REVERSIBLE COMPUTATIONS IN B

Thesis submitted to the University of Teesside for the degree of Doctor of Philosophy in the School of Computing

July 2007

By
Dipl.-Inform. Frank Zeyda
School of Computing
Table of Contents

Abstract 7
Declaration 9
Copyright 10
Acknowledgements 11

1 Introduction 12
   1.1 Hypothesis .............................................. 14
   1.2 Objectives .............................................. 15
   1.3 Chapter Overview ...................................... 15

2 Logical Reversibility 17
   2.1 Introduction ........................................... 17
   2.2 Thermodynamics of Computation ......................... 18
   2.3 Reversible pGCL ........................................ 23
   2.4 Conclusions ............................................ 26

3 Reversible Computations in B 27
   3.1 Introduction ........................................... 27
   3.2 The B Method and Notation ............................ 27
   3.3 Reversible Computing, Backtracking and
        B-GSL .................................................. 32
   3.4 Conclusions ............................................ 35

4 Expression Transformers 36
   4.1 Introduction ........................................... 36
   4.2 A First Look at Bunch Theory .......................... 36
   4.3 Intuitive Definition of Prospective-value Expression
        Transformers ......................................... 37
        4.3.1 Generalised Assignment .......................... 40
   4.4 Applications in Programming ........................... 40
        4.4.1 Finding all Results of a Search .................. 40
        4.4.2 Evaluating Set Expressions, Feasibility and
              Quantifications ................................. 41
| 4.4.3 | A Functional Language which Incorporates State-Changing Operations | 42 |
| 4.4.4 | A Mini-max Algorithm | 44 |
| 4.5 | Conclusions | 45 |

### 5 Improper Bunch Theory

| 5.1 | Introduction | 46 |
| 5.2 | Historical Review | 47 |
| 5.3 | Our Adaptation of Classical Bunch Theory | 48 |
| 5.3.1 | Types | 50 |
| 5.3.2 | Inclusion and Equality | 50 |
| 5.3.3 | Primitive Operators | 51 |
| 5.3.4 | Lifted Operators | 52 |
| 5.3.5 | Packaging and Unpackaging | 53 |
| 5.3.6 | Quantified Variables | 53 |
| 5.3.7 | Axiomatisation | 54 |
| 5.4 | Improper Bunch Theory | 54 |
| 5.4.1 | The Improper Bunch | 54 |
| 5.4.2 | Refinement Lattice of Bunches | 55 |
| 5.4.3 | Extensionality | 56 |
| 5.4.4 | The Guarded Bunch | 56 |
| 5.4.5 | The Pre-conditioned Bunch | 57 |
| 5.4.6 | Bunch Comprehension | 57 |
| 5.4.7 | Substitution | 58 |
| 5.4.8 | Functions | 59 |
| 5.5 | Frequently used Bunch Laws | 60 |
| 5.6 | Impact on Tool Support | 66 |
| 5.7 | Conclusions | 68 |

### 6 Prospective-value Semantics

| 6.1 | Introduction | 70 |
| 6.2 | Related Work | 71 |
| 6.3 | Definition of $S \odot E$ | 72 |
| 6.3.1 | Explanation of the Definition for $S \odot E$ | 72 |
| 6.3.2 | Examples for Calculating $S \odot E$ | 73 |
| 6.3.3 | Notes on the Definition for $S \odot E$ | 76 |
| 6.4 | Rewrite Laws for pv Semantics | 77 |
| 6.4.1 | Proof of the Algebraic Laws | 82 |
| 6.5 | Characteristic Predicates | 91 |
| 6.6 | Connection to wp Semantics | 93 |
| 6.7 | Iteration and Recursion | 98 |
| 6.8 | Conclusions | 103 |
7 Refinement of Reversible Computations
  7.1 Introduction .................................................. 104
  7.2 The Reversible Virtual Machine .............................. 106
  7.3 Refinement vs. *-refinement ................................. 107
      7.3.1 Monotonicity Properties .............................. 110
  7.4 The Knight’s Tour Case Study ............................... 115
      7.4.1 The Abstract Development ............................ 116
      7.4.2 The Concrete Development ............................ 120
      7.4.3 Conventional Refinement Proof ......................... 125
      7.4.4 The Click’n Prove Tool ............................... 128
      7.4.5 Translation and Execution ............................. 130
  7.5 Loop Feasibility .............................................. 131
  7.6 Proving *-refinement of the Knight’s Tour ................ 134
      7.6.1 The General Method ................................. 135
      7.6.2 The Specific Method ................................. 136
  7.7 Piecewise and Stepwise Development ....................... 142
  7.8 Conclusions .................................................. 143

8 Conclusions and Future Work .................................. 144
  8.1 Conclusions .................................................. 144
  8.2 Future Work .................................................. 146

A Symbol and Notation Glossary ................................ 151
  A.1 Predicate Logic ............................................. 151
      A.1.1 Predicate Logic (B-AMN) ............................. 151
  A.2 Set Theory .................................................. 152
  A.3 Functions and Relations ................................... 152
  A.4 Sequences ................................................... 153
  A.5 Bunch Theory ............................................... 153
  A.6 Generalised Substitutions Language ....................... 154
  A.7 Program Semantics ......................................... 154
  A.8 Arithmetics ................................................ 155
  A.9 Miscellaneous ............................................... 155

B Hehner’s Bunch Theory Axioms and Laws ....................... 156
  B.1 Hehner’s Bunch Theory Axioms ............................. 156
  B.2 Laws following from these Axioms ......................... 157

C Alternative Proof of the Seq. Comp. Law ..................... 158

D B Development of the Knight’s Tour ......................... 160
  D.1 Abstract Specification: KnightsTourSolver ............... 160
  D.2 Implementation: KnightsTourSolverI ..................... 162
  D.3 Abstract Specification: KnightsTourSolverUtil .......... 163
  D.4 Implementation: KnightsTourSolverUtilI ............... 164
D.5 Abstract Specification: KnightsTourSolverGeneric ........ 165

E Correctness Proof of the Knight’s Tour .................. 167
E.1 User Theory for Discharging POs in KnightsTourSolver1 .................. 167
E.2 Example of an Interactive Proof using Click’n Proof ................. 170

Bibliography .................................................. 176
## List of Figures

2.1 Cylinder containing one molecule of gas. .......................... 18  
2.2 Pair of compass needles forming a bistable well. .............. 19  
2.3 Energy free toggling of a bit. ................................. 20  
2.4 The irreversible “set to 1” operation. ......................... 21  
2.5 Illustration of Szilard’s Paradox. .............................. 21  
3.1 Rules for wp of Generalised Substitutions Language construct. . . 30  
4.1 Prospective-value expression transformer rules for rewriting $S \diamond E$. 39  
6.1 Definitions of trm and prd of a generalised substitution $S$. 72  
6.2 Characteristic predicates trm and prd of each basic GSL construct. 76  
6.3 Algebraic laws for rewriting $S \diamond E$ for each GSL construct. .... 78  
6.4 Characteristic predicates trm, fis and prd in both wp and pv semantics. 93  
7.1 Blob diagram illustrating proper feasibility-preserving refinement. 109  
7.2 Blob diagram 2 illustrating feasibility-preserving refinement. .... 110  
7.3 CONSTANTS clauses of the KnightsTourSolver B machine. .... 117  
7.4 Operation select_move in KnightsTourSolverUtil. ........... 122  
7.5 Operation check_unvisited in KnightsTourSolverUtil. ....... 122  
7.6 Super-concrete refinement of the select_move operation. .... 141
Abstract

We investigate how formal software development, in particular the B Method, can be adopted to specify and implement reversible computations. By reversible computations we meant those which are reversible in a stepwise manner, i.e. for which each atomic construct of the command language has a right inverse with respect to sequential composition. Doing so we revoke Dijkstra’s law of the excluded miracle by admitting infeasible computations as effectively executable constructs, and thus appeal to their operational interpretation in terms of backtracking.

Our Reversible Virtual Machine provides a run-time platform which implements these constructs by ensuring reversibility of each computational step. In this context we regard non-deterministic choice as provisional choice which clearly plays a different role in the specification of an operation as opposed to implementor’s choice, and is not expected to be reduced in the refinement process. Since semantically these choices are identical, and we naturally use the same symbol for them, a problem we address is how they can be distinguished within the formalism, and how we achieve the desired interaction between them i.e. with regards to refinement.

To increase the expressive power of our approach for developing reversible computations we introduce the idea of expression transformers, allowing us to determine the value of an expression after running a computation without incurring any side-effects. A theory of prospective values is developed which defines the prospective value of any computation expressed in the Generalised Substitution Language of B (GSL) by referring to its semantics in terms of the characteristic predicates trm and prd. We show that prospective-value semantics is isomorphic to other well-established total-correctness semantic models of the GSL. To accommodate the outcome of non-deterministic, infeasible and abortive computations at the level of expressions we adopt Hehner’s bunch theory and incorporate an ‘improper’ bunch for each type in order to describe the outcome of a possibly non-terminating program.

We furthermore investigate the suitability of conventional refinement in the light of stand-alone guards and choice statements in implementations, and address the problem of vacuously refining a computation via magic, which in our setting is an executable statement, by providing a stronger, feasibility-preserving refinement relation. We show the shortcoming of the proposed refinement not being monotonic in all operators, and propose solutions how a piecewise and stepwise
approach may be retained despite loss of monotonicity. We provide a practical illustration by means of a case study how the B Method and tool support can be used in developing the a solver for the Knight’s Tour chess problem in our reversible computing paradigm, and show how feasibility-preserving refinement, in the special case of the implementing operation utilising a loop construct that backtracks across the boundaries of the loop body, may be proved.
Declaration

No portion of the work referred to in this thesis has been submitted in support of an application for another degree or qualification of this or any other university or other institution of learning.
Copyright

Copyright in text of this thesis rests with the Author. Copies (by any process) either in full, or of extracts, may be made **only** in accordance with instructions given by the Author and lodged in the Learning Resource Centre of the University of Teesside. Details may be obtained from the Librarian. This page must form part of any such copies made. Further copies (by any process) of copies made in accordance with such instructions may not be made without the permission (in writing) of the Author.

The ownership of any intellectual property rights which may be described in this thesis is vested in the University of Teesside, subject to any prior agreement to the contrary, and may not be made available for use by third parties without the written permission of the University, which will prescribe the terms and conditions of any such agreement.

Further information on the conditions under which disclosures and exploitation may take place is available from the head of the School of Computing.
Acknowledgements

First of all I would like to thank my PhD supervisor Bill Stoddart and second supervisor Steve Dunne at the University of Teesside for their continual guidance and support which I received over the period of research and write-up for this PhD investigation. The time working with Bill and Steve has been most inspiring and academically stimulating; the fact that it took five years to complete and submit the thesis I would like to be understood as a testimony of the intellectual enjoyment I derived from being a student of Bill and Steve.

On a further note I would like to extend my sincere thanks to Marc Cavazza who is Dean of Research in the School of Computing at the University of Teesside for providing financial support in form of two bursaries, in particular when working towards the submission of the thesis. I am also grateful to ESF for providing additional funding during the first two years of the PhD degree, and the University of Teesside for sponsoring various conference visits.

There are a number of academics I would like to name in particular for their help and notable contribution: Dominique Cansell for educating us in the Atelier B interactive theorem prover and the Click’n Prove tool, Louis Mussat for his interest in pv semantics and a suggestions at the ZB 2005 conference leading to a nifty theorem and simplification of a complex proof, Henri Habrias for his hospitality during three visits to the University of Nantes on which I also was given the opportunity to present some of my research in a faculty seminar, Eric Hehner for helpful discussion on bunch theory at the First International Symposium on Unifying Theories of Programming, Joe Morris and Alexander Bunenburg for providing us with an early copy of their “Term Transformer Semantics” paper which inspired some of the work on prospective-values semantics in this thesis, and a special friend from Brunswick who I had many an interesting evening and discussion with on computer science and mathematics in general.

Special thanks go to my parents and sister in Germany for their loving support and moral encouragement.
Chapter 1

Introduction

This PhD thesis has two major themes. The first one is to investigate the effect on formal software development of regarding computation as an essentially reversible process. Programs written in sequential programming languages normally erase information as they run; for example the assignment $x := 7$ erases the former value of $x$. Our target execution platform, the Reversible Virtual Machine (RVM), preserves such information, and is able to reverse back to any previous state, though it cannot, of course, reverse the effect of any interaction it may have had with its environment. Whereas the RVM, which has been collaboratively developed in the Formal Methods and Programming Research Group at the University of Teesside, demonstrates reversibility of program execution at a pragmatic level, this PhD thesis sets out to investigate the repercussions of our particular method of making reversible computations available to the implementor by means of an abstract command language such as the Generalised Substitution Language of the B Method on a semantic level. This is done with particular emphasis on subsequent integration into the B Method and tool support.

The motivations for such an investigation are threefold. Firstly, it is possible to describe reversible computations in the Generalised Substitution Language, or indeed any other similar model of sequential programming, such as, for example, of Hoare-He designs, in a concise and elegant way. This is achieved by revoking the ‘Law of the Excluded Miracle’ of Dijkstra, and interpreting non-deterministic choice as a provisional choice. Thus we obtain a language which can easily generate the results of a backtracking search. This is widely known, and was indeed one of the original motivations for interest in non-determinism. However, it was not known what additional forms of reversibility could be described, or how best the presence of miracles should be handled in managing the refinement of computations. In conventional backtracking implementations\(^1\) we usually have to provide program logic which updates the state variable(s) that record(s) the current point of exploration in the search space. How in particular this takes place

\(^1\)Here we clearly consider languages that do not have any special support for backtracking.
using either global or local variables, or recursion in favour of a loop may vary between programs, however the relevant feature they share is that they have to deal with two cases, one in which exploration proceeds (the current search path is extended), and another in which exploration ‘backtracks’ (the current search path is restored to a previous location in the search space). In the approach to backtracking we propose here the restoration of the variable(s) representing the search path does not have to be carried out explicitly by the program; it is implicitly implied by the semantics of the underlying programming constructs. The consequence of this is that we obtain a more concise formulation of backtracking algorithms which also lends itself more easily for formal analysis. Indeed this form of backtracking is also mentioned by Hehner in [Heh93]. The second motivation is that organising computations in a reversible manner has practical advantages in terms of garbage collection, enabling us to provide a greater scope for mathematically tractable data types in our implementations. Our RVM platform exploits this by includes an efficient and fully general implementation of finite typed sets. With this we can consequently liberalise the restrictions impose on the type of concrete (meaning program) variables while retaining a high degree of efficiency and integrity of the run-time system. The third motivation is that studies in the energy requirements of computation suggest strongly that the only inevitable requirements for energy dissipation occur during computation steps which are non-reversible. Thus our interest in reversibility does not imply that we distance ourselves from the underlying physical mechanisms of a computation, as is the case for functional and logic programming styles, but rather gives us the scope to describe computations in the manner which leads to the minimum possible inescapable energy requirements.

The second major theme of the thesis is an alternative semantics of sequential programs based on the transformation of expressions rather than predicates. This theme emerges from our attempts to widen the kind of reversible computations which can be described using B-GSL, or related formalisms. We first meet expression transformers as a means of characterising the set of all possible results from a backtracking search, and indeed they are executable constructs on our RVM. However, it seems that in addition to this specific use within a reversible programming language, we can moreover use them to provide a characterisation of any sequential program within the refinement lattice of generalised substitutions. In this respect our aim is to develop a theory of prospective-values which offers an alternative semantic foundation for the GSL. Part of the preliminaries of this work require the extension of Hehner’s bunch theory, which we utilise to represent the outcome of non-deterministic computations, in such a way that non-termination can be expressed at the level of values. This will subsequently lead to a theory of bunches tailored for our purposes to reason about generalised substitutions in terms of prospective values, this theory we will call Improper Bunch Theory.

Throughout the period of research, a number of related practical programming projects were undertaken. OpenB is an extensible Java component library which
currently provides facilities to parse and type check B machine constructs and per-
forms some limited translation from B to the target language provided by the Re-
versible Virtual Machine. Unlike current commercial products (i.e. the B-Toolkit and Atelier B) it provides a complete type analysis, and thus does not depend on particular syntactic forms for the sake of making type information available. A second practical contribution is the sets package within the RVM which provides a complete implementation of finite sets. The third is the HeapWatch tool, which can be optionally linked when the RVM is compiled, and which is used to ensure that reversible computation really does recover all dynamic memory allocations. These contributions are not specifically reported in this thesis, partly because we see the main contribution of this PhD research project being of a theoretical nature, but they have informed our approach during the course of the research. The sets package and heapwatch tool are documented in the RVM manual, available from our web-site at http://www.scm.tees.ac.uk/formalmethods/. The OpenB tool library will shortly be made available under the GPL in a preliminary release at http://openb.scm.tees.ac.uk/.

1.1 Hypothesis

Our thesis is that there are advantages in conceptualising computations as re-
versible processes, that the semantics of reversible programs may be expressed in terms of standard formalisms such as the Generalised Substitution Language (GSL), and that the B Method can provide a convenient formalism and framework for their formal development. This entails in particular the revocation of Dijkstra's Law of the Excluded Miracle on the level of implementation in order to exploit the interaction of infeasibility and non-deterministic choice to obtain a form of inherent backtracking within our program semantics which lends itself in particular well to be implemented by a reversible execution architecture, such as, for example, the Reversible Virtual Machine (RVM) which we developed and propose as a pragmatic experimentation platform for reversible computing.

A secondary thesis which emerges in the work is that we can obtain an alternative form of semantic description, equivalent in power to the wp calculus, through considering the prospective values of expressions that might arise from a (possibly non-deterministic) computation. This description of computation allows us to overcome a short-coming of information being erased when backtracking occurs as a consequence of guards and choice statements, and its particular benefits are that we can collect, for example, all the results of a backtracking search rather than just one, and thereby increase the expressive capabilities of the paradigm of formally developing reversible computations which we propose in this PhD.
1.2 Objectives

Our objectives to achieve the hypothesis formulated above are as follows:

1. to outline the theory of reversible computation and its motivation in terms of the energy requirements and the thermodynamics of computation;

2. to show how reversibility can be expressed within a formal programming semantics and how a combination of guards and choice can be used to describe the execution of a backtracking algorithm;

3. to introduce and axiomatise a variant of Hehner’s Bunch Theory, designed to simplify the presentation of our theories;

4. to show how we can extend the idea of backtracking search by means of a prospective-value formalism, allowing a program that can find one result of a search to be encapsulated in an expression representing all results of such a search;

5. to show how our prospective-values formalism can form an alternative semantic foundation for a sequential programming language exhibiting equal expressive power as, for example, the wp calculus;

6. to demonstrate the development of a backtracking algorithm on a reversible computing platform and consider how the theory of refinement must be adapted to cope with the particular circumstances arising in our formulation of reversibility.

7. to investigate the applicability of B tools for developing applications exploiting reversibility through backtracking by means of a case study substantial enough to raise issues associated with the piecewise and stepwise development methodologies.

1.3 Chapter Overview

Chapter 2: “Logical Reversibility”. In this chapter we review related work on the energy requirements of computations, from the early analysis of Landauer in 1961 to recent work of Zuliani on reversible pGCL. We establish the important connection between the minimum energy requirement of a computation, and the irrevocable loss of information in its steps. Furthermore we explain how program assignment can be implemented in a genuinely reversible manner.

Chapter 3 “Reversible Computations in B”. This chapter commences with an overview of the B Method in particular mentioning those aspects which are relevant to our work. We then show how repealing the Law of the Excluded Miracle can allow us to relate reversibility and backtracking, and to describe the effects of a backtracking search that finds a single solution.
Chapter 4 “Expression Transformers”. In this chapter we introduce the idea of expression transformers to overcome a limitation in our approach which makes it impossible to remember the intermediate result of a computation prior to reversing. Importantly, this also allows us to generate all the solutions of a backtracking search rather than just one. The result is a language with an extended sub-language of expressions in which operations can be embedded without incurring the penalty of state-changing side-effects. In practical terms side-effects can always be undone through reverse execution, making the expression transformer presented in this chapter in particular amenable for implementation in our RVM. We also show how expression transformers may be used in combination with recursion.

Chapter 5 “Improper Bunch Theory”. This chapters lays the foundation for a subsequent formal introduction of prospective-value semantics in the following chapter. Since our ambition is to represent the outcome of all computation within the refinement lattice of generalised substitutions, we need an expression formalism rich enough to describe the outcome of non-deterministic, infeasible and abortive computations. This led to the development of Improper Bunch Theory which is in detail presented in this chapter.

Chapter 6 “Prospective-value Semantics”. In this chapter we develop a formal theory of prospective values by relating them to the semantics of a generalised substitutions in terms of the characteristic predicates trm and prd. We show how the prospective value of any computation can be calculated in practice, and present a number of algebraic rewrite laws that simplify evaluation. We furthermore establish alternative links between the prospective value and wp semantics of a computation, and conclude the chapter by considering the prospective value of loop constructs.

Chapter 7 “Refinement of Reversible Computations”. In this chapter we conduct a case study illustrating how the B Method and tool support can be exploited to develop the non-trivial application, finding a solution for the Knight’s Tour problem in a reversible manner. The main contribution of this chapter is to introduce the stronger notion of feasibility-preserving refinement to address the problem of “over-refinement”. We provide a method that allows us to verify this stronger refinement in cases where backtracking occurs across the boundaries of a loop bodies.

Chapter 8 “Conclusions”. In Chapter 7 we draw our conclusions and propose directions for future work.
Chapter 2

Logical Reversibility

2.1 Introduction

In this chapter we consider the physical properties of reversible computation and in particular the relationship between reversibility and energy requirements. Hereby we do justice to objective 1 of the thesis, see Section 1.2. We then review the work of Zuliani on reversible Guarded Command Language and provide a more exact analysis of the reversibility of the assignment statement. The latter addresses to some extent objective 2 of our investigation which is to show how reversibility can be expressed within a formal programming semantics; the primary account on this topic, however, will follow in the next chapter.

As computer scientists we are accustomed to abstracting away from particular computing mechanisms when thinking about the meaning of computation in mathematical terms. Our aim in this section, however, is to consider a computation as a physical process with particular regard to its necessary energy requirements. One way to anticipate our arguments in this chapter is to consider a collection of balls moving on an idealised billiard table (with no pockets) on which balls roll and rebound from the cushions with no energy loss. The laws of motion in such a system, which due to its conservation of energy is called a “conservative system”, are deterministic and reversible, and at any time its previous history could be recovered if we could exactly reverse the direction of movement of each of the balls. In comparison, on a system where damping occurs, e.g. kinetic energy of the balls is absorbed by the environment through friction, the balls will eventually come to rest, and the system no longer contains the information required to recover its past history. This example suggests an association between damping (energy consumption) and the loss of information (irreversibility) which we will now attempt to develop, along with a discussion of the minimum energy requirements involved.
2.2 Thermodynamics of Computation

In a talk given in 1949 and later published in [vN66], John von Neumann remarked that there must be a dissipation of $k \ast T \ast \ln(2)$ units of energy per elementary act of information, that is per elementary decision of a two way alternative and per elementary transmission of one unit of information. His analysis is based on the assumption that each “elementary act” removes one bit of uncertainty from the result of a computation, thus reducing the entropy within the computer by the classical thermodynamic quantity $k \ast T \ast \ln(2)$, and requiring an equivalent energy dissipation to the environment.

This roughly sketched theory remained unchallenged until 1961 when Landauer provided an analysis based on determining the essential function of energy consumption during computation [Lan61], and found that it was only necessary for ‘standardising signals and making them independent of their exact logical history’, i.e. that energy consumption was only required for the irreversible steps of a computing process.

To explore these concepts we need a physical model that allows information to be recorded in a material way. Because the laws of thermodynamics apply in a very general way, we will not analyse a practical model of memory storage, but rather one which allows us to perform our demonstrations in the simplest of ways. Fig. 2.1 shows a cylinder containing a single molecule of ideal gas having a piston at each end. Initially the molecule is free to move anywhere within the piston. A zero is registered by moving inwards the piston on the left, and thus restricting the molecule to the right half of the cylinder.

![Figure 2.1: Cylinder containing one molecule of gas.](image)

A value of one is similarly registered by moving in the piston on the right as shown above. This model has been used by Feynman [Fey96], Bennett [Ben82] and others, and in a slightly more elaborate form dates back to a paper from by Szilard in 1929 [Szi29].

In the lower diagram of Fig. 2.1 the piston on the right has been moved in against the pressure of the gas (whose molecular movement we interpret in a time-averaged sense) so that the gas is compressed to half its previous volume. As the piston starts to move the initial effect is to increase the energy of the molecule, which is now rebounding from a moving surface and thus gains speed. However,
we will assume an isothermal compression, that is one in which the additional energy of the molecule is rapidly absorbed by the environment. Therefore we can assume the compression of the gas takes place at a constant temperature. The Ideal Gas Law tells us that the pressure and volume of a gas at temperature $T$ are related by $P \times V = N \times k \times T$ where $k$ is Boltzmann’s constant ($\approx 1.3806503 \times 10^{23} \text{JK}^{-1}$), $N$ is the number of molecules in the gas and $T$ is its absolute temperature in degrees Kelvin. If the distance between the pistons in our cylinder is $L$, and the area presented to the gas by each piston is $A$, the volume of the gas is $V = A \times L$. For a single molecule we thus have $P \times A \times L = k \times T$. The force exerted by the gas on one piston, given by $F = P \times A$, is thus $k \times T \times L^{-1}$.

To find the minimum work needed to compress the gas to half its volume we need to integrate this force between an initial distance separating the pistons, say $L_0$, and $\frac{1}{2}L_0$:

$$W = \int_{L_0}^{\frac{1}{2}L_0} \frac{k \times T}{L} \, dL = k \times T \times \ln\left(\frac{L_0}{\frac{1}{2}}\right) - k \times T \times \ln(L_0) = -k \times T \times \ln(2)$$

A very interesting property of this result is that it depends neither on the mass of the molecule or the size of the cylinder, and is, in fact, the general result for the change of entropy associated with constraining a particle to half its phase space along one of its degrees of freedom in any thermal system.

We can similarly recover from this compressed gas $k \times T \times \ln(2)$ of free energy when allowing it to re-expand. Note that too this energy is not obtained from the gas, but from the environment: its availability is due to the particular configuration of the system.

The representation of a bit of data is more realistically characterised as some form of bistable well. Fig. 2.2 represents orientations of a pair of magnets, linked so that they are always at the same angle. The ensemble has two positions of stable equilibrium, respectively representing the 1 and 0 state.

![Figure 2.2: Pair of compass needles forming a bistable well.](image)

In Fig. 2.2 we see the magnets being moved from a 1 to a 0 state. The graph
below represents the potential energy of the ensemble as the two linked needles pass through different angles of rotation. The reversible operation of switching from a 1 to a 0 state can essentially be performed without consumption of energy, since the energy required to move “up” to the state of unstable equilibrium can, in principle, be recovered whilst moving “down” to the 0 state. The same analysis applies to switching from a 0 state to a 1 state, but what about the operation of just toggling the bit (without knowing its current state)? This can be done in an energy free manner by rotating the whole ensemble about its centre point as shown in Fig. 2.3.

![Figure 2.3: Energy free toggling of a bit.](image)

Now let us consider the irreversible operation “set to 1” illustrated in Fig. 2.4. Recall that a conservative system obeying the laws of motion is both reversible and deterministic. An energy free “set to 1”, therefore, would have a deterministic reverse trajectory. However, the reverse trajectory of “set to 1” is non-deterministic, since it has to include the possibilities of returning to a previous state of either 0 or 1. This contradiction tells us that our assumption that “set to 1” can be performed in a conservative system is incorrect, and therefore some damping is required in this case. Before we definitively accept this conclusion however we must dispose of the following counter argument. ‘The reverse trajectory need only be non-deterministic if all previous history has been lost by the “set to 1” operation. So long as we allow some residual difference in the configuration of a 1 bit that was previously a 1 and, a 1 bit that was previously a 0, the argument breaks down. Also, such differences are seen on real devices, as, for example, in the ability to perform forensic analysis of a blanked area of a hard disk and retrieve its previous contents’. Although this argument has some force as far as one single operation is concerned, it is not possible to employ it in any physically realistic way over a continued sequence of operations without the necessary residuals building up and disabling the device’s capacity to store information. As Landauer says ‘the physical many-to-one mapping, which is the source of the entropy change, need not happen in full detail during the machine cycle which performed the logical function, but it must eventually take place, and this is all that is relevant for the heat generation argument’.

