)+
    apply fastsimp
    done
next
  case (land t1 t2) thus ?case
    apply (simp (no_asm_simp))
    apply safe

    apply (simp only: add_assoc | elim le_push | intro const_commute)+
    apply fastsimp
    apply (simp only: add_assoc | elim le_push | intro const_commute)+
    apply fastsimp
    apply (simp only: add_assoc | elim le_push | intro const_commute)+
    apply fastsimp
    apply (simp only: add_assoc | elim le_push | intro const_commute)+
    apply fastsimp
    apply (simp only: add_assoc | elim le_push | intro const_commute)+
    apply fastsimp
    apply (simp only: add_assoc | elim le_push | intro const_commute)+
    apply fastsimp
    apply (simp only: add_assoc | elim le_push | intro const_commute)+
    apply fastsimp
    apply (simp only: add_assoc | elim le_push | intro const_commute)+
    apply fastsimp
    apply (simp only: add_assoc | elim le_push | intro const_commute)+
    apply fastsimp
    apply (simp only: add_assoc | elim le_push | intro const_commute)+
    apply fastsimp
    apply (simp only: add_assoc | elim le_push | intro const_commute)+
    apply fastsimp
    apply (simp only: add_assoc | elim le_push | intro const_commute)+
    apply fastsimp
    done
next
  case (md t1 t2) thus ?case
    apply (simp (no_asm_simp))
    apply safe

    apply (simp only: add_assoc | elim le_push | intro const_commute)+
    apply fastsimp
```

```
    apply (simp only: add_assoc | elim le_push | intro const_commute)+
    apply fastsimp
    apply (simp only: add_assoc | elim le_push | intro const_commute)+
    apply fastsimp
    apply (simp only: add_assoc | elim le_push | intro const_commute)+
    apply fastsimp
    apply (simp only: add_assoc | elim le_push | intro const_commute)+
    apply fastsimp
    apply (simp only: add_assoc | elim le_push | intro const_commute)+
    apply fastsimp
    apply (simp only: add_assoc | elim le_push | intro const_commute)+
    apply fastsimp
    apply (simp only: add_assoc | elim le_push | intro const_commute)+
    apply fastsimp
    apply (simp only: add_assoc | elim le_push | intro const_commute)+
    apply fastsimp
    apply (simp only: add_assoc | elim le_push | intro const_commute)+
    apply fastsimp
    apply (simp only: add_assoc | elim le_push | intro const_commute)+
    apply fastsimp
    apply (simp only: add_assoc | elim le_push | intro const_commute)+
    apply fastsimp
    done
qed

theorem
"ts t ≤ 2 ^ alt t * size t * pdepth t * 180 - size t * 60" (is ?ts_bound)
"ss t ≤ size t * 20" (is ?ss_bound)
"wds t ≤ 2 ^ alt t * size t * pdepth t * 360 - size t * 120"
  (is ?wds_bound)
proof -
  from alt_bound(3) [of t] have "(2 :: int) ^ nwd t ≤ 2 * 2 ^ alt t"
  proof -
    have "(2 :: int) ^ nwd t ≤ 2 ^ (alt t + 1)"
      by (rule power_increasing) (insert alt_bound(3), auto)
    thus ?thesis by simp
  qed
  with unlift_bound [of t] and alt_bound [of t] show ?wds_bound
    by (auto elim: le_push)
  from unlift_bound [of t] and alt_bound [of t] show ?ts_bound
    by (auto elim: le_push)
  from unlift_bound [of t] show ?ss_bound
    by auto
qed

end
```

C. Configurations of Rodin's Auto-Tactic

In the following, I define the configurations of Rodin's auto-tactic used in Chapter 5. The configuration `default` refers to Rodin's default configuration. The configurations whose names have the prefix `unrestricted` or `filtered` use the auto-tactic with the following sub-tactics:

- True Goal
- False Hypothesis
- Goal in Hypothesis
- Functional Goal
- Bounded Goal with finite Hypothesis
- Find Contradictory Hypotheses
- Belongs to domain
- Functional image membership
- Datatype Destructor WD
- Goal Disjunct in Hypotheses
- Partition Rewriter
- Simplification Rewriter
- Type Rewriter
- Shrink Implicative Hypotheses
- Implicative Goal
- Functional Image
- For-all Goal
- Exists Hypotheses
- Shrink Enumerated Set

- Implicative Hypotheses with Conjunctive RHS
- Implicative Hypotheses with Disjunctive LHS
- Remove disjunction in a disjunctive goal
- Use Equals Hypotheses
- Clarify Goal
- Generalized Modus Ponens
- Put in Negation Normal Form
- Conjunctive Goal
- Functional Overriding in Goal
- One Point Rule in Goal
- Functional Overriding in Hypothesis
- One Point Rule in Hypotheses
- Meta prover

The Meta prover always includes the external provers ML, PP, and NewPP (in this order). The timeout of each automated prover is indicated by the name of the configuration; e.g., `unrestricted10` runs ML, PP, and NewPP with timeouts of 10 seconds, respectively. The maximum number of steps of NewPP is chosen as $6t$, where t is the timeout of NewPP in milliseconds.

The configurations whose names have the prefix `unrestricted` only use the filter called "unrestricted". The configurations whose names have the prefix `filtered` use the following filters:

- Restricted
- Meng-Paulson(0.4, 1.2)
- Sub-Expr(0.3, 3)
- Meng-Paulson(0.8, 1.2)
- Sine
- Lasso
- Dcr(0.3, true)
- Unrestricted

The configuration of the Meta prover is similar to the one recommended in [Röder, 2010]. The only exceptions are the inclusion of the ML prover and the "restricted" and "unrestricted" filters.