So far we have considered two separate arguments based respectively on the energy needed to constrain a particle along an information bearing degree of
freedom, and on the reversibility of the laws of motion in a conservative system. To draw these together consider again our cylinder of gas. What if, instead of compressing the molecule into one half of the cylinder by exerting force on one of the pistons, we had trapped the molecule on one side of the cylinder by inserting a partition, as shown in the upper cylinder of Fig. 2.5. This is an operation that, in principle, requires no work.

We could then push in the piston from the side that does not contain the gas. Since we are not moving the piston against any resisting force this action again, in principle, requires no work. We could then remove the partition and be in a position to extract work from the piston without having put any work in.

This apparent paradox was proposed by Szilard [Szi29] in 1929 as an aid to analysing the closely related “paradox” of Maxwell’s Demon. Szilard looked for a compensating energy input in the measurement that would need to be made before it could be decided which piston to move. His analysis was universally accepted until Landauer’s colleague at IBM Research, Charles Bennet, showed that measurement is not intrinsically an energy consuming process, and, applying Landauer’s analysis, pointed out that the mechanism that registered which side the molecule was on would itself have to perform an irreversible operation analogous to the “set to 1” operation described above; indeed this is where a necessary energy input must occur.

Landauer’s analysis led him to the conclusion that a computation which is reversible at each step would, in principle, have a zero minimum energy requirement. What exactly constitutes a step is dependant on the underlying physical architecture performing the computation, however in [Lan61] he argues that for
the sake of the entropy generation argument we may abstract away from the physical details of the computing process in such a way that we only have to consider its logical steps\(^1\) since, as he concludes in his paper, “logical irreversibility implies physical irreversibility”.

Landauer also reasoned that computing inevitably uses irreversible steps, as for example the assignment \(x := 0\) cannot be reversed because it destroys the initial value of \(x\). Such steps are inevitably associated with the consumption of a certain minimum amount of energy. If a computing process could be contrived which used only reversible steps, then the laws of thermodynamics would not impose any minimum energy requirement for the computation. He notes that individual steps in a computing process can be made reversible by providing additional memory storage to preserve data that would otherwise be lost, but rejects this as a general technique as the result would be an unpredictable requirement for additional memory which would need to be irreversibly initialised to a known value: ‘Our unwieldy machine has therefore avoided the irreversible operations during the running of the program, only at the expense of added comparable irreversibility during the loading of the program.’[Lan61]

This conclusion was incorrect, because we can organise the required additional memory efficiently as a stack, and regard its initialisation as a one-off cost which, once paid, will allow us to run all subsequent programs in a reversible manner. Despite this one erroneous conclusion Landauer’s 1961 paper made the seminal contribution in setting the terms for a debate on reversibility and was republished in volume 44 of the IBM Journal of Research and Development in 2000.

In 1963 Lecerf [Lec63] formulated a reversible Turing Machine which potentially indicated how reversible computations could be managed, but this work did not feed into the reversible computing debate. However in 1973, Bennett described how an arbitrary (one tape) Turing Machine could be translated into a reversible three tape machine [Ben73]. The latter performs the calculation of the original machine, storing any overwritten data on the (originally blank) second tape. It then copies the result to the third tape. Finally it reverses its calculations so as to terminate with the first tape being back in its original condition, the second tape once again blank, and the result left on the third tape. Fundamental to Bennett’s analysis is that writing to a blank tape is a reversible operation. The blank second tape plays the role of the pre-initialised memory mentioned above. Bennett linked his machines to the energy requirements of computing with the comment that ‘[Turing] Machines may be made logically reversible at every step. This . . . makes plausible the existence of thermodynamically reversible computers which could perform useful computations at useful speed while dissipating considerably less than \(kT\) of energy per logical step.’

\(^1\)Logical steps may be expressed in terms of state transition functions, some abstract command language such as the Generalised Substitution Language, etc.
2.3 Reversible pGCL

An interesting contribution to reversible computing is given in the paper “Logical reversibility” by P. Zuliani [Zul01]. The author similarly provides a formulation of a reversible computation in terms of a non-reversible computation. However, rather than using Turing Machines he formulates his translation in terms of the pGCL, a probabilistic extension of the Guarded Command Language of Dijkstra [MM99]. His work extends reversibility to computations involving non-deterministic and probabilistic choice, and presents it in a form suitable for incorporation into a software development method.

His technique for making an irreversible language reversible is to add some extra state, in the form of a single Boolean variable $b$ and a history stack, and to transform each operation $S$ in the language into a reversible operation $S_r$ which has the same effect as $S$ on the original state space, and which uses the history stack to preserve any information that would otherwise be lost when $S$ is executed. For each $S_r$ he provides an inverse operation (actually its right inverse with regards to sequential composition) $S_i$ such that $S_r; S_i = \text{skip}$. The technique is illustrated in the following table where we give these constructs for the assignment and choice statements.

<table>
<thead>
<tr>
<th>$S$</th>
<th>Reversible Operation $S_r$</th>
<th>Inverse Operation $S_i$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$v := e$</td>
<td>$\text{push } v; v := e$</td>
<td>$\text{pop } v$</td>
</tr>
<tr>
<td>$R [] T$</td>
<td>$\text{push } b$; $\left( R_r; \text{push true} \right) [\left( T_r; \text{push false} \right)$</td>
<td>$\text{pop } b; \left( R_r \bullet b \triangleright T_i \right); \text{pop } b$</td>
</tr>
</tbody>
</table>

We would like to take this analysis a stage further because it does not, as it stands, provide the stepwise reversibility which the arguments of the previous section has emphasised. For example, the inverse of the assignment to $v$ is $\text{pop } v$, a non-reversible operation, and the transformed assignment statement, $\text{push } v; v := e$, contains the irreversible step $v := e$.

We will show an alternative approach to Zuliani of a reversible assignment which gives a transformation consisting of the sequential composition of reversible steps. We initially limit our discussion to a language with integer variables and we consider only the assignment of a single variable rather than a variable list. We first note that we have some reversible assignment statements to call upon, namely those of the form $x := x + e$, where $x \notin e$ meaning $x$ does not occur free in $e$. Such a statement has an inverse $x := x - e$. We set ourselves the problem of implementing general assignment purely in terms of reversible assignment statements.
We first convince ourselves that assignment to a zero-valued variable is reversible since:

\[ x = 0 \mid x := e = x = 0 \mid x := x + e[x\backslash 0] \]

Here the precondition \( x = 0 \) allows us the freedom to implement \( x := e \) by the reversible command \( x := x + e[x\backslash 0] \), note that \( x \) is not free in \( e[x\backslash 0] \).

In what follows we will again assume the implicit availability of a history stack, and briefly remind the reader of its purpose. In general terms the history stack provides a means for recording information that would otherwise be lost in the computational process, and thereby constitute one approach to model reversible computations by maintaining sufficient information to undo each computational step. We should note in response to the examiners’ comments that this is not the only possible approach, e.g. see Bennet’s 3-tape reversible Turing Machine explained in the previous section. What these models however have in common is the notion of a history of states or traces which they update as execution proceeds. The reason we decided to use a stack for this purpose (as Zuliani) is that the additional information required to record the history can be efficiently organised in this way requiring only an initial one-off investment of energy to initialise each element of the stack to some known value. This architecture moreover mirrors very closely the working of our Reversible Virtual Machine, to be discussed in Section 7.2.

The role of the history stack will be taken now by an integer array \( h \) having a large enough size \( hsize \). We also employ an array index \( i \) which will, loosely speaking, be used as a stack pointer. Note \( h \) and \( i \) are fresh variables with respect to the original program. We assume the elements of \( h \) are initialised to zero and \( i \) is initialised to one (requiring an initial investment of energy).

We can give a reversible transformation of \( x := e \) as a sequence of reversible commands, as shown in the following command trace:

<table>
<thead>
<tr>
<th>Assignment</th>
<th>( h(i - 1) )</th>
<th>( h(i) )</th>
<th>( h(i + 1) )</th>
<th>( x )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( h(i) := h(i) + x )</td>
<td>?</td>
<td>0</td>
<td>0</td>
<td>( x_0 )</td>
</tr>
<tr>
<td>( h(i + 1) := h(i + 1) + e )</td>
<td>?</td>
<td>( x_0 )</td>
<td>0</td>
<td>( x_0 )</td>
</tr>
<tr>
<td>( x := x - h(i) )</td>
<td>?</td>
<td>( x_0 )</td>
<td>( e )</td>
<td>( x_0 )</td>
</tr>
<tr>
<td>( x := x + h(i + 1) )</td>
<td>?</td>
<td>( x_0 )</td>
<td>( e )</td>
<td>( e )</td>
</tr>
<tr>
<td>( h(i + 1) := h(i + 1) - x )</td>
<td>?</td>
<td>( x_0 )</td>
<td>0</td>
<td>( e )</td>
</tr>
<tr>
<td>( i := i + 1 )</td>
<td>( x_0 )</td>
<td>0</td>
<td>0</td>
<td>( e )</td>
</tr>
</tbody>
</table>

The respective reverse operation of \( x := e \) may be derived from the inverse operations of each of the above steps:
The transformation of $x := e$ and the inverse operation given above are only correct under the assumption $h(i) = 0 \land h(i + 1) = 0$, which is implied by the stronger condition:

$$i \in 1..\text{hsize} - 1 \land h(1..\text{hsize})^2 = \{0\}$$

which is invariant in the sense that it is established by the initialisation of the history stack, and is preserved by each forward and reverse assignment under the assumption that hsize is sufficiently large. It must similarly be preserved by all other operations.

A second remark we need to make concerning the above analysis is that we consider the value of the expression $e$ to have been computed and stored in some suitable register before the assignment is made. Subsequent to the reverse execution of the assignment would therefore be the “uncomputation” of the value $e$.

**Note on Non-integer Assignment**

We can describe the stepwise construction of non-integer assignments using an extra level of refinement in which we introduce some detail that, for normal purposes, can be left to the compiler of our implementation-level code to incorporate.

This implicit level of extra refinement interprets the execution of non-integer assignment statements in terms of a reference semantics. Let the values of some type $D$ that arise during a calculation be denoted by $d_1, d_2, \ldots$. With each value $d_i$ we associate a unique reference $rd_i$ of type $N$. Let $\text{Ref}_D \in N \mapsto D$ be the partial injection $\{rd_1 \mapsto d_1, rd_2 \mapsto d_2, \ldots\}$. We now consider the statement $v := d_i$ as an abstract assignment, even though it may occur in the concrete implementation language, which is refined (by the compiler) into $v_r := dr_i$. The refinement formally requires an abstraction predicate, here given by $\text{Ref}_D(v_r) = v$.

Since all assignment is now implemented as integer assignment, our previous analysis will suffice to show that assignment in general can be carried out in a

\[\text{Note that } (\ldots \ldots) \text{ is the relational image operator, see Appendix A.3.}\]
stepwise reversible manner. Note that as in the theory of B we interpret assignment throughout as value assignment e.g. $x := y$ always means assign $x$ to the value of $y$ (not let them point to the same object). This is important since the absence of pointer or reference variables avoids associated problems which might occur as a consequence of aliasing; such we therefore don’t have to deal with in our work.

2.4 Conclusions

We have reviewed the area of reversible computation, and its association with the minimum energy requirements of a computation, from the early analysis of Landauer to the recent work of Zuliani on pGCL. The latter presents reversible computing in a suitable context for incorporation into a formal development method, but does not show explicitly how irreversible (and thus energy consuming) steps may be eliminated during reverse execution of assignments. We have given a more detailed analysis of reversible assignments which covers this point.
Chapter 3
Reversible Computations in B

3.1 Introduction

Having introduced the idea of reversible computation from a physical as well as logical point of view through the reversible Guarded Command Language, in this chapter we turn our attention to the language used in the B Method, which is also called the Abstract Machine Notation (AMN). Here we review the essential notations of B and compare them to related formalisms. We then consider, under the assumption of a reversible execution platform, how we can exploit the interaction of guards and choice to provide a simple form of backtracking. The content of this chapter satisfies objective 2 of the thesis showing how reversibility can be expressed within a formal programming semantics, in particular the one of generalised substitutions.

3.2 The B Method and Notation

The B Method is well documented [Abr96b], but we will summarise its main points and the notations we will be using, and set it in a general context of formal methods and notations for software specification and development.

B is a state-based formal method for specification and software construction. As advocated by Morgan [Mor88], a single language is used to encompass both specifications (thought of as abstract programs) and implementations (concrete programs). Analysis of programs written in B is performed in terms of a modified form of Dijkstra’s weakest-precondition analysis [Dij76]. B has many points in common with the Z specification language [Spi92] which was also proposed by Abrial. These points include the use of a strongly-typed or multi-sorted set theory in which types are maximal sets, the extensive use of partial functions, and the use of classical two-valued logic. One point the B Method and Z differ significantly is that B makes a very clear distinction between preconditions and guards, with separate predicate transformer rules for these two constructs. Another difference is that Z is conceived as a specification notation, and has no executable subset.
Z, like the approach of Hehner in [Heh93] and Hoare-He “designs” presented in their unifying theories of programming [HJ98], regards computations as relations by means of before/after-state predicates.

The combination of partial functions and classical logic found in B and Z raises issues of undefined terms, and has been criticised by Jones, e.g. in “Partial Functions and Logic, a warning” [Jon95]. Jones’ own formal method, VDM [Jon86], uses a before/after predicate approach with a special three-valued logic of partial functions.

Along with the other methods mentioned here, but with the notable exception of Hehner’s approach, B has no finer-grained concept of time other than termination. Thus computations developed in B are guaranteed to terminate within their precondition, but no upper bound for the time this will take can be subject to reasoning.

One aspect of B that will be significant for our approach is the decomposition of selection constructs into more primitive choice and guard constructs, an approach first formulated by Hehner and described in the recent paper “Retrospective and Prospective for Unifying Theories of Programming” [Heh06]. In this approach, an IF . . . ELSE . . . construct is decomposed into a choice between two naked guarded commands. The existence in B of the naked guarded command with form \( g \implies S \), which arises from this decomposition, also suggests an event-based interpretation in which \( g \) acts as a firing or enabling condition, and \( S \) describes the effect of the event. This interpretation, known as “Event B” [Abr96a], has shown itself to be extremely fruitful in modelling distributed systems, as in [ACM03], and we have used it ourselves to model an interrupt driven scheduler [SCZ07]. Although event-based modelling is not investigated in this thesis, its development has led to a certain flexibility in the way support tools for B treat naked guarded commands, which are not allowed in normal software development as they offend Dijkstra’s “law of the excluded miracle”, and more importantly cannot be translated directly into the supported target languages by the B tools. Checks that reject such commands can be turned off in the B support tools we have used, namely the Atelier B, and this is something we crucially had to make use of.

Software development in B is supported by a set of mature software tools:

- most notable the Atelier B from Clesry, extensively used in developing software for the French railway industry;
- Click’n Prove, a graphical front end for the theorem prover to the Atelier B;
- B-Toolkit, produced by B-Core in the UK;
- a new generation of tool support for B currently being produced as part of the RODIN project funded by EPSRC (Rigorous Open Development Environment for Complex Systems).
In B, software is specified in terms of “abstract machines”. An abstract machine typically describes a number of operations along with the data encapsulated by the machine. Descriptions at this level are not generally mechanically translatable into code as they make use of the full power of mathematics. For example, given some property \( P \) which describes a variable or variable list \( x \), an operation that establishes \( P \) can be described by:

\[
\text{Op} \triangleq x : P
\]

Descriptions of operations at the abstract level may also make use of “abstract constants”, that is mathematical constants which are intended for use in specifying operations, but which are not intended to appear in an implementation.

An abstract machine consists of a number of clauses, which will typically include the following:

- a **CONSTANTS** clause, which names any abstract constants used in the machine;
- a **PROPERTIES** clause, which describes the constants in terms of constraining predicates;
- a **VARIABLES** clause, which lists the names of the machine’s variables;
- an **INVARIANT** clause, which describes the values which can be taken by the data. This includes type information for each of the variables, though types need not be stated explicitly;
- an **INITIALISATION** clause, which describes initial values to be taken by the variables;
- an **OPERATIONS** clause, which describes the machine’s operations acting on the variables.

An abstract machine has a number of associated “proof obligations”. Some examples are that the initialisation for instance must place the variables of the machine in a state that satisfies the invariant, and any invocation of an operation within its precondition (and assuming the invariant holds) must terminate in a state that also satisfies the invariant.

An abstract machine may be refined by a corresponding implementation component which includes a re-expression of the operations of the abstract machine in a representation that can be mechanically translated into an executable language.

The language used to describe operations within an abstract machine is known as the Abstract Machine Notation (AMN), and that used within the operations of an implementation as B0. They are subsets of a single language, containing constructs most appropriate for specification and implementation, respectively. For
example, AMN does not include an iteration construct or sequential composition of operations.

For purposes of formal analysis, language constructs from AMN and B0 are translated into equivalent ones given in the Generalised Substitution Language, which we also refer to as B-GSL. This is a more succinct and obviously mathematical language, which was not used as the user language of the B Method for fear of increasing customer resistance to the introduction of formal methods. The AMN/B0 statement

\[
\text{IF } x < 10 \text{ THEN } x := x + 1 \text{ END}
\]

is rendered in B-GSL as the following choice of guarded commands:

\[
x < 10 \implies x := x + 1 \land \neg x < 10 \implies \text{skip}
\]

and this construct would be analysed in terms of the wp rules for choice, guard and assignment. These are given along with rules for the other B-GSL constructs in the table in Fig. 3.1.

In this thesis we generally want the reader to be aware of the underlying predicate-transformer meaning of our examples, and we make extensive use of B-GSL constructs within specifications and programs, generally using them in preference to AMN/B0 constructs. This is so since computations written in AMN may always be translated into corresponding GSL constructs, indeed such a transformation is carried out mechanically by tools such as the B-Toolkit and Atelier-B prior to generation of proof obligations. Having performed our analysis in terms of the GSL thus moreover generalises to the case of computations expressed in AMN. However, we make an exception to this practice in Chapter 7, which is based on a case study requiring machine readable notation.

Use of \( x \) in the assignment of the above table may refer to a single variable,
but may also refer to a variable list, as in the statement:

\[ x, y := 1, 2 \]

An operation \( S \) may be characterised by two predicates: \( \text{trm}(S) \), which states the conditions under which it is guaranteed to terminate, and \( \text{prd}(S) \), which describes its effect. \( \text{trm}(S) \) is a predicate on the current state, and \( \text{prd}(S) \) is a predicate on the before and after state with dashes being used to represent after-state variables. For a substitution \( S \) acting on a state variable (or variable list) \( z \) these predicates are defined as:

\[
\begin{align*}
\text{trm}(S) & = \wp(S, \text{true}) \quad (3.1) \\
\text{prd}(S) & = \neg \wp(S, x \neq x') \quad (3.2)
\end{align*}
\]

Thus the requirement for termination is that the operation should be able to establish the weakest possible result, namely \( \text{true} \). The double negation in the definition of \( \text{prd} \) is necessary to cope with non-determinism. We will see an example of its use presently.

There are some departures, in this thesis, from \( \text{B} \) notation as used in the \( \text{B-Book} \).

- There is a tradition in \( \text{B} \) of requiring identifiers to be at least two characters in length, and reserving single-length identifiers for meta-variables. This convention is not enforced in some later tools, such as “\( \text{B4Free} \)”, and we only employ it in this thesis if an associated tool requires us to do so.

- We use \( \wp(S, Q) \) for the weakest precondition that \( S \) will establish \( Q \). In the \( \text{B-Book} \) the notation \( [S]Q \) is used.

- We write equals between predicates to indicate that they are equivalent at the top level (i.e. in the absence of any supporting hypotheses). We also employ two sizes of equal sign, = and \( = \), with identical meaning but the larger having lower binding power. This technique is borrowed from Hehner [Heh93].

- In some cases we allow the values of expressions to be bunches. In short bunches represent collections of objects just as sets, however without their structuring ability. To explain the details of bunch theory in general, and our particular use of it in this work, a comprehensive account will follow in subsequent Chapters 4 and 5. Bunches will not in any sense replace set theory, on which \( \text{B} \) is largely based, but will be used in rules that formulate the possible results of a computation.
• We introduce the idea of “frame extension”. Operations in B have definitions of the form $Op \triangleq S$ where $S$ is a generalised substitution. Operations are always defined in the context of a machine with a certain variable list, and this context is sometimes required to complete the meaning of the operation. For example, the meaning of an operation $Op \triangleq \text{skip}$ taken from a machine with state variable $x$ is given by the substitution $x := x$. It is often convenient, during general discussions, to ignore the distinction between operations and substitutions, or to introduce operations without having to define an associated machine. To this end we will use a frame extension notation to provide any additional context information required. We write $S_w$ for the substitution $S$ with its frame extended by the variables of $w$. Thus $\text{skip}_x$ would be equivalent to $x := x$.

We terminate this short review of the B Method with a note on the architecture of a B development. B developments are usually not monolithic, but rather consists of a number of machines which stand in particular relationships. In this thesis we use just three of these relationships: 
- **REFINES**
- **SEES**
- **IMPORTS**

To claim that an implementation component $M_I$ refines an abstract machine $M_A$ we state “$\text{REFINES } M_A$” within the description of the machine $M_I$. This implies that $M_I$ defines operations that have the same names and signatures as those of $M_A$, and whose behaviour, as observed through these operations, would not cause a user to suppose he was not observing the operations as specified in $M_A$. Note that this allows for a different representation of internal data, and also for a reduction of non-determinism, so long as each operation remains feasible.\(^1\) A machine that **SEES** another machine may refer to the constants used in that machine. Finally, the operations of an implementation $M_I$ may invoke those specified in a machine $MUtil_A$ say by stating “$\text{IMPORTS } MUtil_A$”. Here $MUtil_A$ is an abstract machine specification that would in turn be refined by an implementation (say $MUtil_I$). However, reasoning about the effects of the operations of $MUtil$, as they are used in $M_I$, is done totally in terms of the specification of these operations, and does not have access to any of the implementation details that may have been added in $MUtil_I$.

### 3.3 Reversible Computing, Backtracking and B-GSL

Our aim is to exploit reversible computation to introduce automatic backtracking and more abstract (mathematical) data types into an implementation-level

\(^1\)As we will see, our approach to non-determinism will be a little more complex; in addition to the traditional use of non-determinism we also use it to represent provision choice within a backtracking context, and this will affect the way in which non-determinism may be reduced during refinement.
language. Rather than adding additional state to provide reversibility, as described in the previous chapter, the formal technique we employ is to use non-deterministic choice as provision switch choice instead of (or as well as) implementor’s choice. Though not often discussed, this idea has a long history. As far back as 1967, in his paper “Non-deterministic Algorithms” [Flo67], Floyd talked of ‘programs governed in part…by final causes for the sake of which their effects are carried out.’ In “The Specification Statement” [Mor88] Morgan mentions the possibility as follows: “Ordinarily we limit the syntax of our programming language so that miracles cannot be written in it. If we relax this restriction, allowing naked guarded commands, then operational reasoning suggests a backtracking implementation.” He gives as an example this refinement of the program

\[ i : [a[i] = v] \]

(given as a specification statement) which finds one position of an element in an array.

\[
\text{if } i := 0 [] \ldots [] i := N - 1; a[i] = v \implies \text{skip fi}
\]

He comments: “We are using the generalised if…fi which allows abortion if its body if miraculous, and the body is miraculous only when no branch of the alternative can avoid the miraculous behaviour to follow. In this context if…fi resembles the “cut” of Prolog, allowing failure (preventing backtracking) if no solution is found.”

A similar possibility is noted by Hehner in [Heh93], although, as he remarks, his timing calculus does not work in conjunction with a backtracking interpretation.

To begin with, we will introduce a limited form of backtracking simply by allowing naked guarded commands of the form \( P \implies S \). Note that the use of ‘\( \implies \)’ for guards is borrowed from B and preferred over the normal ‘\( \rightarrow \)’ because we will presently be using the latter to represent a different form of guard. This requires the repeal of Dijkstra’s “law of the excluded miracle”.

As a simple example of the backtracking effect obtained with guards and choice consider

\[ S \equiv x := 1 [] x := 2 ; x = 2 \implies \text{skip} \]

An operational interpretation of this program is that it first makes a choice of assigning either 1 or 2 to \( x \). If it assigns 1 the following statement is infeasible, which provokes reverse execution. The second alternative is then tried, making \( x \) equal to 2. The following command is now feasible and the program terminates with \( x = 2 \). Alternatively, if the first command initially assigns \( x \) to 2 the second command is immediately feasible and again the program terminates with \( x = 2 \).

The formal demonstration of this may be performed by the calculation of \( \text{prd}(S) \), which we show to be \( x' = 2 \). Note that we trivially assume that for this operation
\[ \text{trm}(S) = \text{true}. \]

\[ \text{prd}(S) \]

\[ = \text{"Defn of } S\text{"} \]

\[ \text{prd}(x := 1 \; ; \; x := 2 \; ; \; x = 2 \implies \text{skip}) \]

\[ = \text{"Defn of prd"} \]

\[ \neg \text{wp}(x := 1 \; ; \; x := 2 \; ; \; x = 2 \implies \text{skip}, x \neq x') \]

\[ = \text{"wp rule for Sequential Composition (see Fig. 3.1)"} \]

\[ \neg \text{wp}(x := 1 \; ; \; x := 2, \text{wp}(x = 2 \implies \text{skip}, x \neq x')) \]

\[ = \text{"wp rule for Guard (see Fig. 3.1)"} \]

\[ \neg \text{wp}(x := 1 \; ; \; x := 2, x = 2 \implies \text{wp}(\text{skip}, x \neq x')) \]

\[ = \text{"wp rule for Skip (see Fig. 3.1)"} \]

\[ \neg (\text{wp}(x := 1, x = 2 \implies x \neq x') \land \text{wp}(x := 2, x = 2 \implies x \neq x')) \]

\[ = \text{"Logic (de Morgan)"} \]

\[ \neg \text{wp}(x := 1, x = 2 \implies x \neq x') \lor \neg \text{wp}(x := 2, x = 2 \implies x \neq x') \]

\[ = \text{"wp rule for Assignment (see Fig. 3.1)"} \]

\[ \neg (x = 2 \implies x \neq x')[x\backslash 1] \lor \neg (x = 2 \implies x \neq x')[x\backslash 2] \]

\[ = \text{"Substitution"} \]

\[ \neg (1 = 2 \implies 1 \neq x') \lor \neg (2 = 2 \implies 2 \neq x') \]

\[ = \text{"Logic"} \]

\[ \neg \text{false} \implies 1 \neq x' \lor \neg \text{true} \implies 2 \neq x' \]

\[ = \text{"Logic"} \]

\[ \neg \text{true} \lor \neg 2 \neq x' \]

\[ = \text{"Logic"} \]

\[ x' = 2 \quad \square. \]

A more extensive example of the use of this form of backtracking is found in our treatment of the Knight’s Tour problem in Chapter 7, which we develop later in this thesis using a version of Atelier B modified to accept naked guarded commands. The specification of the problem obtains the solution in a single choice, which obtains a sequence of moves satisfying the requirements of the problem. The implementation uses a loop which finds the solution step by step, and our refinement proof ensures the result meets the specification. Search heuristics may be introduced, such as making the most constrained choice first. These make use of our knowledge of the order in which non-deterministic choices are taken in our
implementation. This knowledge is not recorded in our semantics of choice, so any performance gains are outside the scope of our formal analysis.

3.4 Conclusions

In this section we have introduced the B method and notation, and outlined some variations that will be used in our approach to using B to describe reversible computations. Most significant of these is the use of non-deterministic choice as provisional choice, with an associated backtracking interpretation. Whilst backtracking need not be implemented in terms of reversibility (and usually is not) it is one way in which reversible computations can be exploited. Another is for garbage collection, allowing us to make efficient use of reference semantics when incorporating more abstract (mathematical) data types.
Chapter 4

Expression Transformers

4.1 Introduction

A shortcoming of the backtracking technique introduced in the previous section is that it is limited to finding a single solution, and that backtracking through reversibility completely erases any information found. We may wish to find and record all solutions to a problem, or a set of solutions that collectively satisfy some criteria. This will be the focus of our investigation in the current chapter, and we will attempt to formulate a solution in terms of generalised substitutions.

This satisfies objective 4 stated in Section 1.2 of showing how we can extend the idea of backtracking search by means of a prospective-value formalism, allowing a program that can find one result of a search to be encapsulated in an expression representing all results of such a search. The importance of this work within the wider context of our research into supporting the formal development of programs that can effectively exploit a reversible execution environment is that, via the formalism we propose and develop in this and the following two chapters, we extend the expressive capabilities of the implementation language to overcome the limitation of reversibility vs general, meaning non-oblivious backtracking. That such a formalism can be developed with similar expressive power to the wp calculus is moreover one hypothesis of this PhD investigation.

4.2 A First Look at Bunch Theory

For the purposes of our theory presentation it will be convenient to use “bunches” [Heh81, Heh93]. A bunch is the “contents of a set” (Hehner) without the packaging that allows set representation to build up nested structures. A bunch of bunches is self-flattening, and that property will simplify our presentation. Any single value of a type is an elementary bunch, or element. For example, 2 is a bunch. In set theory we must distinguish between 2 and \{2\}, i.e. between an element and a set containing just that element. In bunch theory there is no distinction between an element, and the bunch that consists of that element. The
empty bunch is written as null. The bunch comprehension \( \{ x \mid E \} \) is the bunch of all values taken by \( E \) as \( x \) ranges over the values of its type, where its type must be deducible from type analysis of \( E \). If \( A \) and \( B \) are bunches then their union and intersection, written as \( A \cup B \) and \( A \cap B \) respectively, are also bunches. We write \( A : B \) to say \( A \) is a subbunch of \( B \). As with sets, the repetition and order of elements has no significance. A more extensive account of bunches will be provided in the next chapter.

### 4.3 Intuitive Definition of Prospective-value Expression Transformers

Abrial’s Generalised Substitution Language [Abr96b] uses the mechanism of syntactic substitution to describe updates to a system’s state in terms of predicate transformers. Thus the weakest precondition for \( x := F \) to guarantee a post-condition \( Q \), written \([x := F]Q\), is obtained by substituting \( F \) for each free occurrence of \( x \) in \( Q \). For example

\[
[x := x + 1] x < 10 \Rightarrow x + 1 < 10 \Rightarrow x < 9
\]

interprets as: to guarantee that \( x := x + 1 \) will deliver a state in which \( x < 10 \), we must start from a state in which \( x < 9 \). This form of substitution in a predicate converts a post-condition to a precondition: one might say it works “backwards” in the sense that it converts a predicate to be guaranteed after carrying out the computation into a predicate that has to hold before.

We can apply the same mechanical substitution in expressions, but what would that mean? Consider

\[
[x := x + 1] x + 10 \Rightarrow x + 1 + 10 \Rightarrow x + 11
\]

with the interpretation “the value of expression \( x + 10 \) after executing \( x := x + 1 \) would be \( x + 11 \).” It appears that applying substitutions to expressions might form a basis for describing the effects computations would have were they to be carried out. Substitution in an expression converts it into another expression representing the value the original expression would have after the substitution. One might say it works in a forward direction in the sense that it converts an expression on the before state into the value the expression would have after performing the computation.

The GSL generalises the idea of predicate transformer to cover all the syntactic constructs of an abstract command language (preconditions, guards, choice, sequential composition and local variables). We will do the same with expression transformers, though with a slightly different range of constructs as given in Fig. 4.1, to arrive at the definition of \( S \circ E \): an expression on the current state space which gives the prospective values that could be taken by \( E \) after the
execution of $S$. This newly introduced construct we call the prospective-value expression transformer, or for short pv expression transformer.

We will see later that a prospective-value semantics is equivalent in power to that of predicate transformers, and that they can form an alternative to predicate transformers as a basis for formal analysis. In this chapter, however, we are interested in using them in quite a different way.

We propose an extended expression syntax for our implementation-level language which contains terms of the form $S \diamond E$. The operational interpretation of such a term is that the program $S$ is executed, and the subsequent value of $E$ is recorded. The execution then reverses, and if an unexplored choice is found, forward execution will recommence and another result is recorded. After all possible execution paths have been explored, execution finally reverses to the start of $S$, at which point the system state is restored.

$S \diamond E$ is the bunch of all possible values that could be taken by $E$ after the execution of $S$. We do not, however, support the use of bunches on our execution platform, the Reversible Virtual Machine. This was a deliberate design decision simplifying the implementation of the RVM, and allowing us to retain a higher level of run-time efficiency. Instead we provide a fully general implementation of finite sets, and, when used in our implementation language$^1$ we require terms which denote non-elementary bunches to be enclosed within set brackets. For example $x := 1 \parallel x := 2 \diamond x$ is not suitable as a general term for use in our language as it evaluates to the non-elementary bunch $1, 2$. It may however occur within set brackets, e.g. $\{x := 1 \parallel x := 2 \diamond x\}$, and this evaluates to $\{1, 2\}$.

Generally, a term of the form $S \diamond E$ can be rewritten as a term on the current state. The rules for doing this for each GSL construct are given in Fig. 4.1$^2$.

To illustrate the use of these rules let us consider the possible values taken by the expression $2 \ast x$ after running the program:

$$x := 1 \parallel x := 2 ; x := x + 1$$

Intuitively we can see that these values will be 4 and 6. We can derive this result formally as follows:

$$x := 1 \parallel x := 2 ; x := x + 1 \diamond 2 \ast x$$

$\Rightarrow$ “Sequential Composition”

$$x := 1 \parallel x := 2 \diamond x := x + 1 \diamond 2 \ast x$$

---

$^1$We admit here that our “implementation language” does not yet exist in a fully defined form. However, the features we discuss are all directly implemented in the intermediate stack-based language of our Reversible Virtual Machine.

$^2$We do not actually give a rule for the pre-conditioned substitution. For the moment we assume that when used within an extended expression language, we protect ourselves from using an operation outside its precondition by discharging the relevant proof obligations. In subsequent chapters, however, we will take a more general view which treats preconditions explicitly on the semantics level.
Figure 4.1: Prospective-value expression transformer rules for rewriting $S \diamond E$.

```
= "Assignment"
  x := 1 [] x := 2 * (x + 1)

= "Choice"
  (x := 1 * 2 * (x + 1)), (x := 2 * 2 * (x + 1))

= "Assignment"
  (2 * (1 + 1)), (2 * (2 + 1))

= "Arithmetic"
  4, 6

We can derive additional properties for various forms of substitution. For example:

**Proposition 1.** $S ; x := E \diamond x = S \diamond E$

**Proof.** $S ; x := E \diamond x$

= "Sequential Composition"
  $S \diamond x := E \diamond x$

= "Assignment"
  $S \diamond E \square$.

Note that our extended expressions include terms which are not directly subject to the law of referential transparency: we cannot infer from $E \equiv F$ that $S \diamond E \equiv S \diamond F$. For example $x \equiv y$ does not imply that $x := 1 \diamond x := x := 1 \diamond y$. We have a similar situation with predicate transformers, where $Q \equiv R$ does not allow us to infer $[S]Q \equiv [S]R$. We must eliminate terms of the form $S \diamond E$ before applying referential transparency in a proof, and the table of rules given above enables us to do so.
4.3.1 Generalised Assignment

An interesting case of assignment arises when the left hand of the assignment statement is undefined. Consider for example

\[ \overline{S} \triangleq x := \frac{1}{0} \]

Our interpretation of \( \frac{1}{0} \) is that it is equal to the “improper” bunch \( \perp \) which will be introduced in the following chapter 5, and which generally is used to represent the outcome of a non-terminating computation. In our current exposition we require the left hand of all assignments to be elementary (and thus proper) values, and we would protect ourselves from evaluating something like \( \overline{S} \circ E \) by suitable proof obligations. Nevertheless it should be noted that indeed our formalism of prospective-value expression transformers, which we will fully develop in the following two chapters, is powerful enough to give a meaning to \( \overline{S} \circ x \). We do so by relaxing the constraint on assignment allowing the assigned expression to be any arbitrary bunch. Accordingly we may define a more general form of assignment in terms of the elementary assignment introduced earlier:

**Definition 2.** Let \( x \) be a variable and \( E \) an arbitrary bunch expression of appropriate type. Generalised assignment is defined through

\[ x := E \triangleq \text{df} \quad E \neq \perp \mid @x' \cdot x' : E \implies x := x' \]

Here \( x' \) quantifies over all elementary values of \( E \), this will be further clarified and explained in Section 5.3.6. The previous definition then allows us to formally conclude that \( x := \frac{1}{0} \) is equal to `abort`, a luxury which is not within the semantic scope of conventional B.

Note that Def. 2 is not utilised or referred to at any other place in the thesis, it is merely an interesting possibility and feature which we might further investigate and exploit in follow-up publications, and which the PhD viva has given rise to incorporate in the thesis.

4.4 Applications in Programming

4.4.1 Finding all Results of a Search

Consider the following program which finds an occurrence of \( a \) in the sequence \( \text{seq} \) and leaves its position in the variable \( i \).

\[ Op \triangleq a \in \text{ran}(\text{seq}) \mid i \in \text{dom}(\text{seq}) ; \text{seq}(i) = a \implies \text{skip} \]

This is a backtracking program which works by choosing an index value and seeing if the element at that position has the required value. If so, the program will terminate, otherwise it will reverse and make another choice.
We can use such a program to find all positions at which \( a \) occurs in \( \text{seq} \), by using the extended expression:

\[
\{ \text{Op} \circ i \}
\]

where \( \text{Op} \) must be replaced by the body of its definition before applying the expression transformer rules. Note that for such a use we can, indeed, drop the precondition of \( \text{Op} \). This precondition has the rôle (in this particular case) of ensuring that \( \text{Op} \) is feasible, but we do not need this property for use of \( \text{Op} \) within an expression transformer: if \( a \) does not occur in \( \text{seq} \) then \( \{ \text{Op} \circ i \} \) will evaluate to an empty set.

### 4.4.2 Evaluating Set Expressions, Feasibility and Quantifications

We present some executable constructs which illustrate the expressive capabilities of a prospective value formalism. First the prospective-value expression transformer evaluations of set union, intersection, and set difference. We assume the existence of a variable \( e \) of suitable type, and \( A \) and \( B \) are finite sets represented in the state space of our machine:

\[
A \cup B = \{(e \in A) \land (e \in B) \circ e\}
\]

\[
A \cap B = \{e :\in A; e \in B \Rightarrow \text{skip} \circ e\}
\]

\[
A \setminus B = \{e :\in A; e \notin B \Rightarrow \text{skip} \circ e\}
\]

We can evaluate whether an operation is feasible:

\[
\text{fis}(S) = \{S \circ x\} \neq \emptyset \quad \text{where}\; x \text{ refers to the variable(s) assigned by } S.
\]

The set operations are reasonably efficient, but the expression for \( \text{fis}(S) \) evaluates all execution paths of \( S \) whereas it only needs to find one such path to know \( S \) is feasible.\(^3\)

We can furthermore evaluate quantifications as binary expressions:

\[
(\exists x \bullet x \in A \land P) = \text{fis}(x :\in A; P \Rightarrow \text{skip})
\]

\[
(\forall x \bullet x \in A \Rightarrow P) = \text{card } (A) = \text{card } (\{x :\in A; P \Rightarrow \text{skip} \circ x\})
\]

\(^3\)In fact all these expressions are implemented directly as primitives of our virtual machine, as described in our previous papers [Sto00, SZ02].
A Functional Language which Incorporates State-Changing Operations

In B we distinguish constants from operations. A function which returns the greatest common divisor of two numbers would be declared in the \texttt{PROPERTIES} clause of a machine and could be described as:

\begin{verbatim}
gcd \in \mathbb{N}_1 \to \mathbb{N}_1 \land
\begin{align*}
gcd(x, y) &= \\
& \text{if } x = y \text{ then } x \\
& \text{else } \\
& \quad \text{if } x > y \text{ then } gcd(x - y, y) \\
& \quad \text{else } gcd(x, y - x) \\
& \text{end} \\
& \text{end}
\end{align*}
\end{verbatim}

A further distinction is made between abstract constants, which provide us with power of mathematical description, and concrete constants, which are implementable. In the B-Book, and in early tool support for the B Method, the allowable range of concrete constants has been limited, but without impinging on the B Method we can assume that concrete constants could be any that are essentially implementable, and we can certainly admit functions such as \texttt{gcd} within this category.

One advantage of doing so is to allow us to make use of recursive definitions. Recursion is not generally available within operations in B since we interpret the invocation of an operation in terms of syntactic replacement. An operation with output parameter \(a\) and input parameter \(b\) has a definition of the form:

\[ a \leftarrow Op(b) \equiv S \]

When \(Op\) is subsequently invoked as say \(c \leftarrow Op(E)\) its effect can be described by the simple syntactic replacement \(S[a, b \backslash c, E]\), that is by rewriting \(S\) with \(a\) replaced by \(c\), and \(b\) replaced by \(E\). Thus \(a \leftarrow Op(b) \equiv a := b + 1\) invoked as \(c \leftarrow Op(E)\) becomes \(a := b + 1[a, b \backslash c, E] = c := E + 1\).

Recursively defined operations cannot be handled in this way, and are not supported in the classical B development tools, although Abrial and Lafitte have developed a treatment of recursion in implementations based on set transformers, which is presented in Chapter 12.5 of the B-Book \cite{Abr96b}. Recursively defined constants have no such problem, but cannot refer to state variables. We will look at the use of recursive definitions which refer to state and to state-changing operations by using terms of the form \(S \odot E\). Our aim is to show how we might
be able to use a functional style of programming whilst incorporating state and making use of the operations of a state-based language.

Consider a definition:

\[ g(x) \triangleq \text{gcd}(x, y) \]

where \( y \) is an integer variable. Here we have a definition which refers to a variable and which uses the recursive definition of \( \text{gcd} \). Note that we cannot consider \( g \) as a constant since it refers to the variable \( y \). However, B machines provide a \textbf{DEFINITIONS} clause where such a construct can be defined. In classical B the meaning of such a definition is given by textual replacement - wherever \( g(a) \) appears, it is replaced by \( \text{gcd}(a, y) \). However, this will not work for a recursive definition. We will see in the next section how useful such a recursive definition can be. For the moment we are just asking how it can be analysed in terms of \( \text{pv} \) expression transformers. With this in mind we propose an equivalent definition to that of \( g \). We call this new definition \( g_1 \):

\[
\begin{align*}
g_1(x) & \triangleq \\
& \text{if } x = y \text{ then } x \\
& \text{else} \\
& \quad \text{if } x > y \text{ then } g_1(x - y) \\
& \quad \text{else } y := y - x \odot g_1(x) \\
& \text{end}
\end{align*}
\]

Here we remind the reader that the assignment \( y := y - x \), as used here, will not result in an invocation of \( g_1 \) permanently changing the state of \( y \), since any change will be restored by reverse execution. Our interpretation of the term \( y := y - x \odot g_1(x) \) is that it represents the value \( g_1(x) \) would have were it to be executed after performing the assignment \( y := y - x \).

Thus we have a way of referring to a state variable within a recursive definition, and we can use this as an alternative to passing the value of that variable as a parameter.

We do, however, have a mechanical problem in applying our expression transformer rules to analyse such a definition: when we replace the definition by its body we do not eliminate it, since the definition is recursive. We therefore need to find some other way to make the \( y \) within the definition of \( g_1 \) visible to our expression transformer rules.

Fortunately we can do this by means of lambda abstraction. We replace the \( g_1(x) \) that occurs in \( y := y - x \odot g_1(x) \) by \((\lambda y \cdot g_1(x))(y)\), which allows us to
eliminate the ◦-operator from our definition:

\[ y := y - x \diamond g_1(x) = y := y - x \diamond (\lambda y \cdot g_1(x))(y) = (\lambda y \cdot g_1(x))(y - x) \]

### 4.4.4 A Mini-max Algorithm

As an example of how useful a recursive style of definitions can be when combined with pv expression transformers, we present a mini-max algorithm to further illustrate our approach to using state in a functional setting. We can make use of state, as when updating the position of the game after a move, but the expressions that perform such state updates are evaluated without ultimate side effects, since any changes of state are subsequently restored by reverse execution.

We remind the reader that the mini-max algorithm for two player games requires that there should be some heuristic to “score” any game position. We will consider that that a high score indicates an advantage to player A and a low score an advantage for player B.

We assume functions \texttt{max} and \texttt{min} which find the maximum and minimum of a non-empty set of integers. We assume variables which represent the state of the game, a numerical expression \texttt{score} which implements the heuristic, and operations \texttt{AMOVE} and \texttt{BMOVE} which non-deterministically choose and play a move for players A and B respectively. We code the algorithm from the point of view of player A, assuming that A has just made a provisional move and wishes to evaluate the resulting position.

\[
\text{eval}(n) \triangleq \\
\text{if } n = 0 \text{ then } \text{score} \\
\text{else } \\
\min \{ \\
\text{BMOVE} \diamond \text{max} \{ \text{AMOVE} \diamond \text{eval}(n - 1) \} \\
\}
\]

Now we can use \texttt{eval(n)} to provide the mini-max score for the current board position based on a 2*n move lookahead. As in pure functional programming it is an expression which represents a value. Unlike pure functional programming \textit{we can carry state with us}. Some examples: neither \texttt{eval} nor \texttt{score} need to be passed the state of the game, since they can access its representation in state variables; likewise \texttt{AMOVE} and \texttt{BMOVE} have their effect by changing the current state. There is a guarantee of “no side effects” even though we are able to change the current state of the game during our mini-max search.
4.5 Conclusions

In this chapter we have considered how to extend the backtracking interpretation of GSL, introducing the formalism of prospective-value expression transformers to describe all the results of a search. We have noted that the constructs described are implemented in our execution platform, the Reversible Virtual Machine. We have described some possible applications, including the representation of some set expressions and the incorporation of expression transformers within recursive definitions.
Chapter 5

Improper Bunch Theory

5.1 Introduction

In this chapter of the thesis we present a theory which is an extension of ‘bunch theory’ originally proposed by Hehner in [Heh81, Heh93], and which we call Improper Bunch Theory. Hereby we satisfy objective 3 of the investigation stated in Section 1.2 of introducing and axiomatising a variant of Hehner’s bunch theory to simplify the presentation of our theories, and effectively work towards a formal presentation of prospective-value semantics, its feasibility being one of the hypothesis of the PhD.

It was decided to devote an entire chapter of the thesis to the topic of bunches since during the course of investigation for this PhD they became a major part of the contribution. In particular, bunches enable us to give a more concise formulation of prospective-value (pv) semantics which is subsequently developed in Chapter 6. Initially we confined ourselves to a set-based approach here which suffered from a slight inhomogeneity arising from the pv simplification (or rewrite) law for sequential composition. The application of bunch theory nicely eliminated this problem. Secondly we discovered that bunches were interesting and worthwhile objects of study in their own right which haven’t been extensively investigated and used yet as a formal theory in computer science. This might render this chapter moreover a valuable contribution beyond the scope of exploiting bunches for our particular purpose in this PhD, which is to describe the prospective value of reversible computations.

When first utilising bunches to characterise computations in terms of how they change the value of an expression after execution as illustrated in Chapter 4 (remember that we call such values “prospective values”), we noticed that bunch theory in its original form wasn’t expressive enough to describe the prospective value of all computations within the refinement lattice of generalised substitutions. This in particular included those which exhibit potential non-termination as part of their behaviour, namely pre-conditioned substitutions. The first objective of Improper Bunch Theory was hence to accommodate the outcome of non-terminating computations on the level of (bunch) expressions; such is also
reflected by the prefix ‘Improper’ being not so much a confession that our bunch theory is not ‘proper’ in some way, but referring to a certain augmentation we perform in order to deal with the non-terminating case of computation, and consequently render the theory of prospective values as discriminating on a semantic level as the one of weakest preconditions. A further objective that guided our endeavours was to develop a variant of bunch theory which would lend itself more easily for prospective integration into the B Method and tools framework. A comprehensive integration examining all practical consequences, including the customisation of existing or development of new tools is beyond the scope of this chapter and ambition of the PhD, however with the knowledge presented here we hope to lay some of the foundations for such a project, and in the least sensitise the reader for its possible challenges and emerging difficulties.

Last but not least we introduce some useful extensions in Improper Bunch Theory not within the classical treatment. Such are for example a more general notion of comprehension along with the the ‘guarded’ and the ‘pre-conditioned’ bunch. They not only simplify the presentation of pv semantics and its associated axioms, but also yield a more tractable formalism supported by numerous propositions and algebraic laws facilitating reasoning and proof in bunch theory. This is nicely exemplified in Section 6.4.1 where we conduct a series of non-trivial proofs involving bunch expression, and draw heavily on the mathematical properties of improper bunches.

An overview of the chapter’s content is as follows. In Section 5.2 we conduct a historical review of some relevant publications on bunch theory in general. In Section 5.3 we present and review those aspects of classical (meaning Hehner’s) bunch theory which principally carry over in our own exposition. This means that the adaptions performed in this section are not of primary significance, but nevertheless important in the context of our theory. The main contribution follows in Section 5.4 introducing improper bunches and those features singular to Improper Bunch Theory which make our bunch theory ultimately useful in the context of prospective value semantics. Section 5.5 provides a collection of useful laws for reasoning about bunches expressions. We finally draw our conclusions in Section 5.7.

Note Most of the content and ideas in this chapter have been reviewed and published in [SZ03, ZSD05, Sto06].

5.2 Historical Review

In this section we shall give a brief overview of relevant publications in the area of bunch theory and its applications.

The first formal presentation of bunch theory was given by Hehner [Heh81]. He proposed bunch theory as a light-weight alternative to set theory that is more adequate in the context of computer science. As he pointed out, we often don’t
require the entire power and flexibility of set theory, but instead invent pseudo notations in cases where a ‘weaker’ set theory would likely serve its propose better. Another argument given by Hehner was that sets of uncountable cardinality are rarely needed in computer science, this is also reflected in the theory of B in which all base types consisting of infinitely many elements are countable.

In this first publication Hehner presented the idea of bunches, as well as compared a simple axiomatisation of bunch theory to some classical set theory axioms. Although this paper already contained the fundamental notion of a bunch and some associated operators, a more comprehensive exposition of bunch theory followed later in his book “A practical theory of programming”[Heh93, Heh07] including a comprehensive list of operators, their axiomatic definitions, and associated laws to support reasoning and proof within bunch theory. This publication is also the main source upon we construct our own theory of bunches.

Further notable work on bunches has been carried out by Morris and Bunkenburg [MB01]. Their motivation was to develop a bunch theory which incorporates predicates and functions being bunches while preserving most laws from classical logic, and moreover treating functions defined through $\lambda$-abstractions as elementary objects; this makes the theory especially useful for application in the context of functional programming.

A possible omission in Hehner’s work was to provide a model for bunch theory, and thereby construct a proof that bunch theory is consistent with respect to the underlying model. Hehner would argued that bunch theory doesn’t require a model because it is no less fundamental than set theory is. It is hence not surprising that in [Heh93] set theory is introduced and defined a posteriori in terms of bunch theory. In Morris and Bunkenburg’s work, however, complications arise since their approach requires the introduction of non-flat function types, as well as 4-valued logic resulting from predicates being Boolean bunches. Such could clearly be one reason why defining a denotational model for bunches is a desired verification step as there seems to be a potential risk of inconsistencies, this is hence what they do in [MB01]. In their paper [MB02] they investigate one particular inconsistency which can enter theories of non-deterministic functions. Interestingly they point out that this inconsistency is the more treacherous being all but obvious at first glance. Although in this publication there is no particular reference to bunches, what they call a ‘non-deterministic expression’ is conceptually very similar to a bunch.

## 5.3 Our Adaptation of Classical Bunch Theory

Bunch theory was invented by Hehner as a simplification of set theory [Heh81]. Hehner pointed out that when reasoning about a collection of objects in the context of computer science, we often don’t require set theory in its entire power.

---

1To be more accurate they don’t distinguish between predicates and Boolean expressions as we do in the theory of B.
and generality. In particular the structuring ability to form nested sets such as \(\{1,\{2\}\}\) is in many cases irrelevant to the cause. Thus he proposed a ‘weaker, and notationally more convenient’ set theory he called bunch theory having ‘just the right power’ for its application. Prospective-value semantics revealed itself to be one of those applications that considerably benefited from the use of bunches in terms of theory presentation which made bunches a major subject of this PhD investigation.

As with sets, bunches record collections of objects disregarding of repetition and order, i.e. all that we can know about a bunch is whether it includes a given element, or not. Unlike sets the elements of a bunch are not packaged, and as a consequence bunches always remain flat in structure. Syntactically this shows in omitting encapsulating parentheses when enumerating the elements of a bunch. For example, \(1, 2, 3\) is the bunch consisting of the first three natural numbers. Notice the linguistic difference between a bunch consisting of its elements, and a set containing them. Ordinary single values such as \(1\) or \(2\) are also bunches — indeed singleton bunches consisting of only one element. Such bunches we call elementary, or just elements, and no distinction is to be made between elementary bunches and the element they consist of. A special bunch is the bunch \(\text{null}\) which doesn’t include any elements, and hence is called the empty bunch\(^2\).

Importantly by thus extending our language of mathematical expressions to incorporate bunches we don’t abandon sets as such; both bunches and sets coexist within the same mathematical framework. Moreover we are free to construct bunches of sets such as \(\emptyset, \{1\}, \{1, 2\}\). On the other hand encapsulating a bunch into curly set brackets yields the set containing exactly the elements the bunch consists of, this operation is known as the packaging of a bunch.

At this point the reader may wonder whether there could be a syntactic ambiguity between packaging of the bunch \(1, 2, 3\) and the set enumeration \(\{1, 2, 3\}\). In fact there is none in Hehner’s theory of programming simply because no operator for set enumeration is defined: the only way to specify a set in terms enumerating its elements is by appealing to bunch enumeration and packaging. In the theory of \(B\), however, this is clearly not so. Indeed this specific ambiguity is not harmful since it doesn’t produce any ambiguity in value. In other situations however potential issues may arise from different uses of comma, for example should \((1, 2)\) be interpreted as the tuple \(1 \mapsto 2\) (being an element), or the parenthesised (non-elementary) bunch \(1, 2\)? We resolve the issue by using alternative notations where ambiguities may arise, for example utilise maplet notation \(\mapsto\) throughout for pairs. In this way we are able to reserve comma for use in bunch construction, or uses where no interference is possible as in separating the variables of a quantification or simultaneous substitution. Parentheses on the other hand are exclusively used for delineating expressions in our syntax. In certain cases the constraint of well-formedness of expressions with regards to typing is

\(^2\)Strictly speaking there is an empty bunch \(\text{null}_T\) for each type \(T\). Throughout the presentation we however take the liberty of not making the type of \(\text{null}\) explicit whereby assuming it can be inferred from the context via type analysis if necessary.
assumed to provide sufficient information for disambiguation, e.g. in the simultaneous substitution $E[x, y\backslash x', y']$ clearly $x', y'$ cannot be a bunch for the overall expression to be well-typed.

In the following subsections we will explain how we adapt “classical” bunch theory, meaning the bunch theory published by Hehner in [Heh93], to make it fit our purposes which is within a strongly typed and multi-sorted environment.

5.3.1 Types

Unlike in Hehner’s theory which has implicitly typed bunches our bunches are strongly typed, therefore all elements of any given bunch must belong to the same a priori declared type $T$. As is the case in B, types are maximal sets in our universe, such as $\mathbb{N} = \{0, 1, 2, \ldots\}$ or $\text{BOOL} = \{\text{TRUE}, \text{FALSE}\}$.

A more uniform approach here may have been to associate types with bunches instead. An argument for this is that in Hehner’s bunch theory the concept of a bunch is more fundamental than that of a set. The reason for keeping sets as types is not to deviate unnecessarily from the theory of B which considers types to be (maximal) sets, thereby do justice to the set goal of developing a variant of bunch theory which lends itself readily for future integration into the B Method. The idea of bunches being types may be pursued by the author in further publications on the subject.

Nevertheless in some cases we might want to refer to the bunch consisting of all elements of a certain type $T$, this bunch we call the carrier bunch of $T$ and us the syntax $T_b$.

5.3.2 Inclusion and Equality

For bunch inclusion the notation $E : F$ is used stating that every element of the bunch $E$ also belongs to the bunch $F$. Alternatively we say that $E$ is a subbunch of $F$. The extensionality axiom for bunch inclusion, which is not explicitly included in Hehner’s presentation of bunch theory but part of ours, is given by:

**Definition 3.**

$$E : F \equiv_{df} \forall x \bullet x : E \Rightarrow x : F$$

Naturally equality is defined in terms of mutual inclusion leading to the following definition:

**Definition 4.**

$$E = F \equiv_{df} E : F \land F : E$$

Whereas bunch inclusion is somewhat similar to subset inclusion in set theory, there is no need for a designated membership operator in bunch theory; in order to say that $x$ is an element of the bunch $E$ we as well employ the notation $x : E$. 
Remark Note that Def. 3 will subsequently have to be qualified with another conjunct in order to accommodate the extensions we make to bunch theory in Section 5.4. This is somewhat cumbersome in terms of presentation, however we decided it was more important to introduce the concept of equality as soon as possible.

5.3.3 Primitive Operators

The familiar set-theory operators ‘∪’, ‘∩’ and ‘−’ moreover have their counterparts in bunch theory where they are called bunch union, bunch intersection and bunch difference respectively. Accordingly, if $E$ and $F$ are bunches, then their

- union $E, F$ is the bunch that comprises the elements of both $E$ and $F$;
- intersection $E \cap F$ is the bunch that comprises those elements which simultaneously belong to $E$ and $F$;
- difference $E \setminus F$ is the bunch that comprises those elements of $E$ which are not also elements of $F$.

Note that there is no separate construct for bunch enumeration similar to set enumeration since the syntax of an enumerated bunch is already entailed in bunch union. If we write $1, 2, 3$ we thus mean $(1, 2), 3$.

To complete the list of primitive bunch operators we mention two more, the cardinality of a bunch, and the property of a bunch being elementary. The cardinality, or number of elements, of a bunch $E$ is given by $\#E$. Thus for example $\#7 = 1$, $\# (1, 2) = 2$ and $\# \text{null} = 0$. In case of $E$ including infinitely many elements we consider $\#E$ to be undefined, therefore make no stipulations about its value\(^4\). There are two reasons for this deviation from Hehner’s theory: first in the mathematical theory of B the natural numbers don’t include the additional maximal element $\infty$ which we would require to conform with Hehner’s definition, and adding it would be a major incision to B and its tool support of the kind we set ourselves to avoid. Secondly evaluation of $\#E$ within our Reversible Virtual Machine would, if we supported this operator directly, almost certainly fail with unpredictable behaviour if $E$ encompassed infinitely many elements. Our definition already reflects this faithfully. To assert that a bunch consist of only one element and hence possesses the elementary property we write $\Delta E$.

It may be remarked that in Morris and Bunkenburg’s theory of bunches the $\Delta$ operator is used for a slightly different purpose, namely stating that some bunch conceptually consists of only one entity, and not necessarily implying that $\#E = 1$. What we (and Hehner) refer to as elementhood they call atomicity and

\(^3\)Note that Hehner doesn’t provide a bunch difference operator in [Heh93] however defining one by means of a suitable axiom such as $x : E \setminus F \equiv x : E \land \neg(x : F)$ is not a great difficulty.
\(^4\)Compare this to Hehner who defines $\# \mathbb{N}_b = \infty$ where $\infty$ extends the type of naturals with an artificial maximal element.
use a different operator $\nabla E$ for it. We won’t explore in depth here the reason for this, nor the difference between these two operators in their theory of bunches, but only inform the reader that in our work there is no need for this distinction since we abstain from introducing so-called non-flat (function) types as Morris and Bunkenburg do in their theory of bunches. In this respect we stay closer to Hehner’s bunch theory.

### 5.3.4 Lifted Operators

A secondary source for obtaining operators on bunches is through lifting. Any operator defined on elementary values can be lifted naturally to become an operator on bunches by demanding it distributes (in all its operands) through bunch union, and is strict with respect to $\text{null}^5$.

For example, the addition operator ‘$+$’ which applies to natural numbers only is lifted in the prescribed way to become an operator on bunches of naturals, so that for instance

$$(1, 2) + (3, 4) = 1 + 3, 1 + 4, 2 + 3, 2 + 4 = 4, 5, 6$$

and

$$\text{null} + 1 = \text{null} = 1 + \text{null}.$$

Here we agree on the assumption that all conventional operators which haven’t been explicitly defined on bunches are lifted in this way.

The lifting that was previously described only applies to genuine operators whose result is a value of some type. If logical operators were lifted in the same way we would necessarily leave the realm of two-valued logic, e.g. $1, 2 < 2$ would be equivalent to $\text{true}, \text{false}$.

This is a step which despite its feasibility is not anticipated in our work. First it would bring us further away from the theory of B which, of course, is founded on standard two-valued logic, secondly with introduction of the improper bunch in Section 5.4 we would be left with yet another truth value to take into account essentially forcing us to develop or adopt a 5-valued logic — a complication we decided to avoid for the purpose if this PhD investigation.

To solve the previously described problem of lifting logical operators we propose a different method of lifting for such which ensures their application always evaluates to a proper truth value i.e. $\text{true}$ or $\text{false}$. Taking, for example, set membership, $E \in F$ is true exactly when it is point-wise true for all elements of $E$ and $F$. Formally

$$E \in F \overset{\text{df}}{=} \forall x, y \bullet x : E \wedge y : F \Rightarrow x \in y.$$ 

The lifting of subset inclusion and various other operators$^6$ such as the relational

---

$^5$This criterion will further have to be refined in the light of improper bunches, see Section 5.4.1.

$^6$Note that lifting of logical operators only applies to those which aren’t explicitly defined, hence it doesn’t apply for example to bunch equality which was defined in Section 5.3.3.
ones on numbers is done exactly in the way illustrated above, i.e. point-wise by means of a universal quantification. Accordingly we have for example \((1, 2) < 3\) being true, but on the other hand \((1, 2) < (2, 3)\) being false since \(2 \neq 2\).

In contrast to lifting logical operators by demanding they have to hold true point-wise for all combinations of elements of the two bunches, we also considered an alternative method for lifting logical operators where the result of the lifted operator is true if there is at least one pair of elements taken from the operand bunches for which the operator is true if applied. This is exemplified by the following definition:

\[
E \in^* F = \text{def} \exists x, y \cdot x : E \land y : F \Rightarrow x \in y
\]

Note that whenever we want to appeal to the particular lifting of an operator in this way we used the superscript ‘\(*\)’. Thus \((1, 2) <^* 2\) and \(1 \in^* \{1\}, \{2, 3\}\) are both true.

**Remark** Indeed in our work so far we didn’t find very much use for the second approach of lifting logical operators, nevertheless we included it in this section as in some other application contexts this may be a preferred choice over the point-wise lifting of logical operators.

### 5.3.5 Packaging and Unpackaging

Two more special operators enabling us to either convert a bunch into a corresponding set, or vice versa a set into a bunch. The packaging operator \(\{\_\}\) takes a bunch \(E\), and returns the set containing exactly the elements the bunch consists of. We define it as Hehner does through the following axiom:

**Definition 5.** \(E \in \{F\} = \text{def} \ E : F\)

Here \(\in\) is lifted as described in Section 5.3.4. Note that packaging is one of the few operators that doesn’t distribute through bunch union, and further isn’t strict with respect to null. It is also unfortunately not monotonic with respect to bunch inclusion, and hence must be avoided in situation where monotonicity is an essential prerequisite as for example in fixed-pointer treatment.

The unpacking operator \(\sim\) takes a set and yields the contents of the set being the bunch of its elements. It is only applicable to sets, may be lifted in the way explained in Section 5.3.4, and trivially we postulate, as Hehner does, is the inverse operation of packaging.

**Definition 6.** \(\sim\{E\} = \text{def} \ E\)

### 5.3.6 Quantified Variables

Although in our universe values are generally those of bunch expressions, we agree that quantified variables only range over elementary values of the appropriate
type. This applies to any kind of bound variable, e.g. being part of universal or existential quantifications, lambda expressions, bunch comprehension expression, etc. If for some reason we deviate from this rule it will be explicitly stated.

In our syntax of quantification we don’t explicitly mention the type over which the quantified variable ranges explicitly, but instead assume it is always inferable from the context via type analysis. For example, it is implicitly determined that in the existential quantification

$$\exists x \cdot x < 5$$

the bound variable $x$ ranges over the elementary values of type $\mathbb{N}$. Besides we are always free to make the type explicit by writing

$$\exists x \cdot x \in \mathbb{N} \land x < 5 \quad \text{or alternatively} \quad \exists x \cdot x : \mathbb{N}_b \land x < 5$$

instead, but there is only essential need for this if the type of $x$ cannot be inferred from the quantified expression.

### 5.3.7 Axiomatisation

Our presentation of bunch theory in the previous sections has on many occasions been rather informal. This is due to space considerations — bunches are an important aspects of this investigation but not the sole focus of the PhD. Furthermore most of the axioms and laws listed in [Heh93, Heh06] remain valid in our own dispensation of bunch theory, meaning we silently inherit them. A list of them and some associated laws can be found in Appendix B of the thesis. There is no need for re-inventing the wheel entirely from scratch if what is already ‘in store’ adequately fits our purpose. For a comprehensive formal presentation of operators mentioned so far we refer to the exposition by Hehner, of course those features of bunch theory which differ in our treatment are rigorously introduced by virtue of definition and axiom.

### 5.4 Improper Bunch Theory

In this section we present those features which are characteristic to Improper Bunch Theory.

#### 5.4.1 The Improper Bunch

Conventional bunch theory isn’t expressive enough to distinctively describe the outcome of a possibly non-terminating computation, e.g. the value of $x$ after execution of the GSL program `abort`. The best we can do in such cases is to attribute the bunch $\mathbb{N}_b$ as the possible outcome, however then `abort` would be clearly indistinguishable form the non-deterministic assignment $x : \in \mathbb{N}$, at
least from a prospective-value perspective. For partial correctness this may be acceptable, indeed this is what we satisfied ourselves with in the previous chapter and [Sto02], however to reflect the complete total-correctness picture of a GSL computation in its prospective value some more work remains to be done.

For this reason we extend bunch theory here with a new bunch for each type \( T \), called its ‘improper bunch’ \( \perp_T \). In the presentation, the type subscript is often omitted as we rely on context to determine it. Importantly \( \perp_T \) is not some kind of elementary value, but a special bunch having the property of absorbing any other bunch of its type. We formalise this by means of the following two axioms:

- \( E : \perp_T \) for any bunch \( E \) of type \( T \)
- \( \neg \perp_T : T_b \) for any type \( T \)

The first axiom ensures that \( \perp \) resides at the bottom of the refinement ordering of bunches (such is defined by inverse inclusion, see the following section 5.4.2), the second that it is genuinely different from the carrier bunch of all elementary values of its type. It lies close at hand now that our subsequent goal will be to formally define prospective value semantics in such a way that \( x : \in \mathbb{N} \diamond x = \mathbb{N} \) while on the other hand \( \text{abort} \diamond x = \perp \).

From the previous axioms we can conclude that bunch union is certainly strict with respect to \( \perp \). In addition we shall also require strictness of all lifted operators with respect to \( \perp \). Recalling the definition of lifting in Section 5.3.4 should however raise doubts whether this most recent requirement on lifting does not contradict what we previously stated, i.e. that lifted operations are strict with respect to \( \text{null} \). Clearly a binary operator cannot be strict with respect to both bunches, \( \text{null} \) and \( \perp \), in both its arguments, there is a ‘battle of giants’ to be decided either in favour of \( \text{null} \) or \( \perp \). Thus we have to relax the requirement in Section 5.3.4 upon strictness by restating that lifted operators have to be strict with respect to \( \text{null} \) unless one of the operands is equal to \( \perp \), this means \( \perp \) dominates in the presence of \( \text{null} \), e.g. \( \text{null} + \perp = \perp \).

It could be remarked that the previous decision was not taken randomly: the idea behind it is that from a operational point of view evaluation of \( \perp \) potentially may fail and make continuation of the program impossible, whereas \( \text{null} \) can be evaluated without harm although this may result in reversible execution. It is beyond the scope of this PhD to investigate this related implementation issues in depth, but there seems some argument for having \( \perp \) dominate over \( \text{null} \) to reflect the operational behaviour of our reversible virtual machine implementation platform.

### 5.4.2 Refinement Lattice of Bunches

In Improper Bunch Theory the two extreme bunches \( \perp \) and \( \text{null} \) form respectively the bottom and top of the inverted inclusion ordering on the bunches of any given
CHAPTER 5. IMPROPER BUNCH THEORY

5.4.3 Extensionality

In the face of improper bunches the extensionality axiom given in Section 5.3.2 needs to be slightly modified. In Improper Bunch Theory it is not enough to demand for two bunches being equal that they have the same elements. We also have to ensure that if one of them is the improper bunch, so is the other. Note that the maximal bunch of a type $T_b$ cannot be distinguished from the improper bunch $\bot_T$ by merely inspecting the bunches in terms of the elements they included. The modified axiom for extensionality now reads is follows:

**Definition 7.**

$$E : F =_{df} (\forall x \cdot x : E \Rightarrow x : F) \land (\bot : E \Rightarrow \bot : F)$$

With regards to equality we maintain the definition already given in Section 5.3.2, i.e. two bunches equal if they mutually included each other.

5.4.4 The Guarded Bunch

The construct of a guarded bunch is defined as follows:

**Definition 8.** $P \rightarrow E = \begin{cases} E & \text{if } P \text{ is true} \\ \text{null} & \text{otherwise} \end{cases}$

Trivially $P \rightarrow E$ evaluates to the value of the expression $E$ wherever $P$ holds, and the empty bunch otherwise. The notation of guarded bunches greatly simplifies and enhances the presentation of theory and proofs which will subsequently follow in Chapter 6. We also need the concept of guarded bunches to express comprehension in its more familiar form involving a constraining predicate.

---

Indeed in ordinary bunch theory the proper bunches of any given type $T$ also form a complete lattice whose top is $\text{null}$, and whose bottom is the carrier bunch $T_b$ of that type. This is somewhat similar to the subset inclusion ordering on sets. In Improper Bunch Theory we simply extend this structure by ‘plugging in’ a new bottom element $\bot$ below the original one $T_b$. 

---

\footnote{Indeed in ordinary bunch theory the proper bunches of any given type $T$ also form a complete lattice whose top is $\text{null}$, and whose bottom is the carrier bunch $T_b$ of that type. This is somewhat similar to the subset inclusion ordering on sets. In Improper Bunch Theory we simply extend this structure by ‘plugging in’ a new bottom element $\bot$ below the original one $T_b$.}
5.4.5 The Pre-conditioned Bunch

The construct of a pre-conditioned bunch is defined in terms of guarded bunches:

Definition 9. \( P \mid E \) =\(_{df}\) \( P \rightarrow E, \neg P \rightarrow \bot \)

\( P \mid E \) evaluates to the value of the expression \( E \) wherever \( P \) holds, but otherwise to the improper bunch \( \bot \). In fact the first guard \( P \rightarrow \ldots \) is irrelevant since under the condition \( \neg P \) the left hand of the bunch union is consumed by \( \bot \) anyway. Therefore the following identity holds:

\[
P \mid E = E, \neg P \rightarrow \bot \tag{5.1}
\]

and being simpler will be used in preference to Def. 9 when unfolding the definition of a pre-conditioned bunch.

5.4.6 Bunch Comprehension

Bunch comprehension is similar in concept to set comprehension. For this sake we introduce the following fundamental notation:

\[
\§ z \bullet E
\]

where \( z \) is a variable (or possibly a list of variables), and \( E \) an expression. The types of the variables in \( z \) must be inferable from the expression \( E \) as explained in Section 5.3.6, otherwise a guard may be used to further specify a type-constraining predicate if required. The comprehension expression then denotes the bunch of all values taken by \( E \) as each variable of \( z \) ranges over the elementary values of its type.

A more familiar form of comprehension which Hehner calls the solution quantifier [Heh93] restricts the values to be included in the comprehension to those satisfying some predicate \( P \). Using our syntax of comprehension as well as the previously introduced notation of a guarded bunch we can express this nicely in the following form:

\[
\§ x \bullet P \rightarrow x
\]

If moreover we wanted to include all values of some given expression \( E \) for which \( x \) fulfils some predicate \( P \) the according syntax would be

\[
\§ x \bullet P \rightarrow E
\]

Interestingly all these variation of bunch comprehension are entailed by the fundamental notation only involving a quantified variable and expression \( E \).
CHAPTER 5. IMPROPER BUNCH THEORY

Degenerate Bunch Comprehension. A degenerate case of bunch comprehension arises when the list of range variables featuring in the comprehension is empty. Denoting such an empty list by $\emptyset$ we have

Definition 10. $\emptyset \cdot E =_{df} E$.

5.4.7 Substitution

We will write $E[x \setminus F]$ for denoting the expression syntactically derived from $E$ by replacing each free occurrence of the variable $x$ by the expression $F$. Both $E$ and $F$ may be bunch expressions. Additionally we allow simultaneous substitution as for example in $E[x_1, x_2 \setminus F_1, F_2]$. Here we substitute in parallel each variable in a list $x_1, x_2 \ldots x_n$ by the corresponding bunch expressions in the list $E_1, E_2 \ldots E_n$.\footnote{For the degenerate case of an empty list of variables we agree $E[\emptyset \setminus \emptyset] = E$.}

We remind the reader that the ambiguity in syntax using comma for bunch union as well as separating the elements of a list of bunch expressions ought to be resolved by type checking if parenthesis are omitted.

Note that syntactic substitution (i.e. $\beta$-substitution) neither generally distributes through bunch union, nor is strict with respect to $\text{null}$ or $\bot$ in its second argument. The following counter example illustrates this:

$$1[x \setminus \text{null}] = 1 \quad \text{and} \quad \{x\}[x \setminus 1, 2] = \{1, 2\} \neq \{1\}, \{2\} = \{x\}[x \setminus 1], \{x\}[x \setminus 2]$$

Nevertheless we have distributivity through bunch union and strictness with respect to $\text{null}$ and $\bot$ in the first argument of substitution, thus generally

$$\text{null}[x \setminus E] = \text{null}, \quad \bot[x \setminus E] = \bot \quad \text{and} \quad (E_1, E_2)[x \setminus F] = E_1[x \setminus F], E_2[x \setminus F].$$

If all underlying operators in $E$ distribute through bunch union, we also have distributivity in the second argument, since if $E = f(x)$ with $f$ distributing through bunch union we can conclude that

$$f(x)[x \setminus F_1, F_2] = f(F_1, F_2) = f(F_1), f(F_2) = f(x)[x \setminus F_1], f(x)[x \setminus F_2].$$

Whether $E$ can be treated in this way as a distributive function of its free variables might in certain cases be determined syntactically.

Remark We toyed at some point with the idea of defining substitution so that it always distributed through bunch union in both arguments, and indeed was strict with respect to $\text{null}$ and $\bot$ in its second argument. This was for the particular application at hand a desired property, however one of the anonymous reviewers of the respective paper pointed out that this notion of substitution wasn’t syntactic, i.e. couldn’t be eliminated by means of syntactic replacement.
5.4.8 Functions

Functions in our treatment can either be defined through \( \lambda \)-abstractions or, as in the theory of B, by means of their corresponding graph. As in B the graph of a function is the set of its mappings (or maplets), thus the type of a function with domain type \( T_D \) and range type \( T_R \) is \( \mathcal{P}(T_D \times T_R) \) — there is no need for a designated function type.

In bunch theory functions may not just be applied to elementary values but to bunches of such, including the empty and improper bunch. As with lifted operators (see Section 5.3.4) we required function application to be strict with respect to \( \text{null} \) and \( \bot \), and to distribute through bunch union. Formally these properties are:

\[
f(\text{null}) = \text{null}, \quad f(\bot) = \bot \quad \text{and} \quad f(E, F) = f(E), f(F).\]

The following definition of function application fulfils these requirements.

**Definition 11.** Let \( f : D \rightarrow R \) be a partial function applicable to elements, and \( E \) an expression of type \( D \). We lift \( f \) to become a function on bunches through

\[
f(E) =_{df} E \neq \bot \mid \exists x \bullet x : E \rightarrow f(x) \quad \text{where} \quad x \notin E
\]

Note that if \( f \) is applied outside its domain, i.e. \( x \notin \text{dom}(f) \), we agree that \( f(x) = \bot \). An equivalent, and maybe more elegant form of Def. 11 could be given by \( f(E) =_{df} \sim f(\{ E \}) \) making use of (un)packaging and the lifted relational image.

To complete the definition of function application we further have to clarify what is meant by a function being applied to an element. Here bunch theory offers some interesting possibilities since the application \( f(x) \) is not constrained to yield an elementary value. In the theory of B we cannot infer anything about the value of \( f(x) \) if \( f \) was applied outside its domain i.e. \( x \notin \text{dom}(f) \). In Improper Bunch Theory however we can identify \( f(x) \) in such cases with either the specific value \( \bot \), or alternatively \( \text{null} \) if we preferred so. Furthermore in conventional B the syntax of function application can only be sensibly used if the underlying relation (i.e. the graph of the function) has indeed the functional property, namely that each domain element of the relation is associated at most with one range element. Again in bunch theory we can relax this constraint and allow function application to be generally used on relations regardless whether they are functional, or not.

The previous observations and ideas are reflected by the following three definitions which formalised what we mean by ‘function application to elements’ in the context of \( \lambda \)-expressions as well as graphs.

---

\(^9\)Alternatively we could have associated the graph of a function directly with the bunch its mappings. This approach does in fact have advantages with regards to functions equality and refinement, nevertheless we decided not to pursue it here in order to remain more closely within the theory of B.
Definition 12. Let $x$ be an elementary expression, i.e.
asserting $\Delta x$, then

$$(\lambda z \bullet E)(x) \overset{\text{df}}{=} E[z/x]$$

Hence for elementary values function application in $\lambda$-expressions is defined by means of syntactic substitution.

Definition 13. Let $x$ be an elementary expression, i.e.
asserting $\Delta x$, and $f \neq \perp_P(D \leftrightarrow R)$ then

$$f(x) \overset{\text{df}}{=} \begin{cases} x \in \text{dom}(f) \mid \exists y \bullet x \mapsto y : \sim f \mapsto y \\ x \not\in \text{dom}(f) \end{cases}$$

Note that with the alternative approach of lifting logical operators proposed in Section 5.3.4 we could as well have given the slightly more concise definition

$$f(x) \overset{\text{df}}{=} \begin{cases} x \in \text{dom}(f) \mid \exists y \bullet x \mapsto y \in^* f \mapsto y \\ x \not\in \text{dom}(f) \end{cases}$$

managing without the unpackaging of $f$ in place of Def. 13.

The third and concluding definition determines what happens if we apply a function of which the graph is equal to $\perp_P(D \leftrightarrow R)$.

Definition 14. Let $x$ be an elementary expression, i.e.
asserting $\Delta x$, and $f = \perp_P(D \leftrightarrow R)$ then

$$f(x) \overset{\text{df}}{=} \perp_R$$

### 5.5 Frequently used Bunch Laws

In this section we present a collection of elementary bunch laws which subsequently will prove useful when conducting formal proofs in Improper Bunch Theory. For the majority of laws we provide proofs which employ the idea of so-called case tables$^{10}$. Case tables are a way of presenting a proof by cases in tabular form. The underlying cases are usually determined by the combination of truth values of predicates occurring in the proposition. For each case, arising from a particular combination of truth values of the involved predicates, we determine by means of evaluation that the left hand and right hand of the proposition are equal in value.

Note that we don’t make any claim in this section of exhausting the subject of bunch laws in Improper Bunch Theory, the ones we present here are primarily those which proved useful in the context of work for this PhD. Furthermore, since all elementary bunch laws of Hehner’s bunch theory (excluding the ones regarding

---

$^{10}$For supplying the idea of illustrating proofs of bunch laws via case tables I am in particular indebted to Steve Dunne who first utilised them when teaching the fundamentals of pv semantics in the Formal Aspects of Computer Science module at the University of Teesside.
functions) also hold in our theory of bunches, the reader may be directed to [Heh93] as a secondary source for bunch laws.

**Proposition 15.** Bunch guard distributes through bunch union:

\[ P \to (E, F) = P \to E, P \to F \]

*Proof.* The proof is presented as a case table.

<table>
<thead>
<tr>
<th>P</th>
<th>l.h.s.: ( P \to (E, F) )</th>
<th>r.h.s.: ( P \to E, P \to F )</th>
</tr>
</thead>
<tbody>
<tr>
<td>true</td>
<td>true ( \to (E, F) = E, F )</td>
<td>true ( \to E, \text{true} \to F = E, F )</td>
</tr>
<tr>
<td>false</td>
<td>false ( \to (E, F) = \text{null} )</td>
<td>false ( \to E, \text{false} \to F = \text{null, null} ) = null</td>
</tr>
</tbody>
</table>

**Proposition 16.** Bunch guard distributes through bunch intersection:

\[ P \to (E \cdot F) = P \to E \cdot P \to F \]

*Proof.* Similar to the one of Prop. 15 exchanging bunch union for bunch intersection, and using \( \text{null} \cdot \text{null} = \text{null} \).

**Proposition 17.** Bunch pre-conditioning distributes through bunch union:

\[ P \mid E, F = (P \mid E), (P \mid F) \]

*Proof.* The proof is presented as a case table.

<table>
<thead>
<tr>
<th>P</th>
<th>l.h.s.: ( P \mid E, F )</th>
<th>r.h.s.: ( (P \mid E), (P \mid F) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>true</td>
<td>true ( \mid E, F = E, F )</td>
<td>(true ( \mid E), (\text{true} \mid F) = E, F )</td>
</tr>
<tr>
<td>false</td>
<td>false ( \mid E, F = \bot )</td>
<td>(false ( \mid E), (\text{false} \mid F) = \bot, \bot = \bot )</td>
</tr>
</tbody>
</table>

**Proposition 18.** Bunch pre-conditioning distributes through bunch intersection:

\[ P \mid E \cdot F = (P \mid E) \cdot (P \mid F) \]

*Proof.* Similar to the one of Prop. 17 exchanging bunch union for bunch intersection, and using \( \bot \cdot \bot = \bot \).

**Proposition 19.** Expanding of guarded bunch if the guard has the form of a conjunction:

\[ P \land Q \to E = P \to E \cdot Q \to E \]

*Proof.* The proof is presented as a case table.
**Proposition 20.** Expanding of guarded bunch if the guard has the form of a disjunction:

\[ P \lor Q \rightarrow E = P \rightarrow E, Q \rightarrow E \]

*Proof.* The proof is presented as a case table.

<table>
<thead>
<tr>
<th>( P )</th>
<th>( Q )</th>
<th>l.h.s.: ( P \land Q \rightarrow E )</th>
<th>r.h.s.: ( P \rightarrow E \land Q \rightarrow E )</th>
</tr>
</thead>
<tbody>
<tr>
<td>true</td>
<td>true</td>
<td>true ( \land ) true ( \rightarrow E = E )</td>
<td>(true ( \rightarrow E ) \land (true ( \rightarrow E ) ( = E ) \land E )</td>
</tr>
<tr>
<td>true</td>
<td>false</td>
<td>true ( \land ) false ( \rightarrow E = ) null</td>
<td>(true ( \rightarrow E ) \land (false ( \rightarrow E ) ( = E ) \land null )</td>
</tr>
<tr>
<td>false</td>
<td>true</td>
<td>false ( \land ) true ( \rightarrow E = ) null</td>
<td>(false ( \rightarrow E ) \land (true ( \rightarrow E ) ( = E ) \land null )</td>
</tr>
<tr>
<td>false</td>
<td>false</td>
<td>false ( \land ) false ( \rightarrow E = ) null</td>
<td>(false ( \rightarrow E ) \land (false ( \rightarrow E ) ( = E ) \land null )</td>
</tr>
</tbody>
</table>

**Proposition 21.** Expanding of pre-conditioned bunch if the precondition has the form of a conjunction:

\[ P \land Q \mid E = P \mid E, Q \mid E \]

*Proof.* The proof is presented as a case table.
<table>
<thead>
<tr>
<th>$P$</th>
<th>$Q$</th>
<th>l.h.s.: $P \land Q \mid E$</th>
<th>r.h.s.: $P \mid E, Q \mid E$</th>
</tr>
</thead>
<tbody>
<tr>
<td>true</td>
<td>true</td>
<td>$true \land true \mid E$</td>
<td>$(true \mid E), (true \mid E) = E, E$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$= E$</td>
<td>$= E$</td>
</tr>
<tr>
<td>true</td>
<td>false</td>
<td>$true \land false \mid E$</td>
<td>$(true \mid E), (false \mid E) = E, \bot$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$= \bot$</td>
<td>$= \bot$</td>
</tr>
<tr>
<td>false</td>
<td>true</td>
<td>$false \land true \mid E$</td>
<td>$(false \mid E), (true \mid E) = \bot, E$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$= \bot$</td>
<td>$= \bot$</td>
</tr>
<tr>
<td>false</td>
<td>false</td>
<td>$false \land false \mid E$</td>
<td>$(false \mid E), (false \mid E) = \bot, \bot$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$= \bot$</td>
<td>$= \bot$</td>
</tr>
</tbody>
</table>

**Proposition 22.** Expanding of pre-conditioned bunch if the precondition has the form of a disjunction:

$$P \lor Q \mid E = P \mid E \cdot Q \mid E$$

*Proof.* The proof is presented as a case table.

<table>
<thead>
<tr>
<th>$P$</th>
<th>$Q$</th>
<th>l.h.s.: $P \lor Q \mid E$</th>
<th>r.h.s.: $P \mid E \cdot Q \mid E$</th>
</tr>
</thead>
<tbody>
<tr>
<td>true</td>
<td>true</td>
<td>$true \lor true \mid E$</td>
<td>$(true \mid E), (true \mid E) = E, E$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$= E$</td>
<td>$= E$</td>
</tr>
<tr>
<td>true</td>
<td>false</td>
<td>$true \lor false \mid E$</td>
<td>$(true \mid E), (false \mid E) = E, \bot$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$= E$</td>
<td>$= \bot$</td>
</tr>
<tr>
<td>false</td>
<td>true</td>
<td>$false \lor true \mid E$</td>
<td>$(false \mid E), (true \mid E) = \bot, E$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$= E$</td>
<td>$= E$</td>
</tr>
<tr>
<td>false</td>
<td>false</td>
<td>$false \lor false \mid E$</td>
<td>$(false \mid E), (false \mid E) = \bot, \bot$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$= \bot$</td>
<td>$= \bot$</td>
</tr>
</tbody>
</table>

**Proposition 23.** Chaining rule for guarded bunches:

$$P \land Q \rightarrow E = P \rightarrow Q \rightarrow E$$

*Proof.* The proof is presented as a case table.
**CHAPTER 5. IMPROPER BUNCH THEORY**

<table>
<thead>
<tr>
<th>$P$</th>
<th>$Q$</th>
<th>l.h.s.: $P \land Q \rightarrow E$</th>
<th>r.h.s.: $P \rightarrow Q \rightarrow E$</th>
</tr>
</thead>
<tbody>
<tr>
<td>true</td>
<td>true</td>
<td>true $\land$ true $\rightarrow E$ $=$ $E$</td>
<td>true $\rightarrow$ true $\rightarrow E$ $=$ true $\rightarrow E$ $=$ $E$</td>
</tr>
<tr>
<td>true</td>
<td>false</td>
<td>true $\land$ false $\rightarrow E$ $=$ null</td>
<td>true $\rightarrow$ false $\rightarrow E$ $=$ true $\rightarrow$ null $=$ null</td>
</tr>
<tr>
<td>false</td>
<td>true</td>
<td>false $\land$ true $\rightarrow E$ $=$ null</td>
<td>false $\rightarrow$ true $\rightarrow E$ $=$ false $\rightarrow E$ $=$ null</td>
</tr>
<tr>
<td>false</td>
<td>false</td>
<td>false $\land$ false $\rightarrow E$ $=$ null</td>
<td>false $\rightarrow$ false $\rightarrow E$ $=$ false $\rightarrow$ null $=$ null</td>
</tr>
</tbody>
</table>

**Proposition 24.** *Chaining rule for pre-conditioned bunches:*

$$P \land Q \mid E = P \mid Q \mid E$$

**Proof.** The proof is presented as a case table.

<table>
<thead>
<tr>
<th>$P$</th>
<th>$Q$</th>
<th>l.h.s.: $P \land Q \mid E$</th>
<th>r.h.s.: $P \mid Q \mid E$</th>
</tr>
</thead>
<tbody>
<tr>
<td>true</td>
<td>true</td>
<td>true $\land$ true $\mid E$ $=$ $E$</td>
<td>true $\mid$ true $\mid E$ $=$ true $\mid E$ $=$ $E$</td>
</tr>
<tr>
<td>true</td>
<td>false</td>
<td>true $\land$ false $\mid E$ $=$ $\bot$</td>
<td>true $\mid$ false $\mid E$ $=$ true $\mid E$ $=$ $\bot$</td>
</tr>
<tr>
<td>false</td>
<td>true</td>
<td>false $\land$ true $\mid E$ $=$ $\bot$</td>
<td>false $\mid$ true $\mid E$ $=$ false $\mid E$ $=$ $\bot$</td>
</tr>
<tr>
<td>false</td>
<td>false</td>
<td>false $\land$ false $\mid E$ $=$ $\bot$</td>
<td>false $\mid$ false $\mid E$ $=$ false $\mid E$ $=$ $\bot$</td>
</tr>
</tbody>
</table>

**Proposition 25.** *Distributivity of bunch comprehension through bunch union:*

$$\& z \cdot E, F = (\& z \cdot E), (\& z \cdot F)$$

**Proof.** Follows trivially from the definition of bunch comprehension.

**Proposition 26.** *Slide rule for bunch comprehension with guarded bunch:*

$$\& z \cdot P \rightarrow E = P \rightarrow \& z \cdot E \quad \text{if } z \text{ is not free in } P$$

**Proof.** Since $P$ is independent of $z$, whenever $P$ holds all values for $E$ are included while $z$ ranges over the elementary values of its type. If $P$ doesn’t hold both sides of the proposition are equal to null.
Proposition 27. Slide rule for bunch comprehension with pre-conditioned bunch:
\[ \section{z} \cdot \section{P} \mid E = \section{P} \mid \section{z} \cdot E \] if \( z \) is not free in \( P \)

Proof. Analogous to the one of Prop. 26. Since \( P \) is independent of \( z \), whenever \( P \) holds all values for \( E \) are included while \( z \) ranges over the elementary values of its type. If \( P \) doesn’t hold both sides of the proposition are equal to \( \bot \).

Proposition 28. Splitting law for bunch comprehension:
\[ \section{z} \cdot \section{P} \lor \section{Q} \rightarrow E = (\section{z} \cdot \section{P} \rightarrow E), (\section{z} \cdot \section{Q} \rightarrow E) \]

Proof. Expanding the guarded bunch \( \section{P} \lor \section{Q} \rightarrow E \) according to Prop. 20 and using distributivity bunch comprehension (Prop. 25).

Proposition 29. Elimination law for bunch comprehension:
\[ \section{z} \cdot E = E \] if \( z \) is not free in \( E \)

Proof. This identity is evident i.e. as \( E \) in its value doesn’t depend on \( z \), and \( z \) will always range over some elements.

Proposition 30. Elimination law for bunch comprehension with guarded bunch:
\[ \section{z} \cdot \section{P} \rightarrow E = (\exists \section{z} \cdot \section{P}) \rightarrow E \] if \( z \) is not free in \( E \)

Proof. Since the value of \( E \) doesn’t depend on \( z \), all that determines whether the left hand is non-empty and thus equal to \( E \) is whether there is some element \( z \) such that the guarding predicate \( P \) holds.

Proposition 31. Elimination law for bunch comprehension with pre-conditioned bunch:
\[ \section{z} \cdot \section{P} \mid E = (\forall \section{z} \cdot \section{P}) \mid E \] if \( z \) is not free in \( E \)

Proof. If there is at least one element \( z \) for which \( P \) doesn’t hold, the inner pre-condition evaluates to \( \bot \) for this element “swamping” any value \( \section{P} \mid E \) may take for other \( z \), and thus rendering the whole of the bunch comprehension identical to \( \bot \).

Proposition 32. Conventional one-point rule for bunch comprehension:
\[ \section{z} \cdot \section{z} = \section{F} \mid \section{E} = \Delta \section{F} \rightarrow E[z \setminus \section{F}] \]

Proof. As can be seen, the one-point rule for bunch comprehension is very similar to the corresponding one for set comprehension. The important difference is that for \( z = \section{F} \) to be true for any \( z \) at all, we have to insist on \( \section{F} \) being elementary as
z only takes elementary values. Consequently, under the assumption $\neg \Delta F$ the comprehension must be equal to null (and so is the right hand of the proposition).

**Proposition 33.** Generalised one-point rule for bunch comprehension:

$$\exists z \cdot z : F \rightarrow E = (\lambda z \cdot E) F \text{ providing } F \neq \bot$$

**Proof.** Evident by applying the definition of function application, namely Def. 11 and Def. 12.

### 5.6 Impact on Tool Support

In this section we will briefly outline and discuss the impact on tool support if bunches were to be properly integrated into B. As previously stated, performing such an integration in practice is a more complex project and therefore beyond the immediate scope of research and practical work for this PhD.

Generally there appear to be a variety of options in which Improper Bunch Theory as presented in this chapter may be reconciled with tool-supported B development. If our aim would be not to incur substantial changes to the existing tools such as the Atelier B or B-Toolkit, but reuse them as they are, a viable approach may be to translate B specifications that utilise bunch notation into equivalent B specifications which are purely expressed in terms of set theory. Hehner in [Heh93] introduces and defines sets by appealing to bunch theory, and vice versa it seems evident that similarly bunches are definable in terms of sets (in [MB01] Morris and Bunkenburg in principle do this by giving a denotational model for their variant of bunch theory). We stipulate that a translation eliminating bunch expressions from B-AMN code can be carried out mechanically hence without requiring any user assistance or interaction. Clearly the advantage of this method would be that the only tool component which has to be developed is an appropriate translator. Whether type analysis is essential for this, or otherwise translation can be solely directed by syntax analysis of the AMN source is an open question which we won’t attempt to answer in short at this point (note that the latter would be advantageous as it may obviously simplify the development of the translator component). The main problem we are faced with here is the improper bunch which for the sake of this integration we need to represent by some sort of improper value extending each type of a B specification as well as the default base types. This is not a trivial problem, and it’s solution is subject to further research and experimentation.

The previous approach, despite being quite universal in its nature, has clearly some short-comings. An important one may be that when generating proof obligations and using the built-in interactive theorem provers of the commercial tools we might not be able to easily associate theorems with respective elements of AMN in the B components. This is so since all bunch syntax is eliminated in the translation phases which must happens prior to any further analysis performed.
CHAPTER 5. IMPROPER BUNCH THEORY

by the B tools. Another problem is that the translation engine may not be able to recognise all syntactic and type errors in the specification, and it could occasionally become difficult to match errors when they are reported retrospectively by the tools to their occurrence in the originally AMN text. Thus an alternative strategy may be suggested which attempts to integrate bunches directly into the respective B tools. Doing so we clearly would need access to the source code and documentation of either the B-Toolkit or Atelier B. How successful this method could turn out depends largely on the quality of the code, design and documentation of the existing tools. The effort is justified by providing scope for a tighter integration as we are not only supporting bunches on a syntactic level while characterising their semantics in terms of syntactic transformation into sets as before, but also axiomatise them within the tools by appropriate theory files encapsulating laws in form of proof rules for each newly introduced element of syntax. A general problem here is that all default theories of B may have to be scrutinised whether they contain rules which might not generally hold anymore in the presence of bunches. Unfortunately even fundamental properties such as distributivity of arithmetic (indeed of all lifted) operators fall into this category. This is illustrated by the following example:

\[ 1, 2 \times (3 + 4) = (1, 2) \times 7 = 7, 14 \]

however applying the distributivity law for arithmetic expressions we obtain

\[ ((1, 2) \times 3) + ((1, 2) \times 4) = (3, 6) + (4, 8) = 7, 10, 11, 14 \]

Notice that we nevertheless have distributivity under the assumption that all constituent bunch expressions are elementary. This can be exploited by guarding the problematic proof rules with conditions requiring elementhood of all involved expressions. Otherwise we might still be able to say something weaker in the case of expressions being bunches, notice that in the above example 7, 14 is a subbunch of 7, 10, 11, 14 which suggests an alternative property we may call semi-distributivity replacing equality in the original rule by the weaker subbunch inclusion.

Particular caution pursuing this approach (indeed others too) would have to be given to avoid clashes with existing syntax and notations (in Section 5.3 we already pointed out the potential risk for syntactic clashes e.g. due to the ambiguous use of comma). Nevertheless this shouldn’t give rise to any fundamentally insoluble practical problems. The feasibility of this approach depends ultimately whether the source code and documentation for one of the commercial tools can be obtained. Clearsy has presumably already steered into this direction by making a stripped-down version of the Atelier B freely available with the “B4free” distribution; introducing an open source version of the same tool could be a further step which they hopefully will considered at some point in the future.

The final approach which we want to mention here, and which indeed was the one we adopted for the practical purpose of this PhD research, is to develop
our own set of tools “from scratch” supporting the essential features of the commercial tools, but providing an open design that allows for flexible adaption and modification of the underlying AMN syntax, processing rules, proof obligation generation, etc. This is clearly the most flexible approach as the design of the application can be tailored and actively geared towards making extensions as easy as possible to be integrated by the user, however as it shows it seems also the strategy requiring the most overall software development effort. We think that in this context it would be best not to develop a stand-alone application, but to provide a tool library with a set of reusable components. This library may then form the basis for a variety of tools and extensions to the B formalism, in particular it might also help us to achieve RB0 translation which is briefly discussed in Section 7.2 and 7.4.5. The essential components of such a library, in the context of integrating bunches into B, would be a syntactic analyser, type-checker, and processing utility able to generation standard proof obligations for the abstract machines and refinements of a B development. To avoid having to re-invent the wheel entirely we suggest that parts of the commercial tools such as the automatic and interactive theorem provers may be reused if possible. A very appealing solution would be to achieve integration with B4free and Click’n Proof, in particular since they are both freely available and Click’n Proof provides some interesting and useful features which even the theorem provers of the commercial B tools lack.

To conclude this section we remark that a considerable amount of development work has already been invested into the production of a component library written in Java which we termed “OpenB”, and which moreover was presented as a poster session at the ZB 2005 conference. It should soon be openly available in form of a beta release from the website \url{http://openb.scm.tees.ac.uk/}. Ultimately we hope that this software project will enable us to realise the third approach described of integrating bunches into the B Method without the investment of a disproportionate amount of time and effort, mostly since the core components of the tool library have already been developed and tested over the duration of time for this PhD investigation.

5.7 Conclusions

In this chapter of the thesis we have developed an extension of Hehner’s bunch theory by most notably enriching it with an improper bunch \( \perp_T \) for each type \( T \). These special bunches have the property of absorbing any other bunch, and importantly all functions and indeed most operators are strict with respect to them. The motivation for incorporating improper bunch into our bunch theory was to provide the means for describing the outcome of non-terminating computations \textit{at the level of expressions}.

Another important property of our bunch theory is that it is more in line with the theory of B than other existing versions of bunch theory [Heh81, Heh93,
MB01], and thus should potentially lend itself more readily for a future integration into the B tool support. This is first so since our bunches are strictly typed, but additionally by avoiding to a large extent potential incisions on the theory of B and its underlying logic which may result from such an integration. To give two examples, Improper Bunch Theory consequently prevents bunches from entering the realm of predicative logic (previous work on bunch theory seems to be more liberal on this issue), and moreover functions are treated in a way that is compatible with the approach taken in the B Method.

To render Improper Bunch Theory more useful in the context of prospective-values semantics we additionally defined the guarded and pre-conditioned bunch operators. Subsequently we presented a collection of laws which facilitate formal reasoning and proof involving the introduced operators. For the sake of proving identities in bunch theory we propose the notion of case tables to present such proof in succinct manner.
Chapter 6

Prospective-value Semantics

6.1 Introduction

In Chapter 4 we presented the idea of expression transformers by first introducing them informally, and giving some motivating examples and arguments which justify their usefulness and importance in the context of work for this PhD. In this chapter we will develop a formal theory of expression transformers by linking them directly to the, in the B world, familiar relational semantic model of the Generalised Substitution Language involving the characteristic predicates trm and prd. This theory, as stated already in Chapter 4, we call prospective-value (for short pv) semantics. Its development satisfies objective 5 of the investigation stated in Section 1.2 showing that pv semantics can form an alternative semantic foundation for a sequential programming language exhibiting equal expressive power as, for example, the wp calculus. The feasibility of this project is moreover a secondary hypothesis of this PhD. We will moreover see that beyond the scope of reversible computations, prospective values offer an interesting view of program semantics, and an alternative way to reason about programs; therefore pv semantics may be regarded as a rewarding and worthwhile field of study and research in its own right.

The important feature of the theory of prospective values developed in this chapter is that it accommodates all aspects of behaviour of generalised substitutions including non-determinism, infeasibility and non-termination. With the introduction of Improper Bunch Theory in the previous chapter we have laid the foundations for succeeding in this endeavour by extending the expressive power of expressions, in particular to reflect the abortive outcome of a computation on the level of bunch values. We will show that our theory of prospective-value semantics is as discriminating as wp semantics, hence is isomorphic to other commonly used total-correctness semantic models of the GSL such as, for example, the wp calculus, or alternatively the relational and set-based semantic model of the GSL presented in the B-Book [Abr96b].

The structure of this chapter is as follows. In Section 6.2 we acknowledge previous work inspiring our contribution on the pv calculus in this chapter. We
then present in Section 6.3 the fundamental definition of the prospective value \( S \circ E \) of an expression \( E \) after executing computation \( S \) by means of a closed-form, and provide a few examples of how this definition is to be applied in practice. The subsequent Section 6.4 gives a comprehensive collection of rewrite laws along with their proofs, which simplify the evaluation of prospective values and justify the term ‘calculus’ to be used in connection with pv semantics. In Section 6.5 we explain how the characteristic predicates of a generalised substitution can be derived from its prospective-value transformer. Section 6.6 gives a few results on linking pv semantics directly to wp semantics, and in the following Section 6.7 we examine the prospective value of the while-loop iteration construct by characterising it as a fixed point in expressions. Finally, in Section 5.7 we draw our conclusions on the material presented in this chapter.

**Note** Most of the content and contribution in this chapter draws from the author’s publications at the ZB 2003 and ZB 2005 conferences [SZ03, ZSD05].

### 6.2 Related Work

The most significant contribution regarding the use of expression transformers to reason about programs was in an unpublished paper by Morris and Bunkenburg with the title ‘Term Transformer Semantics’ [MB99]. The motivation for this work was to reconcile functional and imperative programming in a unified theory without having to give up the advantage of referential transparency enjoyed in pure functional languages due to the absence of side-effects. In this article they define the notion of what they call a term transformer, and introduced the syntax \( C \star E \) for it. Here \( C \) stands for a computation expressed in a specimen language they define in [MB99], and \( E \) for an expression of some type. This paper in the preliminary version that was available to us contained already a wealth of interesting ideas, raising for example issues how the invariance theorem for loops may be generalised in terms of *invariant terms*, what are the fundamental properties of \( C \star E \) i.e. healthiness conditions, how do term transformers relate to wp semantics, etc.

Apart from [MB99] we are unaware of any other work on the subject developed in this chapter, in particular none which offers a term (or expression) transformer semantics for the GSL.
### 6.3 Definition of $S \diamond E$

In prospective-value semantics, the meaning of a computation $S$, expressed in the generalised substitution language, is given by the value an expression $E$ may take were $S$ to be carried out. We denote this value (being a bunch) by $S \diamond E$, it is indeed the prospective value of $E$ after execution of $S$. Similarly to transforming a predicate in wp semantics, the application of $\_ \diamond \_ $ in pv semantics has no state-changing side effects. Note that the variables occurring free in $S \diamond E$ refer to the state before executing $S$. The following closed form involving the trm, prd and frame\textsuperscript{1} of $S$ serves as a semantic definition for the prospective value $S \diamond E$:

**Definition 34.** Let $S$ be a generalised substitution with $s$ being its frame, and $E$ an arbitrary expression of some type. Then

$$S \diamond E =_{df} \text{trm}(S) \mid \% s' \bullet \text{prd}(S) \longrightarrow E[s \setminus s']$$

To remind the reader of the wp definitions for trm$(S)$ and prd$(S)$ they are included in Fig. 6.1. The syntax $\% z \bullet E$ is used for bunch comprehension, yielding the bunch of values for $E$ where variable $z$ ranges over the *elementary* values of some type which can be inferred from the expression $E$. The guarded form $\% z \bullet P \longrightarrow E$ gives us the bunch of all values $E$ where the selected elements for $z$ moreover satisfy $P$. Note that in Def. 34 the placeholder $s$ being the frame of $S$ may not just stand for a single variable but in fact a list of such, which may well be empty e.g. in the case of *skip* or *abort*.

#### 6.3.1 Explanation of the Definition for $S \diamond E$

To gain a better intuitive understanding of Def. 34 let us dissect it by examining the part inside the pre-conditioned bunch first.

\footnote{Formally frames for generalised substitutions were introduced in [Dun02]. The frame is the list of variables a generalised substitution acts upon. Frames prove to be necessary when considering generalised substitutions outside the context of a B operation, in particular to give a meaningful definition of parallel composition. For example, *skip* and $x := x$ cannot be distinguished by their wp effect alone; however they are provably different which is revealed by putting them into parallel composition, for example, with $x := x + 1$: whereas $\text{skip} \parallel x := x + 1 = x + 1$ we have that $x := x \parallel x := x + 1 = \text{magic}$.}

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Definition</th>
<th>Remark</th>
</tr>
</thead>
<tbody>
<tr>
<td>trm($S$)</td>
<td>wp($S, \text{true}$)</td>
<td></td>
</tr>
<tr>
<td>prd($S$)</td>
<td>$\neg \text{wp}(S, s \neq s')$ where $s$ is the frame of $S$</td>
<td></td>
</tr>
</tbody>
</table>

Figure 6.1: Definitions of trm and prd of a generalised substitution $S$. 
In \( \dot{s} \, \dot{s}' \cdot \text{prd}(S) \rightarrow E[s \backslash s'] \) the variable list \( s' \) designates the after-state variables determining the value of \( E \) by substituting the free occurrences of \( s \) in it. The guarded bunch ensures that \( E \) is evaluated and included in the result of the bunch comprehension only when \( \text{prd}(S) \) holds, in other words whenever \( s \) and \( s' \) are selected to represent some possible behaviour of \( S \). Since \( s' \) is quantified in the bunch comprehension, \( E \) is evaluated in every possible after-state which could follow the before-state \( s \). Note that \( s \) remains free in the overall comprehension expression.

Taking \( \dot{s} \, \dot{s}' \cdot \text{prd}(S) \rightarrow E[s \backslash s'] \) by itself might suffice to obtain a partial-correctness picture of \( S \) in terms of its prospective value. This is so since wherever \( S \) doesn’t terminate, \( \text{prd}(S) \) holds true yielding the computation as entirely non-deterministic, thus including all possible values for \( E \) while \( s \) ranges over the elementary values of its type. The previous fact is a consequence of the healthiness condition \( \neg \text{trm}(S) \Rightarrow \text{prd}(S) \) (Prop. 2.1 in [Dun02]) holding true for the \( \text{trm} \) and \( \text{prd} \) of a generalised substitution. In order to distinguish however the non-deterministic assignment \( x \in \mathbb{N} \) from the non-terminating computation \text{abort}_x \) we additionally have to capture the possibility of non-termination. This is, at the level of expressions, achieved by augmenting our theory with improper bunches, and at the level of pv semantics through pre-conditioning the comprehension expression in Def. 34 with \( \text{trm}(S) \).

Doing so we obtain the full total-correctness semantic picture of \( S \). For example, under the assumption \( \neg \text{trm}(S) \) we have \( S \odot E = \bot \) which can never arise naturally from the bunch comprehension expression alone under the assumption that \( E \neq \bot \).

### 6.3.2 Examples for Calculating \( S \odot E \)

To illustrate the application of the closed-form definition Def. 34 let us consider the following generalised substitution:

\[
S \triangleq x > 0 \mid (x := x + 1 \parallel x := x + 2)
\]

To apply Def. 34 we first have to calculate the frame, \( \text{trm} \) and \( \text{prd} \) of \( S \). To aid their derivation we employ some of the laws given in Fig. 6.2 instead of appealing directly to the definition of \( \text{trm} \) and \( \text{prd} \) in Fig. 6.1. Trivially we can see that the frame of \( S \) only consists of the variable \( x \) being of type \( \mathbb{N} \). For the \( \text{trm} \) and \( \text{prd} \) of \( S \) we have

\[
\text{trm}(S) = \text{trm}(x > 0 \mid x := x + 1 \parallel x := x + 2) = x > 0 \land \text{trm}(x := x + 1) \land \text{trm}(x := x + 2) = x > 0
\]
and

\[
\begin{align*}
\text{prd}(S) & = \text{prd}(x > 0 \mid x := x + 1 \land x := x + 2) \\
& = x > 0 \Rightarrow \text{prd}(x := x + 1 \land x := x + 2) \\
& = x > 0 \Rightarrow \text{prd}(x := x + 1) \lor \text{prd}(x := x + 2) \\
& = x > 0 \Rightarrow x' = x + 1 \lor x' = x + 2
\end{align*}
\]

With the derived characteristic predicates trm(S) and prd(S) we can now calculate the prospective value for any given expression \( E \) using Def. 34. To show this, let us for example derive the prospective value of the expression \( 2x \) after execution of \( S \):

\[
S \circ 2x
\]

\[
\begin{align*}
& = \text{“Defn of } S \circ E \text{ (Def. 34) with frame}(S) = x” \\
& = \text{trm}(S) \mid \frac{x'}{x} \cdot \text{prd}(S) \to (2x)[x'\backslash x'] \\
& = \text{“Rewriting trm}(S) \text{ and prd}(S) \text{ as derived above”} \\
& = x > 0 \mid \frac{x'}{x} \cdot x > 0 \Rightarrow (x' = x + 1 \lor x' = x + 2) \to (2x)[x'\backslash x'] \\
& = \text{“Substitution”} \\
& = x > 0 \mid \frac{x'}{x} \cdot x > 0 \Rightarrow (x' = x + 1 \lor x' = x + 2) \to 2x' \\
& = \text{“Logic”} \\
& = x > 0 \mid \frac{x'}{x} \cdot \neg(x > 0) \lor x' = x + 1 \lor x' = x + 2 \to 2x' \\
& = \text{“Splitting bunch comprehension (Prop. 28)”} \\
& = x > 0 \mid (\frac{x'}{x} \cdot \neg(x > 0) \to 2x'), \frac{x'}{x} \cdot x' = x + 1 \lor x' = x + 2 \to 2x' \\
& = \text{“Bunch law, } \neg(x > 1) \text{ is independent of quantified variable } x’” \\
& = x > 0 \mid \neg(x > 0) \to (\frac{x'}{x} \cdot 2x'), \frac{x'}{x} \cdot x' = x + 1 \lor x' = x + 2 \to 2x' \\
& = \text{“Distributivity of pre-conditioned bunch (Prop. 17)”} \\
& = (x > 0 \mid \neg(x > 0) \to \frac{x'}{x} \cdot 2x'), \\
& = x > 0 \mid \frac{x'}{x} \cdot x' = x + 1 \lor x' = x + 2 \to 2x' \\
& = \text{“Bunch identity: } P \mid \neg P \to E = P \mid \text{null”} \\
& = x > 0 \mid \frac{x'}{x} \cdot x' = x + 1 \lor x' = x + 2 \to 2x' \\
& = \text{“Splitting bunch comprehension (Prop. 28)”} \\
& = x > 0 \mid \frac{x'}{x} \cdot x' = x + 1 \to 2x', \frac{x'}{x} \cdot x' = x + 2 \to 2x' \\
& = \text{“One-point rule for bunch comprehension (Prop. 32)”} \\
& = x > 0 \mid (2x')[x'\backslash x + 1], (2x')[x\backslash x + 2] \\
& = \text{“Substitution”} \\
& = x > 0 \mid 2(x + 1), 2(x + 2)
\end{align*}
\]
Note that the result is an expression on the state before executing $S$. Its interpretation is that the value of $2x$ would be $x > 0 \mid 2(x+1), 2(x+2)$ after running $S$. Since $S$ behaves non-deterministically we obtain a bunch of elementary results when $x > 0$, namely $2(x+1), 2(x+2)$. In particular, if we invoke $S$ from a state where $x \neq 0$ (e.g. $x = 0$) the computation fails, which is indicated by yielding the outcome $\bot$. Formally, we are able to prove, for example, that the following statements are universally true:

$$x = 1 \Rightarrow S \sqcap 2x = 2, 3 \quad \text{and} \quad x = 0 \Rightarrow S \sqcap 2x = \bot$$

**Special case of calculating $S \sqcap E$ when the frame is empty.**

To strengthen our confidence in Def. 34 we submit it to the boundary case where the frame of the generalised substitution $S$ is empty. Thus we consider $S$ to be the substitution **skip**, and remind the reader that $\text{trm(skip)}$ and $\text{prd(skip)}$ are both equivalent to $\text{true}$.

**skip $\sqcap E$**

$$\Rightarrow \text{“Defn of } S \sqcap E \text{ (Def. 34) with frame(skip) = } \emptyset$$

$$\text{trm(skip) } \mid \emptyset ) \bullet \text{prd(skip)} \rightarrow E[\emptyset, \emptyset]$$

$$\Rightarrow \text{“Substitution: } E[\emptyset, \emptyset] \text{ simplifies to } E$$

$$\text{trm(skip) } \mid \emptyset ) \bullet \text{prd(skip)} \rightarrow E$$

$$\Rightarrow \text{“Rewriting trm(skip) and prd(skip)”}$$

$$\text{true } \mid \emptyset ) \bullet \text{true} \rightarrow E$$

$$\Rightarrow \text{“Simplifying pre-conditioned and guarded bunch”}$$

$$\emptyset ) \bullet E$$

$$\Rightarrow \text{“Degenerate case for bunch comprehension (Def. 10)”}$$

$$E$$

This coincides with our expectations since **skip**, while guaranteeing to terminate, has no state-changing effects, and therefore cannot alter the value of the expression $E$ when executed.

With the two examples provided here we hope to sufficiently illustrate the general idea of how the prospective value of a computation $S$ and expression $E$ in practical terms can be worked out using Def. 34. To simplify the calculations involved, the laws we summarised in Fig. 6.2 regarding the $\text{trm}$ and $\text{prd}$ of the various GSL constructs prove to be useful. Nevertheless we will see in what follows that there is an easier way of formally deriving the prospective value $S \sqcap E$ by means of rewrite laws inductively defined for each construct of the Generalised Substitution Language.
### Chapter 6. Prospective-Value Semantics

<table>
<thead>
<tr>
<th>Sub</th>
<th>( \text{trm}(\text{Sub}) )</th>
<th>( \text{prd}(\text{Sub}) )</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>skip</td>
<td>true</td>
<td>true</td>
<td></td>
</tr>
<tr>
<td>( z := E )</td>
<td>true</td>
<td>( z' = E )</td>
<td></td>
</tr>
<tr>
<td>( P \mid S )</td>
<td>( P \land \text{trm}(S) )</td>
<td>( P \Rightarrow \text{prd}(S) )</td>
<td></td>
</tr>
<tr>
<td>( P \Rightarrow S )</td>
<td>( P \Rightarrow \text{trm}(S) )</td>
<td>( P \land \text{prd}(S) )</td>
<td></td>
</tr>
<tr>
<td>( S \parallel T )</td>
<td>( \text{trm}(S) \land \text{trm}(T) )</td>
<td>( \text{prd}(S) \lor \text{prd}(T) )</td>
<td></td>
</tr>
<tr>
<td>( S ; T )</td>
<td>( \text{wp}(S, \text{trm}(T)) )</td>
<td>( \text{trm}(S) \Rightarrow (\text{prd}(S) \lor \text{prd}(T)) )</td>
<td></td>
</tr>
<tr>
<td>( @ z \cdot S )</td>
<td>( \forall z \cdot \text{trm}(S) )</td>
<td>( \exists z, z' \cdot \text{prd}(S) )</td>
<td></td>
</tr>
<tr>
<td>( S \parallel T )</td>
<td>( \text{trm}(S) \land \text{trm}(T) )</td>
<td>( (\text{trm}(T) \Rightarrow \text{prd}(S)) \land (\text{trm}(S) \Rightarrow \text{prd}(T))^a )</td>
<td></td>
</tr>
<tr>
<td>( S_y )</td>
<td>( \text{trm}(S) )</td>
<td>( \text{prd}(S) \land w = w' )</td>
<td>( w ) is ( y - s )</td>
</tr>
</tbody>
</table>

Note: This property underlies the liberalised parallel composition definition of generalised substitutions given in [Dun02].

Figure 6.2: Characteristic predicates \( \text{trm} \) and \( \text{prd} \) of each basic GSL construct.

#### 6.3.3 Notes on the Definition for \( S \diamond E \)

An important observation regarding Def. 34 is that in principle it is independent of the fact that we limit our attention to computations expressed in the Generalised Substitution Language. In other words, any computation that can be described by having a frame, \( \text{trm} \) and \( \text{prd} \) which obey certain healthiness conditions could potentially be subjected to this definition. Since these characteristic predicates may themselves be derived from the wp effect of a given computation, Def. 34 could as well be understood as establishing a link between pv and wp semantics.

Notice that fundamentally it is not obligatory to define pv semantics in the way we decided here, i.e. relating it indirectly to wp semantics by means of the characteristic predicates \( \text{trm} \) and \( \text{prd} \). An alternative approach would have been to develop a pv semantics for the GSL \( \text{ab initio} \) by providing individual defining laws for the pv effect of each generalised substitution. Nevertheless there are strong arguments in favour of the closed-form definition: first Def. 34 is very suggestive from an intuitive point of view; it encapsulates all there is to know about pv semantics in one concise equation. Secondly, predicate transformers in general, and moreover in the specific context of the GSL, have been extensively studied and widely used in practical application of proving program correctness.

---

2This link, more specifically, is an isomorphism relating the semantic domains of the wp predicate transformer and pv expression transformer. We argue formally for this to be true in Section 6.5.
such as the B Method. The concept of characteristic predicates of a generalised substitution is well understood, and there is little doubt whether Def. 34 could not be flawed leading to some inconsistency in the underlying calculus, e.g. violation of the axiom of Leibniz for referential transparency.

At this point we may content ourselves with what has been achieved in terms of providing a convincing formal definition of prospective values semantics, however it may also have become apparent that the proposed method of evaluating \( S \circ E \) could potentially lead to quite elaborate and long-winded calculations as it depends on calculating the trim and prd of the computations first, and subsequently substituting them in Def. 34. Confining ourselves to the closed-form definition, pv semantics doesn’t seem to inherit much of the elegance and simplicity of the wp calculus. Fortunately, it is possible to restore this elegance by virtue of algebraic rewrite laws which exist for each construct of the GSL. These laws enable us to ‘short circuit’ evaluation of \( S \circ E \), altogether avoiding reference to Def. 34 whereby in practical terms effectively dispersing with it. Furthermore, they crucially are not more complicated than the equivalent defining wp laws, especially as they take profit of Improper Bunch Theory and auxiliary operators such as the guarded and pre-conditioned bunch. In our first publication on pv semantics [SZ03] we actually used these laws to define prospective values, however we departed from this method in [ZSD05] according to the reasons explained earlier in this section.

### 6.4 Rewrite Laws for pv Semantics

The complete collection of algebraic rewrite laws including one for each construct of the GSL is given through Fig. 6.3. Note that \( S \) and \( T \) are arbitrary generalised substitutions with respective frames \( s \) and \( t \).

Our Assignment rule, in analogy to the the one presented in [Dun02], entails the case of Abrial’s simple as well as multiple assignment because \( z \) here denotes a list of variables (which may also be empty). No rule for parallel composition is provided as in pv semantics (similarly to wp semantics) no such simple rule seems to exist. Reasoning about parallel composition is nevertheless possible by falling back on Def. 34, and the characterisation of \( S \parallel T \) in Fig. 6.2. Iteration is not included in Fig. 6.3 since it is dealt with separately in Section 6.7 where we appeal to the fixed-point semantics of the while-loop construct; doing so we derive its prospective value as a fixed point in expressions.

The following paragraphs provide comments for each of the algebraic rewrite laws, explaining and justifying them operationally. Note that by means of these laws the \( \_ \circ \_ \) operator can be eliminated from any expression much in the same way as the wp rules for generalised substitutions allow for elimination of the wp operator from any predicate. The B Method and tool support take particular advantage of this when generating the proof obligations associated with a component by first translating AMN constructs into equivalent GSL computations,
and secondly eliminating generalised substitutions from all proof obligation predicates. Though our aim in this PhD is not to integrate the pv transformer into the B Method, the availability of pv rewrite laws is certainly relevant to facilitate this.

Skip

**Proposition 35.** The algebraic law for Skip is

\[ \text{skip} \circ E = E \]

**Comment.** Skip is the statement that “does nothing”, and thus has no state-changing effect. Therefore it cannot alter the value of the expression \( E \), accordingly must leave it unchanged. Skip may indeed be regarded as a special case of assignment where the list of assigned variables is empty.

Assignment

**Proposition 36.** The algebraic law for Assignment is

\[ z := F \circ E = E[z \setminus F] \]

**Comment.** Assignment changes the values of the variables in the list \( z \) simultaneously to the ones of expressions in the list \( F \). Both lists must have the same length, and in the limit case of the lists being empty Assignment behaves exactly like Skip (the latter is in fact provable from Def. 34).
Interestingly, observe that the pv rewrite laws for Assignment and Skip are exactly the same as the respective ones in the wp calculus.

Precondition

**Proposition 37.** *The algebraic law for Precondition is*

\[ P \mid S \odot E = P \mid S \odot E \]

**Comment.** The pre-conditioned substitution behaves like \( S \) from a state where its precondition holds, otherwise it fails to terminate. This is reflected by the rule, which, by means of a pre-conditioned bunch, ensures that the prospective value equals \( \bot \) under the assumption \( \neg P \), no matter what value of the expression \( E \).

Notice that pre-conditioning a generalised substitutions naturally translates into pre-conditioning of a bunch in pv semantics.

Guard

**Proposition 38.** *The algebraic law for Guard is*

\[ P \implies S \odot E = P \rightarrow S \odot E \]

**Comment.** The guarded substitution behaves like \( S \) from any state where its guard holds, otherwise is considered infeasible. Operationally we can think of \( P \implies S \) refusing to be executed under the assumption \( \neg P \), in wp semantics doing so would result in a ‘miracle’ achieving any desired post-conditions including contradictory ones.

In pv semantics we cannot expect any result(s) for values of \( E \) to be delivered under the condition \( \neg P \), clearly because \( S \) cannot execute then. This is reflected by the prospective value of the guarded substitution being equal to the empty bunch if the guard is false. In resemblance to the case of pre-conditioning, the guarding of a generalised substitution naturally translates into guarding of a bunch in pv semantics.

Choice

**Proposition 39.** *The algebraic law for Choice is*

\[ S \downarrow T \odot E = S \odot E, T \odot E \]

**Comment.** Choice, which as in B we interpret as non-deterministic choice, as the operational interpretation that it could behave according to either of its alternatives, and know knowledge can be gained according to which of them at any
time\(^3\). The prospective value of the choice construct captures this by recording the prospective value of both alternatives giving their bunch union. Any nondeterminism exhibited by \(S\) and \(T\) individually will naturally be flattened in the resulting bunch of elementary outcomes; this exemplifies why bunches are superior to sets in building the pv calculus.

Sequential Composition

**Proposition 40.** The algebraic law for Sequential Composition is

\[ S ; T \diamond E = S \diamond T \diamond E \]

**Comment.** Sequential Composition is presumably the only algebraic law which might appear counter-intuitive. In particular, if \(S ; T \diamond E\) is the prospective value of \(E\) after first running \(S\) and then \(T\), an immediate question may arise why not postulate \(S ; T \diamond E = T \diamond S \diamond E\) instead reflecting this order of evaluation?

Fortunately we don’t have to answer this question, or otherwise leave it to the reader for contemplation, since in what follows we will formally prove that the rule for sequential composition is correct as given above. A simple example follows giving the reader an indication that the sequential composition law holds:

\[
\begin{align*}
&x := 7 ; x := x + 1 \diamond x \\
&= \text{“Sequential Composition”} \\
&\quad x := 7 \diamond x := x + 1 \diamond x \\
&= \text{“Assignment”} \\
&\quad x := 7 \diamond x[x\backslash x + 1] \\
&= \text{“Substitution”} \\
&\quad x := 7 \diamond x + 1 \\
&= \text{“Assignment”} \\
&\quad (x + 1)[x\backslash 7] \\
&= \text{“Substitution & Arithmetic”}
\end{align*}
\]

Since \(x := 7 ; x := x + 1\) is equivalent to \(x := 8\) as a computation, this is indeed the expected result. On the other hand, if we swap the substitutions around replacing \(x := 7 \diamond x := x + 1 \diamond x\) by \(x := x + 1 \diamond x := 7 \diamond x\) and carry out the same derivation, we conclude the prospective value of \(x\) to be 7 which is clearly wrong.

\(^3\)A slightly exaggerated example is that \(S \mid \top\) \(T\) could behave like \(S\) in the morning, and \(T\) in the afternoon. The equivalent operational interpretation of a prospective value given through a non-elementary bunch is that the underlying computation will yield one element of that bunch. Which one we don’t know, and it could indeed be different with each run of the program.
Unbounded Choice

**Proposition 41.** The algebraic law for Unbounded Choice is
\[ @z \cdot S \diamond E = \xi z \cdot S \diamond E \quad \text{if} \quad z \notin E \]

**Comment.** While universal quantification generalises the choice law in wp semantics, which is expressed in terms of conjunction, bunch comprehension fulfils a similar function in pv semantics. Here the choice law is formulated in terms of bunch union, and its generalisation to Unbounded Choice makes use of a bunch comprehension instead. Operationally, the prospective value of Unbounded Choice is the bunch of all possible prospective values resulting from behaviours of \( S \) when \( z \) ranges over the elements of its type.

As in Assignment \( z \) here stands more generally for a list of variables. In the limit case of an empty variable list the bunch comprehension degenerates to \( S \diamond E \), hence \( @\emptyset \cdot S \diamond E \) is equal to \text{skip}.

Frame extension

**Proposition 42.** The algebraic law for Frame extension is
\[ S_y \diamond E = S \diamond E \]

**Comment.** Frame extension is somewhat a special operator not being part of the classical exposition of the GSL presented by Abrial in the B-Book[Abr96b]. Abrial didn’t need to formally introduce frames since his generalised substitutions are always assumed to operate within the context of some B component. In [Dun02] however it is argued that in order to provide a meaningful definition of parallel composition for generalised substitutions considered outside the context of a B component, we have to introduce the notion of frames into the GSL. The frame of a generalised substitution determines which variables could potentially be affected by it.

Operationally it is not possible to distinguish \( S \) from \( S_y \) since both exhibit the same behaviour (\( S_y \) potentially could modify \( y \), but it doesn’t if \( y \) lies outside the frame of \( S \), otherwise we are talking about the same substitution anyway). The difference between \( S \) and \( S_y \) becomes only ever apparent if we compose them in parallel with another computation having the frame \( y \). In such
\[ \text{skip} \parallel x := 1 \quad = \quad x := 1 \quad \text{whereas} \quad \text{skip}_x \parallel x := 1 \quad = \quad x = 1 \implies x := 1 \]

**Remark.** Strictly we didn’t have to use Dunne’s frame-based theory of generalised substitution when building the theory of prospective values for the purpose of this PhD, but clearly frames yield a more complete and general theory of generalised substitutions more discriminating than B’s theory of the GSL.
6.4.1 Proof of the Algebraic Laws

In this section we will as indicated prove the correctness of each of the previously introduced rewrite laws with respect to the closed-form definition Def. 34.

For some laws this may seem a tedious or even redundant exercise i.e. with the law being intuitively evident, however the importance of this step is to increase our confidence in the pv calculus by showing that the two views, the closed-form pv definition and the bottom-up approach via rewrite rules yield to the same theory, or at least don’t result in logical inconsistency. A welcome side-effect of this exercise is moreover that it extensively illustrates the application of Improper Bunch Theory.

Having once proved each of the propositions Prop. 35 to Prop. 42 we may effectively discard Def. 34 for the sake of evaluating $S \diamond E$ as the rewrite laws make this a much more efficient process.

**Skip.** Prop. 35 is $\text{skip} \diamond E = E$.

**Proof.**

\[
\text{skip} \diamond E
\]

\[= \quad \text{"Defn of } S \diamond E \text{ (Def. 34) with frame(\text{skip}) = } \emptyset"
\]

\[
\text{trm(\text{skip}) | } \not\emptyset \bullet \text{prd(\text{skip}) } \longrightarrow E[\emptyset \emptyset]
\]

\[= \quad \text{"Degenerate case of substitution: } E[\emptyset \emptyset] = E"
\]

\[
\text{trm(\text{skip}) | } \not\emptyset \bullet \text{prd(\text{skip}) } \longrightarrow E
\]

\[= \quad \text{"Degenerate case of bunch comprehension: } \not\emptyset \bullet E = E"
\]

\[
\text{trm(\text{skip}) | } \text{prd(\text{skip}) } \longrightarrow E
\]

\[= \quad \text{"Rewriting trm(\text{skip}) and prd(\text{skip}), see Fig. 6.2"}
\]

\[\text{true | true } \longrightarrow E
\]

\[= \quad \text{"Simplifying pre-conditioned and guarded bunch"}
\]

\[E \quad \square.
\]

**Assignment.** Prop. 36 is $z := F \diamond E = E[z \setminus F]$.

**Proof.**

\[
z := F \diamond E
\]

\[= \quad \text{"Defn of } S \diamond E \text{ (Def. 34) with frame}(z := F) = z"
\]

\[
\text{trm}(x := F) | \not\emptyset z' \bullet \text{prd}(z := F) \longrightarrow E[z \setminus z']
\]

\[= \quad \text{"Rewriting trm}(z := F) \text{ and prd}(z := F), see Fig. 6.2"}
\]

\[\text{true | } \not\emptyset z' \bullet z' = F \longrightarrow E[z \setminus z']
\]

\[= \quad \text{"Simplifying pre-conditioned bunch"}
\]

\[\not\emptyset z' \bullet z' = F \longrightarrow E[z \setminus z']
\]
CHAPTER 6. PROSPECTIVE-VALUE SEMANTICS

= “One-point rule for bunch comprehension, note that $F$ is elementary$^4$”
  $E[z\backslash F] \square$.

Precondition. Prop. 37 is $P \mid S \odot E = P \mid S \circ E$.

Proof. $P \mid S \odot E$

= “Defn of $S \odot E$ (Def. 34) with frame($P \mid S$) = $s$”
  $\text{trm}(P \mid S) \mid \{ s' \bullet \text{prd}(P \mid S) \rightarrow E[s\backslash s']$

= “Rewriting $\text{trm}(P \mid S)$ and $\text{prd}(P \mid S)$, see Fig. 6.2”
  $P \land \text{trm}(S) \mid \{ s' \bullet P \Rightarrow \text{prd}(S) \rightarrow E[s\backslash s']$

= “Logic”
  $P \land \text{trm}(S) \mid \{ s' \bullet \lnot P \land \text{prd}(S) \rightarrow E[s\backslash s']$

= “Splitting bunch comprehension (Prop. 28)”
  $P \land \text{trm}(S) \mid \{ s' \bullet \rightarrow P \land \text{prd}(S) \rightarrow E[s\backslash s']$

= “Bunch comprehension slide rule (Prop. 26), note that $s' \backslash P$”
  $P \land \text{trm}(S) \mid \rightarrow P \rightarrow E[s\backslash s'], \{ s' \bullet \text{prd}(S) \rightarrow E[s\backslash s']$

= “Bunch law: $P \mid \rightarrow P \rightarrow E, F = P \mid F$ for $\rightarrow P \rightarrow E$ never contributes”
  $P \land \text{trm}(S) \mid \{ s' \bullet \text{prd}(S) \rightarrow E[s\backslash s']$

= “Chaining law for pre-conditioned bunch (Prop. 24)”
  $P \mid \text{trm}(S) \mid \{ s' \bullet \text{prd}(S) \rightarrow E[s\backslash s']$

= “Defn of $S \circ E$ (Def. 34)”
  $P \mid S \circ E \square$.

Guard. Prop. 38 is $P \implies S \circ E = P \implies S \circ E$.

Proof. $P \implies S \circ E$

= “Defn of $S \circ E$ (Def. 34) with frame($P \implies S$) = $s$”
  $\text{trm}(P \implies S) \mid \{ s' \bullet \text{prd}(P \implies S) \rightarrow E[s\backslash s']$

= “Rewriting $\text{trm}(P \implies S)$ and $\text{prd}(P \implies S)$, see Fig. 6.2”
  $P \Rightarrow \text{trm}(S) \mid \{ s' \bullet P \land \text{prd}(S) \rightarrow E[s\backslash s']$

= “Chaining law for guarded bunch (Prop. 23)”

$^4$Elementhood is essential for the one-point rule to be applied in this form (Prop. 32). This is guaranteed since we currently don’t extend the syntax of the GSL to allow for bunch expressions in general, i.e. on the right hand of an assignment. Bunches may occur only packaged within the command language as the content of some set, and if necessary we are able protect ourselves from expressions such as $\{ \bot \} = \bot_{P(\ldots)}$ through suitable proof obligations.
CHAPTER 6. PROSPECTIVE-VALUE SEMANTICS

\[ P \Rightarrow \text{trm}(S) \mid \S s' \bullet P \rightarrow \text{prd}(S) \rightarrow E[s\backslash s'] \]

\[ \text{“Bunch comprehension slide rule (Prop. 26), note that } s' \backslash P \text{”} \]
\[ P \Rightarrow \text{trm}(S) \mid P \rightarrow ((\S s' \bullet \text{prd}(S) \rightarrow E[s\backslash s']) \]

\[ \text{“Defn of pre-conditioned bunch”} \]
\[ P \rightarrow ((\S s' \bullet \text{prd}(S) \rightarrow E[s\backslash s']), \neg(P \Rightarrow \text{trm}(S)) \rightarrow \bot) \]

\[ \text{“Logic (de Morgan)”} \]
\[ P \rightarrow ((\S s' \bullet \text{prd}(S) \rightarrow E[s\backslash s']), P \land \neg\text{trm}(S) \rightarrow \bot) \]

\[ \text{“Chaining law for guarded bunch (Prop. 23)”} \]
\[ P \rightarrow ((\S s' \bullet \text{prd}(S) \rightarrow E[s\backslash s']), \neg\text{trm}(S) \rightarrow \bot) \]

\[ \text{“Defn of pre-conditioned bunch”} \]
\[ P \rightarrow (\text{trm}(S) \mid \S s' \bullet \text{prd}(S) \rightarrow E[s\backslash s']) \]

\[ \text{“Defn of } S \circ E \text{ (Def. 34)”} \]
\[ P \rightarrow S \circ E \quad \square. \]

Choice. Prop. 39 is \( S \parallel T \circ E = S \circ E, T \circ E \).

Proof. \( S \parallel T \circ E \)

\[ \text{“Defn of } S \circ E \text{ (Def. 34) with frame}(S \parallel T) = u \text{ and } u = s \cup t” \]
\[ \text{trm}(S \parallel T) \mid \S u' \bullet \text{prd}(S \parallel T) \rightarrow E[u \backslash u'] \]

\[ \text{“Property of frames: } \text{prd}(S \parallel T) = \text{prd}(S_t \parallel T_s)” \]
\[ \text{trm}(S \parallel T) \mid \S u' \bullet \text{prd}(S_t \parallel T_s) \rightarrow E[u \backslash u'] \]

\[ \text{“Rewriting } \text{trm}(S \parallel T) \text{ and } \text{prd}(S_t \parallel T_s), \text{ see Fig. 6.2”} \]
\[ \text{trm}(S) \land \text{trm}(T) \mid \S u' \bullet \text{prd}(S_t) \lor \text{prd}(T_s) \rightarrow E[u \backslash u'] \]

\[ \text{“Splitting bunch comprehension (Prop. 28)”} \]
\[ \text{trm}(S) \land \text{trm}(T) \mid (\S u' \bullet \text{prd}(S_t) \rightarrow E[u \backslash u']), \S u' \bullet \text{prd}(T_s) \rightarrow E[u \backslash u'] \]

\[ \text{“Defn of pre-conditioned bunch”} \]
\[ \neg(\text{trm}(S) \land \text{trm}(T)) \rightarrow \bot, \]
\[ (\S u' \bullet \text{prd}(S_t) \rightarrow E[u \backslash u']), (\S u' \bullet \text{prd}(T_s) \rightarrow E[u \backslash u']) \]

\[ \text{“Logic (de Morgan)”} \]
\[ \neg\text{trm}(S) \lor \neg\text{trm}(T) \rightarrow \bot, \]
\[ (\S u' \bullet \text{prd}(S_t) \rightarrow E[u \backslash u']), (\S u' \bullet \text{prd}(T_s) \rightarrow E[u \backslash u']) \]

\[ \text{“Expanding of pre-conditioned bunch (Prop. 22)”} \]
\[
\neg \text{trm}(S) \rightarrow \bot, \neg \text{trm}(T) \rightarrow \bot, \\
(\forall u' \bullet \text{prd}(S_t) \rightarrow E[u \setminus u']), (\forall u' \bullet \text{prd}(T_s) \rightarrow E[u \setminus u'])
\]

\[= \text{”Defn of pre-conditioned bunch after re-ordering terms“} \]

\[
(\text{trm}(S) \mid \forall u' \bullet \text{prd}(S_t) \rightarrow E[u \setminus u']), \text{trm}(T) \mid \forall u' \bullet \text{prd}(T_s) \rightarrow E[u \setminus u']
\]

\[= \text{”Frame extension law for trm, see Fig. 6.2“} \]

\[
(\text{trm}(S_t) \mid \forall u' \bullet \text{prd}(S_t) \rightarrow E[u \setminus u']), \text{trm}(T_s) \mid \forall u' \bullet \text{prd}(T_s) \rightarrow E[u \setminus u']
\]

\[= \text{”Defn of } S \odot E \text{ (Def. 34) applied to } S_t \text{ and } T_s“ \]

\[S_t \odot E, T_s \odot E \]

\[= \text{”Frame extension law for prospective values, see Fig. 6.3“} \]

\[S \odot E, T \odot E \quad \square.\]

**Sequential Composition.** Prop. 40 is \( S \odot T \odot E = S \odot T \odot E. \)

\begin{proof}
\[ S \odot T \odot E \]

\[= \text{”Defn of } S \odot E \text{ (Def. 34) with } \text{frame}(S ; T) = u \text{ where } u = s \cup t“ \]

\[\text{trm}(S ; T) \mid \forall u' \bullet \text{prd}(S ; T) \rightarrow E[u \setminus u'] \]

\[= \text{”Rewriting trm}(S ; T) \text{ and } \text{prd}(S ; T), \text{see Fig. 6.2“} \]

\[wp(S, \text{trm}(T)) \mid \forall u' \bullet \text{trm}(S) \Rightarrow (\text{prd}(S_t) \odot \text{prd}(T_s)) \rightarrow E[u \setminus u'] \]

\[= \text{”Frame extension law for wp: } wp(S, Q) = wp(S_y, Q)“ \]

\[wp(S_t, \text{trm}(T)) \mid \forall u' \bullet \text{trm}(S) \Rightarrow (\text{prd}(S_t) \odot \text{prd}(T_s)) \rightarrow E[u \setminus u'] \]

\[= \text{”Defn of pre-conditioned bunch“} \]

\[\neg wp(S_t, \text{trm}(T)) \rightarrow \bot, \forall u' \bullet \text{trm}(S) \Rightarrow (\text{prd}(S_t) \odot \text{prd}(T_s)) \rightarrow E[u \setminus u'] \]

\[= \text{”Let } X \text{ and } Y \text{ be the left and right term of the bunch union“} \]

\[X, Y \]

In order to render the proof, which is in fact the most ambitious of the thesis, more readable, we will from hereon rewrite both terms \( X \) and \( Y \) individually. Once they have been sufficiently simplified we will join them together again for the sake of completing the proof. We start by simplifying \( X \):

\[X = \neg wp(S_t, \text{trm}(T)) \rightarrow \bot \]

\[= \text{”Correspondence between wp and the characteristic predicates trm and prd: } \]

\[wp(S, Q) = \text{trm}(S) \land \forall s' \bullet \text{prd}(S) \Rightarrow Q[s \setminus s']“ \]

\[\neg (\text{trm}(S_t) \land \forall u' \bullet \text{prd}(S_t) \Rightarrow \text{trm}(T)[u \setminus u']) \rightarrow \bot \]

\[= \text{”Logic (de Morgan)“} \]

\[\neg \text{trm}(S_t) \lor (\neg \forall u' \bullet \text{prd}(S_t) \Rightarrow \text{trm}(T)[u \setminus u']) \rightarrow \bot \]

\end{proof}
We shall make a note of the intermediate result \( \tilde{X} \) and continue by simplifying.

**CHAPTER 6. PROSPECTIVE-VALUE SEMANTICS**

86

= “Expansion law for guarded bunch with disjunction (Prop. 20)”
\[ \neg \text{trm}(S_i) \rightarrow \bot, (\neg \forall u' \bullet \text{prd}(S_i) \Rightarrow \text{trm}(T)[u \setminus u']) \rightarrow \bot \]

= “Logic: \( \forall z \bullet Q = \neg \exists z \bullet \neg Q \)”
\[ \neg \text{trm}(S_i) \rightarrow \bot, (\exists u' \bullet \neg(\text{prd}(S_i) \Rightarrow \text{trm}(T)[u \setminus u'])) \rightarrow \bot \]

= “Logic (de Morgan)”
\[ \neg \text{trm}(S_i) \rightarrow \bot, (\exists u' \bullet \text{prd}(S_i) \land \neg \text{trm}(T)[u \setminus u']) \rightarrow \bot \]

= “Bunch comprehension elimination law (Prop. 30), note that \( u' \setminus \bot \)”
\[ \neg \text{trm}(S_i) \rightarrow \bot, \# u' \bullet \text{prd}(S_i) \land \neg \text{trm}(T)[u \setminus u'] \rightarrow \bot \]

= “Chaining law for guarded bunch (Prop. 23)”
\[ \neg \text{trm}(S_i) \rightarrow \bot, \# u' \bullet \text{prd}(S_i) \rightarrow \neg \text{trm}(T)[u \setminus u'] \rightarrow \bot \]

= “Let \( \tilde{X} \) be the intermediate result for \( X \)”
\( \tilde{X} \)

We shall make a note of the intermediate result \( \tilde{X} \) obtained for \( X \) at this point, and continue by simplifying \( Y \):

\[ Y = \# u' \bullet \text{trm}(S) \Rightarrow (\text{prd}(S_i) \nmid \text{prd}(T_s)) \rightarrow E[u \setminus u'] \]

= “Frame extension law for \text{trm}, see Fig. 6.2”
\[ \# u' \bullet \text{trm}(S_i) \Rightarrow (\text{prd}(S_i) \nmid \text{prd}(T_s)) \rightarrow E[u \setminus u'] \]

= “Logic”
\[ \# u' \bullet \neg \text{trm}(S_i) \lor (\text{prd}(S_i) \nmid \text{prd}(T_s)) \rightarrow E[u \setminus u'] \]

= “Splitting bunch comprehension (Prop. 28)”
\[ (\# u' \bullet \neg \text{trm}(S_i) \rightarrow E[u \setminus u']), \# u' \bullet \text{prd}(S_i) \nmid \text{prd}(T_s) \rightarrow E[u \setminus u'] \]

= “Bunch comprehension slide rule (Prop. 26), note that \( u' \setminus \text{trm}(S_i) \)”
\[ \neg \text{trm}(S_i) \rightarrow (\# u' \bullet E[u \setminus u']), \# u' \bullet \text{prd}(S_i) \nmid \text{prd}(T_s) \rightarrow E[u \setminus u'] \]

= “Renaming quantified variable \( u' \), possible since \( u' \setminus E \)”
\[ \neg \text{trm}(S_i) \rightarrow (\# u \bullet E), \# u' \bullet \text{prd}(S_i) \nmid \text{prd}(T_s) \rightarrow E[u \setminus u'] \]

= “Defn of relational composition, note that \( u'' \) is fresh here”
\[ \neg \text{trm}(S_i) \rightarrow (\# u \bullet E), \# u' \bullet (\exists u'' \bullet \text{prd}(S_i)[u' \setminus u''] \land \text{prd}(T_s)[u \setminus u'']) \rightarrow E[u \setminus u'] \]

= “Bunch comprehension elimination law (Prop. 30) since \( u'' \setminus E[u \setminus u'] \)”
\[ \neg \text{trm}(S_i) \rightarrow (\# u \bullet E), \# u', u'' \bullet \text{prd}(S_i)[u' \setminus u''] \land \text{prd}(T_s)[u \setminus u''] \rightarrow E[u \setminus u'] \]

= “Bunch comprehension with multiple variables (commutativity)”
\[ \neg \text{trm}(S_i) \rightarrow (\# u \bullet E), \]
§ $u'' \cdot § u' \cdot \text{prd}(S_t)[u' \backslash u''] \land \text{prd}(T_s)[u' \backslash u''] \rightarrow E[u' \backslash u']$

= "Chaining law for guarded bunch (Prop. 23)"
$\neg \text{trm}(S) \rightarrow (§ u \bullet E)$,
§ $u'' \cdot § u' \cdot \text{prd}(S_t)[u' \backslash u''] \rightarrow \text{prd}(T_s)[u' \backslash u''] \rightarrow E[u' \backslash u']$

= "Bunch comprehension slide rule (Prop. 26), note that $u' \backslash \text{prd}(S_t)[u' \backslash u'']$"
$\neg \text{trm}(S) \rightarrow (§ u \bullet E)$,
§ $u'' \cdot \text{prd}(S_t)[u' \backslash u''] \rightarrow§ u' \cdot \text{prd}(T_s)[u' \backslash u''] \rightarrow E[u' \backslash u']$

= "Renaming quantified variable $u'$ into $u''$, okay since $u''$ is fresh"
$\neg \text{trm}(S) \rightarrow (§ u \bullet E)$,
§ $u'' \cdot \text{prd}(S_t)[u' \backslash u''] \rightarrow § u' \cdot \text{prd}(T_s)[u' \backslash u''][u' \backslash u''] \rightarrow E[u' \backslash u'][u' \backslash u''][u' \backslash u']$

= "Simplification of substitution, using moreover that $u' \backslash E"$
$\neg \text{trm}(S) \rightarrow (§ u \bullet E)$,
§ $u'' \cdot \text{prd}(S_t)[u' \backslash u''] \rightarrow § u'' \cdot \text{prd}(T_s)[u, u' \backslash u'', u''] \rightarrow E[u'u''']$

= "Renaming quantified variable $u''$ into $u'$, possible since $u'$ is neither free in
\text{prd}(S_t)[u' \backslash u'']$, nor in \text{prd}(T_s)[u, u' \backslash u'', u''], and clearly not in $E"$
$\neg \text{trm}(S) \rightarrow (§ u \bullet E)$,
§ $u'' \cdot \text{prd}(S_t)[u' \backslash u''][u' \backslash u'] \rightarrow § u'' \cdot \text{prd}(T_s)[u, u' \backslash u'', u''][u' \backslash u'] \rightarrow E[u'\backslash u'][u'\backslash u'][u'\backslash u']$

= "Simplification of substitution, note that $u'' \backslash E"$
$\neg \text{trm}(S) \rightarrow (§ u \bullet E)$,
§ $u'' \cdot \text{prd}(S_t) \rightarrow§ u'' \cdot \text{prd}(T_s)[u, u' \backslash u', u''] \rightarrow E[u' \backslash u'']$

= "Rewrite substitution: $\text{prd}(T_s)[u, u' \backslash u', u''] = \text{prd}(T_s)[u' \backslash u'][u' \backslash u']"$
$\neg \text{trm}(S) \rightarrow (§ u \bullet E)$,
§ $u'' \cdot \text{prd}(S_t) \rightarrow§ u'' \cdot \text{prd}(T_s)[u' \backslash u'[u' \backslash u'] \rightarrow E[u' \backslash u'][u' \backslash u']$

= "Substitution identity: $E[u' \backslash u''] = E[u' \backslash u'][u' \backslash u']$ since clearly $u' \backslash E[u' \backslash u'']"$
$\neg \text{trm}(S) \rightarrow (§ u \bullet E)$,
§ $u'' \cdot \text{prd}(S_t) \rightarrow§ u'' \cdot \text{prd}(T_s)[u' \backslash u'[u' \backslash u'] \rightarrow E[u' \backslash u'][u' \backslash u']$

= "Distributivity of the substitution $u' \backslash u'$ through the 2nd bunch comprehension"
$\neg \text{trm}(S) \rightarrow (§ u \bullet E)$,
§ $u'' \cdot \text{prd}(S_t) \rightarrow (§ u'' \cdot \text{prd}(T_s)[u' \backslash u''] \rightarrow E[u' \backslash u''][u' \backslash u']$

Note that the last step is only valid because the variables in $u$ and $u'$ occurring
in the substitution $u' \backslash u'$ are disjoint with the quantified variables in the list
$u''$.

= "Renaming of quantified variable $u''$ back to $u'$, safe since $u' \backslash E"
The previous result looks already very promising in terms of applying the pv
definition (Def. 34) in a reverse manner. To eventually complete the proof we
rejoin \( \hat{X} \) and \( \hat{Y} \) again in a bunch union, and perform a few more simplification
steps.

\( \hat{X}, \hat{Y} \)

\[\begin{align*}
-\text{trm}(S) & \rightarrow (\exists \, u \bullet E), \\
\exists \, u' \bullet \text{prd}(S_1) & \rightarrow (\exists \, u' \bullet \text{prd}(T_s) \rightarrow E[u\backslash u'])[u\backslash u'] \\
= \text{"Let } \hat{Y} \text{ be the intermediate result for } Y" \\
\end{align*}\]

\( \text{Let } \hat{Y} \) be the intermediate result for \( Y \)

\[\begin{align*}
\text{Intermediate results obtained from simplifying } X \text{ and } Y \\
-\text{trm}(S_1) & \rightarrow \bot, (\exists \, u' \bullet \text{prd}(S_1) \rightarrow -\text{trm}(T)[u\backslash u'] \rightarrow \bot), \\
-\text{trm}(S) & \rightarrow (\exists \, u \bullet E), \\
\exists \, u' \bullet \text{prd}(S_1) & \rightarrow (\exists \, u' \bullet \text{prd}(T_s) \rightarrow E[u\backslash u'])[u\backslash u'] \\
= \text{"Frame extension law for trm, see Fig. 6.2"} \\
-\text{trm}(S_1) & \rightarrow \bot, (\exists \, u' \bullet \text{prd}(S_1) \rightarrow -\text{trm}(T)[u\backslash u'] \rightarrow \bot), \\
-\text{trm}(S_1) & \rightarrow (\exists \, u \bullet E), \\
\exists \, u' \bullet \text{prd}(S_1) & \rightarrow (\exists \, u' \bullet \text{prd}(T_s) \rightarrow E[u\backslash u'])[u\backslash u'] \\
= \text{"Distributivity of bunch guarding (Prop. 15)"} \\
-\text{trm}(S_1) & \rightarrow (\bot, (\exists \, u \bullet E), \\
(\exists \, u' \bullet \text{prd}(S_1) & \rightarrow -\text{trm}(T)[u\backslash u'] \rightarrow \bot), \\
\exists \, u' \bullet \text{prd}(S_1) & \rightarrow (\exists \, u' \bullet \text{prd}(T_s) \rightarrow E[u\backslash u'])[u\backslash u'] \\
= \text{"Absorptive power of the improper bunch"} \\
-\text{trm}(S_1) & \rightarrow \bot, (\exists \, u' \bullet \text{prd}(S_1) \rightarrow -\text{trm}(T)[u\backslash u'] \rightarrow \bot), \\
\exists \, u' \bullet \text{prd}(S_1) & \rightarrow (\exists \, u' \bullet \text{prd}(T_s) \rightarrow E[u\backslash u'])[u\backslash u'] \\
= \text{"Defn of pre-conditioned bunch"} \\
\text{trm}(S_1) & \mid (\exists \, u' \bullet \text{prd}(S_1) \rightarrow -\text{trm}(T)[u\backslash u'] \rightarrow \bot), \\
\exists \, u' \bullet \text{prd}(S_1) & \rightarrow (\exists \, u' \bullet \text{prd}(T_s) \rightarrow E[u\backslash u'])[u\backslash u'] \\
= \text{"Distributivity of bunch comprehension (Prop. 25)"} \\
\text{trm}(S_1) & \mid (\exists \, u' \bullet \\
\text{prd}(S_1) & \rightarrow -\text{trm}(T)[u\backslash u'] \rightarrow \bot), \\
\text{prd}(S_1) & \rightarrow (\exists \, u' \bullet \text{prd}(T_s) \rightarrow E[u\backslash u'])[u\backslash u'] \\
= \text{"Distributivity of bunch guarding (Prop. 15)"} \\
\text{trm}(S_1) & \mid (\exists \, u' \bullet \\
\text{prd}(S_1) & \rightarrow (\neg \text{trm}(T)[u\backslash u'] \rightarrow \bot, (\exists \, u' \bullet \text{prd}(T_s) \rightarrow E[u\backslash u'])[u\backslash u'] )
CHAPTER 6. PROSPECTIVE-VALUE SEMANTICS

Unbounded Choice. Prop. 41 is \( \exists z \mid S \circ E = \exists z \mid S \circ E \) providing \( z \setminus E \).

Proof. \( \exists z \mid S \circ E \)

\[ = \quad \text{“Distributivity of substitution”} \]

\[ \text{trm}(S_t) \mid \exists u' \bullet \]

\[ \text{prd}(S_t) \rightarrow (\neg \text{trm}(T) \rightarrow \bot, \exists u' \bullet \text{prd}(T_s) \rightarrow E[u \setminus u'])[u \setminus u'] \]

\[ = \quad \text{“Defn of pre-conditioned bunch”} \]

\[ \text{trm}(S_t) \mid \exists u' \bullet \text{prd}(S_t) \rightarrow (\text{trm}(T) \mid \exists u' \bullet \text{prd}(T_s) \rightarrow E[u \setminus u'])[u \setminus u'] \]

\[ = \quad \text{“Frame extension law for trm, see Fig. 6.2”} \]

\[ \text{trm}(S_t) \mid \exists u' \bullet \text{prd}(S_t) \rightarrow (\text{trm}(T_s) \mid \exists u' \bullet \text{prd}(T_s) \rightarrow E[u \setminus u'])[u \setminus u'] \]

\[ = \quad \text{“Defn of } S \circ E \text{ (Def. 34) applied to } T_s” \]

\[ \text{trm}(S_t) \mid \exists u' \bullet \text{prd}(S_t) \rightarrow (T_s \circ E)[u \setminus u'] \]

\[ = \quad \text{“Defn of } S \circ E \text{ (Def. 34) applied to } S_t” \]

\[ S_t \circ T \circ E \quad \square. \]

\[ \text{Unbounded Choice. Prop. 41 is } @z \bullet S \circ E = \exists z \bullet S \circ E \text{ providing } z \setminus E. \]
(§z • ¬trm(S) −→ ⊥), §z • §s′ • prd(S) −→ E[\mathcal{s}\setminus s']

= “Distributivity of bunch comprehension (Prop. 25)”

§z • ¬trm(S) −→ ⊥, §s′ • prd(S) −→ E[\mathcal{s}\setminus s']

= “Defn of pre-conditioned bunch”

§z • trm(S) | §s′ • prd(S) −→ E[\mathcal{s}\setminus s']

= “Defn of \mathcal{S} \circ E (Def. 34)”

§z • S \circ E  □.

Frame Extension. Prop. 42 is \mathcal{S}_y \circ E = \mathcal{S} \circ E.

Proof. Note that in the following proof we assume \( y \) is not contained in the frame of \( S \). If such would be the case we may immediately conclude that \( S = S_y \), hence prd(S) = prd(S_y) and trm(S) = trm(S_y). Trivially this makes Prop. 42 true.

\( \mathcal{S}_y \circ E \)

= “\( \mathcal{S} \circ E \) (Def. 34) with frame(\( S_y \)) = \mathcal{s}, y \) assuming \( s \cap y = \emptyset \)”

\( \text{trm}(S_y) \mid \mathcal{s}, y' \cdot \text{prd}(S_y) \rightarrow E[s, y\setminus s', y'] \)

= “Rewriting \( \text{trm}(S_y) \) and \( \text{prd}(S_y) \), see Fig. 6.2”

\( \text{trm}(S) \mid \mathcal{s}, y' \cdot \text{prd}(S) \land y = y' \rightarrow E[s, y\setminus s', y'] \)

= “Chaining law for guarded bunch (Prop. 23)”

\( \text{trm}(S) \mid \mathcal{s}, y' \cdot y = y' \rightarrow \text{prd}(S) \rightarrow E[s, y\setminus s', y'] \)

= “Bunch comprehension with multiple variables”

\( \text{trm}(S) \mid \mathcal{s}, y' \cdot \mathcal{s}, y = y' \rightarrow \text{prd}(S) \rightarrow E[s, y\setminus s', y'] \)

= “Bunch comprehension slide rule (Prop. 26), note that \( s' \setminus y = y'' \)”

\( \text{trm}(S) \mid \mathcal{s}, y' \cdot y = y' \rightarrow \mathcal{s}, y' \cdot \text{prd}(S) \rightarrow E[s, y\setminus s', y'] \)

= “One-point rule for bunch comprehension (Prop. 32)”

\( \text{trm}(S) \mid \mathcal{s}, y' \cdot \text{prd}(S) \rightarrow E[s, y\setminus s', y'][y'' \setminus y] \)

= “Distributivity of substitution”

\( \text{trm}(S) \mid \mathcal{s}, y' \cdot \text{prd}(S)[y'/y] \rightarrow E[s, y\setminus s', y'][y'' \setminus y] \)

= “Simplification of substitution, \( \text{prd}(S) \) and \( E \) don’t mention \( y'' \)”

\( \text{trm}(S) \mid \mathcal{s}, y' \cdot \text{prd}(S) \rightarrow E[s\setminus s'] \)

= “Defn of \( \mathcal{S} \circ E \) (Def. 34)”

\( \mathcal{S} \circ E  □. \)
6.5 Characteristic Predicates

The fundamental definition for $S \circ E$, which we introduced at the beginning of this chapter, describes the prospective value of any computation $S$ by relating it to the characteristic predicates of $S$, namely the frame($S$), trm($S$) and prd($S$). These predicates are familiar and well-established concepts in the world of B and its underlying Generalised Substitution Language. We have an intuitive understanding of them, and if necessary are able to derive them, for example, from the wp predicate transformer of some given GSL computation.

In what follows in this section we will present and prove a few corollaries which vice versa allow us to infer the characteristic predicates of a computation from its pv expression transformer. Whereas Def. 34 establishes a semantic link by mapping the semantic domain of ‘characteristic predicates’ onto that of expression transformers — more accurately the ones representing valid GSL computations which we call pv transformers — the corollaries here show that this mapping can be inverted by effectively retrieving the original semantics of a computation (i.e. in terms of characteristic predicates) from an arbitrary pv transformer. The fact that trm and prd can be retrieved in such a way from the prospective-value operator confirms that Def. 34 indeed establishes an isomorphic relationship between the original semantic domain, and the one of prospective-value expression transformers; in other words, we don’t lose any expressive power when reasoning about GSL computations in terms of their prospective value as compared to, for example, carrying out the same reasoning in wp semantics.

A further notable observation is that the subsequent corollaries seem no less concise or complicated than the corresponding formulations of them in wp semantics; see Fig. 6.4 (in the case of prd the pv formulation even appears to be simpler and immediately more suggestive).

**Corollary 43.** Given the pv transformer of any GSL computation $S$ we can retrieve the trm of $S$ as follows:

$$\text{trm}(S) = S \circ \text{null} : \text{null}$$

**Proof.**

1. $S \circ \text{null} : \text{null}$

2. “Defn of $S \circ E$ (Def. 34) with frame($S$) = $s$”

   $$\text{trm}(S) \models \frac{\text{null}[x \leftarrow E]}{\text{null} : \text{null}}$$

3. “Simplification of substitution: $\text{null}[x \leftarrow E] = \text{null}”$$

   $$\text{trm}(S) \models \frac{\text{null} : \text{null}}{\text{null} : \text{null}}$$

4. “Simplification of guarded bunch: $P \rightarrow \text{null} = \text{null}”$$

   $$\text{trm}(S) \models \frac{\text{null} : \text{null}}{\text{null} : \text{null}}$$

5. “Simplification of bunch comprehension: $\exists z \cdot \text{null} = \text{null}”$$

   $$\text{trm}(S) \models \frac{\text{null} : \text{null}}{\text{null} : \text{null}}$$
“Logic, the above is true exactly if \( \text{trm}(S) \), namely \( \neg \text{trm}(S) \) implies that
\( \text{trm}(S) \models \text{null} = \bot \) and clearly \( \neg \bot : \text{null} \)
\( \text{trm}(S) \square. \)

**Remark.** Operationally we can understand Corollary 43 by acknowledging that only non-terminating behaviour can transform the value of \( \text{null} \) i.e. by increasing it to \( \bot \).

**Corollary 44.** Given the pv transformer of any GSL computation \( S \) we can retrieve the fis of \( S \) as follows:

\[
\text{fis}(S) = \bot : S \circ \bot
\]

**Proof.**
\( \bot : S \circ \bot \)

\[
\begin{align*}
&= \text{Defn of } S \circ E \text{ (Def. 34) with } \text{frame}(S) = s' \\
&= \bot : \text{trm}(S) \mid \# s' \bullet \text{prd}(S) \rightarrow \bot[s'\backslash s']
\end{align*}
\]

\[
\begin{align*}
&= \text{Simplification of substitution: } \bot[x\backslash E] = \bot \\
&= \bot : \text{trm}(S) \mid \# s' \bullet \text{prd}(S) \rightarrow \bot
\end{align*}
\]

\[
\begin{align*}
&= \text{Bunch comprehension elimination law (Prop. 30), since } s' \bot \\
&= \bot : \text{trm}(S) \mid (\exists s' \bullet \text{prd}(S)) \rightarrow \bot
\end{align*}
\]

\[
\begin{align*}
&= \text{Defn of pre-conditioned bunch} \\
&= \bot : \neg \text{trm}(S) \rightarrow \bot, (\exists s' \bullet \text{prd}(S)) \rightarrow \bot
\end{align*}
\]

\[
\begin{align*}
&= \text{Expansion law for guarded bunch (Prop. 20)} \\
&= \bot : \neg \text{trm}(S) \lor (\exists s' \bullet \text{prd}(S)) \rightarrow \bot
\end{align*}
\]

\[
\begin{align*}
&= \text{Logic, } \bot : P \rightarrow \bot \text{ is trivially equivalent to } P \\
&= \neg \text{trm}(S) \lor \exists s' \bullet \text{prd}(S)
\end{align*}
\]

\[
\begin{align*}
&= \text{Logic} \\
&= \text{trm}(S) \Rightarrow \exists s' \bullet \text{prd}(S)
\end{align*}
\]

\[
\begin{align*}
&= \text{Property of trm and prd: } \neg \text{trm}(S) \Rightarrow \text{prd}(S), \text{ this is Prop. 2.1 in } [\text{Dun02}] \\
&= \exists s' \bullet \text{prd}(S)
\end{align*}
\]

\[
\begin{align*}
&= \exists s' \bullet \text{prd}(S) \text{ is an alternative characterisation for the feasibility of } S \text{ since } S \text{ is feasible exactly where it has some behaviour} \\
&= \text{fis}(S) \square.
\end{align*}
\]

**Remark.** Operationally we can understand Corollary 44 by acknowledging that only infeasibility can transform the value of \( \bot \) i.e. by reducing it to \( \text{null} \).
### 6.6 Connection to wp Semantics

So far the focus in developing the prospective-value calculus was mainly on relating it to the characteristic predicates of a generalised substitutions. Though we are able to use

Theorem 46.

\[
\text{wp}(S, Q) = \text{trm}(S) \land \forall s' \cdot \text{prd}(S) \Rightarrow Q[s \backslash s']
\]

to recover the wp effect of a computation being specified in terms of its frame, \text{trm} and \text{prd}, it could still be desirable and useful to have a direct link between pv
and wp semantics not involving the characteristic predicates trm and prd. Such
would moreover nicely conclude our account given in this chapter examining the
relationship of pv semantics and other semantic models for the GSL.

We first present a proposition which allows us to determine the wp effect of a
computation $S$ directly from its prospective-value expression transformer.

**Proposition 47.** Let $S$ be a generalised substitution having the frame $s$, and $Q$
some arbitrary predicate. Then the wp effect of $S$ can be calculated from its pv
transformer using

$$wp(S, Q) = S \circ \text{bool}(Q) : TRUE$$

Note that $\text{bool}$ is the operator which in B-AMN maps predicates to their corre-
sponding truth value of type $BOOL$. We can define it in Improper Bunch Theory
nicely through

**Definition 48.**

$$\text{bool}(Q) = \text{df} Q \rightarrow TRUE, \neg Q \rightarrow FALSE$$

Importantly, realise that Def. 48 is **not** defining a function $\text{bool}(\ldots)$ but a genuine
operator because its argument is not a value in the sense of B (i.e. having a type),
but a predicate\(^5\).

The informal justification of Prop. 47 is as follows: because the weakest pre-
condition of $S$ to establish $Q$ describes exactly those states from which we guar-
antee to satisfy $Q$ if executing $S$, it is fulfilled exactly if the prospective value of
$Q$ is either equal to $true$ (all behaviours of $S$ establish $Q$), or alternatively $null$
($S$ is infeasible, and $Q$ is vacuously established by miracle).

The reader at this point might wonder why we had to incorporate the $\text{bool}$
operator, and could not just have employed $S \circ Q : true$ as the right hand in
Prop. 47. The reason for this is simply that $S \circ Q$ is not a meaningful term in
our theory of bunches since, as explained in in Section 5.3.3, we strictly pro-
hibit bunch values from infiltrating the underlying predicate calculus, whereby
effectively avoiding the complications of a multi-valued logic.

What remains to be carried out is a formal proof of Prop. 47. Fortunately this
is not difficult:

**Proof.** Prop. 47 is $wp(S, Q) = S \circ \text{bool}(Q) : TRUE$

$$S \circ \text{bool}(Q) : TRUE$$

$$= \text{"Defn of } S \circ E \text{ (Def. 34) with frame}(S) = s\text{"}$$

$$\text{trm}(S) \mid \frac{s'}{s} \bullet \text{prd}(S) \rightarrow \text{bool}(Q)[s \backslash s'] : TRUE$$

\(^5\)Predicates in the B Method are not, as in certain theories of logic, identified with Boolean
expressions. For the purpose of Boolean values B has a designated type called $BOOL$, and the
AMN operator $\text{bool}(\ldots)$ is provided to relate predicates to their corresponding Boolean value.
"Bunch Logic: \( \neg P \) implies that \( P \models E = \bot \) and hence \( \neg (P \models E : TRUE) \)"

\[
\begin{align*}
\text{trm}(S) \land \exists s' \cdot \text{prd}(S) &\rightarrow \text{bool}(Q)[s \backslash s'] : TRUE \\
\text{"Bunch Logic: } (\exists z \cdot P \rightarrow E) : F &\rightarrow \forall z \cdot P \Rightarrow E : F"
\end{align*}
\]

"Substitution, note that \( s \) is not free in the definition of \( \text{bool} \)"

\[
\begin{align*}
\text{trm}(S) \land \forall s' \cdot \text{prd}(S) &\Rightarrow \text{bool}(Q)[s \backslash s'] : TRUE \\
\text{"Property of } \text{bool}: Q \leftrightarrow \text{bool}(Q) : TRUE, \text{ note } \text{bool}(Q) \text{ is elementary}
\end{align*}
\]

"wp of characteristic predicates (Theorem 46)"

\[
\begin{align*}
\text{wp}(S, Q) \quad \square.
\end{align*}
\]

If on the other hand we wanted to calculate the prospective value of a computation supposing its wp predicate transformer is known, the trivial solution to this problem is to directly use Def. 34 while expanding \( \text{trm}(S) \) and \( \text{prd}(S) \) into their corresponding formulations in wp semantics (see Fig. 6.1). Doing so we obtain

\[
S \diamond E = \text{wp}(S, \text{true}) \mid \exists s' \cdot \neg \text{wp}(S, s \neq s') \rightarrow E[s \backslash s'] \quad (6.1)
\]

It shows that Eq. 6.1 is not the only way in which we can express \( S \diamond E \) by means of the wp effect of \( S \). An alternative formulation is utilised in the following theorem:

**Theorem 49.** Let \( S \) be generalised substitution and \( E \) an expression with \( x \not\in E \), and moreover \( x \) not contained in the frame of \( S \). Then

\[
S \diamond E = \text{wp}(S, E \neq \bot) \mid \exists x \cdot \neg \text{wp}(S, \neg x : E) \rightarrow x
\]

Using the conjugate weakest-precondition, defined as \( \text{wp}(S, Q) = \neg \text{wp}(S, \neg Q) \), Theorem 49 could equivalently be expressed as

\[
S \diamond E = \neg \text{wp}(S, E = \bot) \mid \exists x \cdot \text{wp}(S, x : E) \rightarrow x \quad (6.2)
\]

Appealing to the conjugate wp here is solely motivated by providing a more intuitive justification of Theorem 49 to the reader (rather then discussion the theorem directly we will explain Eq. 6.2 instead).

Operationally, the conjugate wp tells us what may be achieved by a computation. In other words, if there is some possibility of it establishing the post-condition \( Q \). Contrary to wp, the conjugate wp makes no guarantee that the condition \( Q \) generally holds after execution of the program. Bearing this in mind we can understand the bunch comprehension in Eq. 6.2 as including all values \( x \) (having the same type as the expression \( E \)) which after execution of the computation may be an element of the bunch \( E \). Since \( E \) occurs within the post-condition
of the predicate transformer, the free variables in $E$ really in fact to the state after executing $S$.

To accommodate the possibility of non-termination we have to pre-condition the bunch comprehension with $\neg \wp(S, E = \bot)$. There are actually two sources of non-termination here, one is that $S$ itself might not terminate, and the other that $E$ may evaluate to $\bot$ after executing $S$. The two cases are respectively described by the conditions $\text{trm}(S)$ and $\neg \wp(S, E = \bot)$. However since $\neg \wp(S, E = \bot) = \wp(S, E \neq \bot)$ moreover implies termination of $S$, the former condition $\text{trm}(S)$ is effectively absorbed in their conjunction, which moreover simplifies the pre-condition.

The formal proof of Theorem 49 exploits some fundamental properties of $\wp$ predicate transformers, and is given as follows:

**Proof.** $S \circ E = \wp(S, E \neq \bot) | \exists x \cdot \neg \wp(S, \neg x : E) \rightarrow x$

$S \circ E$

$= \{ \text{Defn of } S \circ E \ (\text{Def. 34}) \}$

$\text{trm}(S) | \exists s' \cdot \text{prd}(S) \rightarrow E[s \setminus s']$

$= \{ \text{Bunch law: } (\exists z \cdot P \rightarrow E) \equiv \neg (\exists z \cdot P \land E = \bot) | \exists x \cdot x : (\exists z \cdot P \rightarrow E) \rightarrow x \}$

$\text{trm}(S) | \neg (\exists s' \cdot \text{prd}(S) \land E[s \setminus s'] = \bot) | \exists x \cdot x : (\exists s' \cdot \text{prd}(S) \rightarrow E[s \setminus s']) \rightarrow x$

$= \{ \text{Chaining law for pre-conditioned bunch (Prop. 24)} \}$

$\text{trm}(S) \land \neg (\exists s' \cdot \text{prd}(S) \land E[s \setminus s'] = \bot)$

$\exists x : \exists z \cdot P \rightarrow E = \exists z \cdot P \land x : E$

$\text{trm}(S) \land \neg (\exists s' \cdot \text{prd}(S) \land E[s \setminus s'] = \bot)$

$\exists x : (\exists s' \cdot \text{prd}(S) \land x : E[s \setminus s']) \rightarrow x$

$= \{ \text{wp characterisation of prd (Fig. 6.1)} \}$

$\text{trm}(S) \land \neg (\exists s' \cdot \neg \wp(S, s \neq s') \land E[s \setminus s'] = \bot)$

$\exists x : (\exists s' \cdot \neg \wp(S, s \neq s') \land x : E[s \setminus s']) \rightarrow x$

$= \{ \text{Logic: } \neg \exists z \cdot \neg Q = \forall z \cdot Q \}$

$\text{trm}(S) \land (\forall s' \cdot \neg (\neg \wp(S, s \neq s') \land E[s \setminus s'] = \bot))$

$\exists x : \neg (\forall s' \cdot \neg (\neg \wp(S, s \neq s') \land x : E[s \setminus s'])) \rightarrow x$

$= \{ \text{Logic (de Morgan)} \}$

$\text{trm}(S) \land (\forall s' \cdot \wp(S, s \neq s') \lor E[s \setminus s'] \neq \bot)$

$\exists x : \neg (\forall s' \cdot \wp(S, s \neq s') \lor \neg x : E[s \setminus s']) \rightarrow x$

$= \{ \text{Disjunctivity of $\wp$ with a frame-independent disjunct}^6 \text{ as } s' \setminus E[s \setminus s'] \}$

---

^6This is Proposition 3 in [Dun02].
The previous theorem particularly shows its usefulness in considerably simplifying the correctness proof of the pv rewrite law for sequential composition. Whereas the original version of this proof included Section 6.4.1 spans nearly four pages, the alternative version of the same proof, included in Appendix C of the thesis, only requires one and a half pages. The reader may be encouraged to compare these two proofs in terms of their complexity and elegance.

At this point it is worth to linger for a moment and recapture what has been achieved so far in this chapter, and also how this fits into the general objectives of the PhD thesis. We have started this chapter by providing a closed-form definition for the prospective value of a GSL computation (Def. 34) — clearly one of the most significant definitions of the thesis. Rather than developing pv semantics from scratch we thus related it to the semantics of generalised substitutions based on the characteristic predicates frame, trm and prd. We then presented a collection of useful algebraic rewrite laws (Prop. 36 to Prop. 42) to eliminate the pv operator from any expressions. Although these laws could have served as a definition for $S \diamond E$ in their own right, together with the fundamental closed-form definition we were able to validate by formal proof the correctness of each law individually, increasing our confidence in the theory developed so far. To complete the semantic link between pv semantics and characteristic predicates we furthermore provided several corollaries (Corollary 43 to Corollary 45) which vice verse allowed us to infer the trm, fis and prd of a computation given as a pv expression transformer. Proving the correctness of these corollaries with regards to the closed-form effectively showed that pv semantics is isomorphic to the one based on characteristic predicates we initially related it to, thereby we don’t lose any expressive power in characterising GSL computations through
The significance of the previous results is that we have succeeded in formalising the initially vague and informal notion of expression transforms given in Chapter 4 characterising the possible outcomes a computation. We have also worked towards providing a useful semantic calculus for the GSL whose laws and axioms are no more complicated than, for example, corresponding ones of the wp calculus, and thereby shown the applicability of Improper Bunch Theory developed in Chapter 5. The motivation for prospective values in the context of objectives for this PhD was on one hand to equip the formal software developer who wants to exploit reversibility in B with some means to “remember” results prior to reversing which otherwise would be destroyed (or so to speak “uncomputed”) through backwards execution, and secondly to support the generation of all results of a backtracking search rather than just one. To subsequently justify the integration of prospective values into the B Method and possibly at some point tool support a formal theory of prospective values had to be developed, which is the contribution of this chapter to the overall thesis objectives.

Remark  Theorem 49 has been inspired by a discussion with Louis Mussat at the ZB 2005 conference, whose valuable input the author would like to acknowledge at this point; the important conjecture provided by Mussat was that $x : S \forall E \Leftrightarrow \neg \wp(S, \neg x : E)$, this indeed proved to be true. Theorem 49 is mostly a specialisation of this which, by appropriately taking into account the non-terminating cases, allows for a complete characterisation of $S \forall E$ by means of Mussat’s initial conjecture.

6.7 Iteration and Recursion

The standard treatment for handling iteration in wp semantics is to first define a suitable ordering on computations, which has to be a complete partial order and with respect of which each construct of the command language must be monotonic. The ordering we typically use for this purpose, fulfilling the monotonicity requirement, is the conventional refinement ordering on generalised substitutions, which is particularly suitable since it reflects in its structure the amount of inherent non-determinism of a computation and thereby gives some special significance to the weakest and strongest solution having some property. The meaning of a loop construct $W$ is then interpreted as the least solution of a certain fixed-point equation. To formulate this equation there are two possible ways, either we characterise $W$ directly as a fixed point in generalised substitutions, or we express for an arbitrary post-condition $Q$ the outcome of $\wp(W, Q)$ as a fixed point in predicates. Both approaches can be found in [BW98].
In pv semantics we may analogously express the meaning of $W \diamond E$ as a fixed point in expressions which is the objective of this section. To do so we first have to determine an ordering on expressions to be employed as the basis for the fixed-point treatment. Here we use the refinement ordering on bunch expressions which was introduced in Section 5.4.2, and which Hehner calls functional refinement ordering, namely $E \sqsubseteq F =_{df} F : E$. This ordering yields a complete lattice on bunches similar to the reverse inclusion ordering on sets over a given type. In defining $\sqsubseteq$ we slightly modify the bunch reverse inclusion lattice on proper bunches by inserting a new bottom element, the improper bunch $\perp$. Note that this doesn't destroy the integrity of the original lattice.

An important property of our refinement ordering on bunch expressions in this context is that all pv transformers arising from generalised substitutions are monotonic with respect to it, this we can prove by structural induction. Consider, for example, the choice construct. Under the assumption $E \sqsubseteq F$, and $S$ and $T$ being monotonic with respect to $\sqsubseteq$, we have to show that $S \triangleright T \diamond E \sqsubseteq S \triangleright T \diamond F$.

$$E \sqsubseteq F \Rightarrow \text{"Assumption: monotonicity of } S \text{ and } T \text{ with respect to } \sqsubseteq"$$

$$S \diamond E \sqsubseteq S \diamond F \land T \diamond E \sqsubseteq T \diamond F$$

$$= \text{"Defn of } \sqsubseteq"$$

$$S \diamond F : S \diamond E \land T \diamond F : T \diamond E$$

$$= \text{"Bunch law: } E : F \land E' : F' \Rightarrow E, E' : F, F'"$$

$$S \diamond F, T \diamond F : S \diamond E, T \diamond E$$

$$= \text{"Prospective value of Choice"}$$

$$S \triangleright T \diamond F : S \triangleright T \diamond E$$

$$= \text{"Defn of } \sqsubseteq"$$

$$S \triangleright T \diamond E \sqsubseteq S \triangleright T \diamond F \quad \square.$$  

In a similar fashion we can easily verify monotonicity for all remaining GSL operators under the respective inductive assumptions, and by structural induction conclude that the pv transformer of any GSL constructs is monotonic with respect to $\sqsubseteq$.

Our satisfying the previous conditions justifies the application of Tarski’s fixed-point theorem [Tar55] and hence enables us to assert that $\mu Y \bullet S \diamond Y$ for any generalised substitution $S$ and expression $Y$ exists, and moreover is uniquely determined. At this point we have to make one allowance deviating from previous considerations, namely that in $\mu Y \bullet S \diamond Y$ the bound variable $Y$ ranges not just over elementary values, but in fact all bunches of its underlying type.

The following theorem characterises the prospective value of a while loop by means of a fixed point in expressions.
Theorem 50. Let \( W \) be a loop construct of the form \( \text{WHILE} \ G \ \text{DO} \ S \ \text{END} \). Then the pv effect of \( W \) for any expression \( E \) is given by

\[
W \circ E = \mu Y \bullet \text{if } G \text{ then } S \circ Y \text{ else } E \text{ end}
\]

Proof. Before deriving the pv effect of \( W \) and thereby proving Theorem 50 we will first establish a lemma which enables us to infer the pv effect of the transitive opening \( S^\wedge \) of a generalised substitution \( S \). The transitive opening is the basic iteration construct introduced by Abrial in [Abr96b]. Formally it is the least solution of the fixed-point equation \( X = (S; X) \parallel \text{skip} \) in generalised substitutions. Operationally, we can think of it as the choice of executing \( S \) an arbitrary number of times, including if feasible the case of infinite repetition\(^7\) leading to divergent behaviour of \( S^\wedge \).

Lemma 51. Let \( S \) be a generalised substitution, and \( E \) an expression. The pv effect of the transitive opening \( S^\wedge \) of \( S \) is given through

\[
S^\wedge \circ E = \mu Y \circ S \circ Y, E
\]

Proof. By definition \( S^\wedge \) is the least solution of \( X = (S; X) \parallel \text{skip} \), thus

\[
S^\wedge = (S; S^\wedge) \parallel \text{skip}
\]

\[\Rightarrow \text{“Leibniz’s law: } S = T \Rightarrow S \circ E = T \circ E\]

\[S^\wedge \circ E = (S; S^\wedge) \parallel \text{skip} \circ E\]

\[= \text{“Prospective value of Choice (see Fig. 6.3)”}\]

\[S^\wedge \circ E = (S; S^\wedge) \circ E, \text{skip} \circ E\]

\[= \text{“Prospective value of Sequential Composition and Skip (see Fig. 6.3)”}\]

\[S^\wedge \circ E = S \circ S^\wedge \circ E, E\]

\[= \text{“Substitution: } Y \equiv S^\wedge \circ E\]

\[Y = S \circ Y, E\]

We have obtained now a fixed-point equation in expressions for \( S^\wedge \circ E \). The remaining open question is whether \( S^\wedge \circ E \) is indeed the weakest \( Y \) that satisfies the equation above. Intuitively we would think so since weaker expressions comprise more elementary values, and thus if occurring as the prospective value of a computation convey more behaviour rendering it less deterministic (note that \( S^\wedge \) is the least-deterministic substitution satisfying \( S^\wedge = (S; S^\wedge) \parallel \text{skip} \)).

When first presenting this theorem in [ZSD05] we satisfied ourselves with this informal justification. Here, however, we would like to go one step further and

\[\text{\(^7\)The related concept } S^* \text{ known as the transitive closure of } S \text{ being the strongest fixed point of } X = (S; X) \parallel \text{skip} \text{ operationally differs from } S^\wedge \text{ by not taking into account infinite repetition, namely under such conditions the computation is rendered infeasible rather than abortive. Whereas the transitive closure is sufficient in partial correctness, the transitive opening is more appropriate in a total-correctness setting.} \]
present a formal proof that in fact $S^\wedge \circ E$ is the weakest expression $Y$ satisfying $Y = S \circ Y, E$. We do so by using a corollary due to Back and von Wright, in particular Corollary 19.6 on page 323 in [BW98]. This corollary is a consequence of what they call the fusion theorem for lattices (Theorem 19.5 on page 322).

**Theorem 19.5 [BW98]** Assume that $h : \Sigma \rightarrow \Gamma$ is a continuous function and that $f : \Sigma \rightarrow \Sigma$ and $g : \Gamma \rightarrow \Gamma$ are monotonic functions on complete lattices. Then

$$h \circ f = g \circ h \Rightarrow h(\mu f) = \mu g$$

Examining the right hand of the implication we realise why the theorem is called 'fusion theorem', i.e. since we are able to prove that $h$ maps the weakest fixed point of $f$ in one lattice onto the weakest fixed point of $g$ in another. For our purposes we won’t apply the fusion theorem directly, but its Corollary 19.6. The proofs for both, the fusion theorem and its corollary are given in [BW98].

**Corollary 19.6 [BW98]** Assume that $f : (\Gamma \rightarrow \Gamma) \rightarrow (\Gamma \rightarrow \Gamma)$ and $g : \Gamma \rightarrow \Gamma$ are monotonic functions on complete lattices and that $x \in \Gamma$ is fixed. Then

$$(\forall h \cdot f(h)(x) = g(h(x))) \Rightarrow (\mu f)(x) = \mu g$$

In the context of our proof $\Gamma$ represents the domain of expressions. The function $f$ accordingly maps expression transformers to expression transformers, and is given here through $f(X) = (S ; X) \parallel \text{skip}$. The function $g$ on the other hand is defined as $g(Y) = S \circ Y, E$.

Note that pv expression transformers can be isomorphically identified with generalised substitutions. Strictly, we additionally have to provide a definition for refinement of pv expression transformers in order to apply Corollary 19.6 [BW98] in the desired form. This is achieved by virtue of the following definition:

**Definition 52.**

$$S \sqsubseteq T =_{df} \forall E \cdot S \circ E \sqsubseteq T \circ E$$

This definition of refinement is identical to conventional refinement of GSL computations formulated in terms of wp (see Def. 53). Thus $S \sqsubseteq T$ according to Def. 52 is true exactly if $S$ is refined by $T$ in wp semantics i.e. $\forall Q \cdot \wp(S, Q) \Rightarrow \wp(T, Q)$. A formal proof of this may be easily constructed appealing to Def. 34.

Defining refinement of computations in pv semantics in such a way is important as it enables us to conclude that $S^\wedge$ is not just the weakest generalised substitution (under standard wp refinement) satisfying $X = (S ; X) \parallel \text{skip}$, but additionally $S^\wedge \circ -$ is the weakest pv expression transformer that does so.

The consequence of Corollary 19.6 [BW98] instantiated as previously described gives us immediately what we want, hence all we have to show is its
CHAPTER 6. PROSPECTIVE-VALUE SEMANTICS

102

antecedent:

\[ \mu X \bullet (S \cdot X) \| \text{skip} \circ E = \mu Y \bullet S \circ Y, E \]

\[ \Leftarrow \text{"Corollary 19.6 [BW98]"} \]

\[ \forall T \bullet (\lambda X \bullet (S \cdot X) \| \text{skip})(T) \circ E = (\lambda Y \bullet S \circ Y, E)(T \circ E) \]

\[ \Rightarrow \text{"Function application"} \]

\[ \forall T \bullet (S \cdot T) \| \text{skip} \circ E = S \circ T \circ E, E \]

\[ \Rightarrow \text{"Prospective value of Choice"} \]

\[ \forall T \bullet (S \cdot T) \circ E, \text{skip} \circ E = S \circ T \circ E, E \]

\[ \Rightarrow \text{"Prospective value of Sequential Composition and Skip"} \]

\[ \forall T \bullet S \circ T \circ E, E = S \circ T \circ E, E \]

\[ \Rightarrow \text{"Equality"} \]

true

This concludes the proof of Lemma 51 showing that \( S^\wedge \circ E \) is indeed the weakest expression \( Y \) that solves \( Y = S \circ Y, E \). In other words

\[ S^\wedge \circ E = \mu Y \bullet S \circ Y, E \quad \square. \]

Equipped with the previous Lemma 51 we can conclude the proof of Theorem 50 now by using Abrial’s characterisation of the while loop involving the transitive opening:

\[ \text{WHILE } G \text{ DO } S \text{ END } =_{df} (G \Rightarrow S)^\wedge ; \neg G \Rightarrow \text{skip} \quad (6.3) \]

\[ W \circ E \]

\[ \Rightarrow \text{"Definition of } W" \]

\[ \text{WHILE } G \text{ DO } S \text{ END } \circ E \]

\[ \Rightarrow \text{"Equation 6.3 characterising the while loop"} \]

\[ (G \Rightarrow S)^\wedge ; \neg G \Rightarrow \text{skip} \circ E \]

\[ \Rightarrow \text{"Prospective value of Sequential Composition (see Fig. 6.3)"} \]

\[ (G \Rightarrow S)^\wedge \circ \neg G \Rightarrow \text{skip} \circ E \]

\[ \Rightarrow \text{"Prospective value of Guard (see Fig. 6.3)"} \]

\[ (G \Rightarrow S)^\wedge \circ \neg G \rightarrow \text{skip} \circ E \]

\[ \Rightarrow \text{"Prospective value of Skip (see Fig. 6.3)"} \]

\[ (G \Rightarrow S)^\wedge \circ \neg G \rightarrow E \]

\[ \Rightarrow \text{"Lemma 51"} \]

\[ \mu Y \bullet G \Rightarrow S \circ Y, \neg G \rightarrow E \]
"Prospective value of Guard (see Fig. 6.3)"
\[ \mu Y \cdot G \rightarrow S \diamond Y, \neg G \rightarrow E \]
= "Rewriting guarded bunches as conditional expression"
\[ \mu Y \cdot \text{if } G \text{ then } S \diamond Y \text{ else } E \text{ end } \]

6.8 Conclusions

In this chapter we formally developed a prospective-value semantics for the Generalised Substitution Language by formally introducing the new operator \( S \diamond E \) being an expression transformer which yields the value of \( E \) after execution of \( S \), but without incurring any of the side-effects \( S \) might have. We have defined the prospective value of a computation by appealing to its semantics in terms of \( \text{trm} \) and \( \text{prd} \), and thereby provided a fundamental closed-form definition for \( S \diamond E \).

To simplify reasoning with prospective values, we gave a collection of algebraic rewrite laws covering each operator of the GSL, and individually proved the correctness of each of the laws. It showed that by utilising Improper Bunch Theory as the underlying expression formalism we can formulate the rewrite laws for \( \text{pv} \) semantics in the most concise and operationally suggestive way.

We further established a vice-versa connect between the characteristic predicates \( \text{trm} \) and \( \text{prd} \) and the \( \text{pv} \) effect of a computation, and thereby effectively proved that \( \text{pv} \) semantics is isomorphic to the relational semantic model of the GSL in terms of these characteristic predicates; hence we don’t lose any expressiveness when reasoning about computations in terms of their prospective values. Another contribution has been to establish a direct link between \( \text{pv} \) and \( \text{wp} \) semantics which lead to an interesting theorem alternatively characterising \( S \diamond E \) in terms of the \( \text{wp} \) effect of \( S \), resulting to a considerably simplification of the rather complex and long-wound correctness proof for the sequential composition rewrite law.

We finally gave a treatment of iteration in \( \text{pv} \) semantics by describing the prospective value of the while-loop construct directly by means of a fixed point in \emph{expressions}, complementing the familiar representations by either a fixed point in predicates or generalised substitutions.
Chapter 7

Refinement of Reversible Computations

7.1 Introduction

In this chapter we are interested in the practical aspects of tool-supported formal software development of applications which exploit the backtracking semantics of guards and choice at the level of implementations. This develops further the ideas presented in Chapter 3 of the thesis by not just considering generalised substitutions, but operations within the context of a B specification and implementation. Here we moreover address the important issue of refinement in the context of reversible computations. The concept of refinement is a crucial one in formal software construction with the B Method, and besides crops up in many other formalisms for rigorous program or systems development. Note that in this chapter we don’t utilise any of the results obtained in Chapter 4 to 6. The contribution of this chapter satisfies objectives 6 and 7 of the PhD investigation stated in Section 1.2. It also proves our central hypothesis that the B Method can be successfully adapted and used as a vehicle for the formal development of applications exhibiting reversibility in the previously described form.

Using the B Method, we typically specify the behaviour of a system first in abstract terms by means of a collection of B machines. Here the aim is primarily to express mathematically in a concise way what each component is set to achieve without having to provide details of how such behaviour operationally may be realised, i.e. by means of concrete data-types and algorithms. The abstract specification may also contain non-determinism defining a ‘don’t care’ situation in which certain details of the component’s behaviour are left indeterminate giving the implementor the freedom of resolving them as he chooses. The refinement of a B machine is essentially a re-expression of the abstract model which however is closer to a possible implementation, revealing aspects of the algorithm and concrete data representation adopted. The implementation of a B machine is a ‘special’ and ultimate refinement which is concrete enough to be translated
directly into some target language.

The methodology underlying B is to turn an abstract specification into a concrete implementation through a series of correctness-preserving refinement steps. Each step should add more detail to the model until the last refinement fulfills the requirement of an implementation (i.e. its operations consist exclusively of constructs permissible in B0 — the implementation-level sub-language of B-AMN). Note that in practice it is often the case that abstract machines are transformed into implementations in only one refinement step without the need of any intermediate models. The justification for such models is primarily to factorise, and thereby simplify, the refinement proof.

When developing software in B taking advantage of reversible computations through guards and choice at the level of execution, we don’t want to abandon the B development approach just outlined. Its advantage lies for instance in the fact that when proving abstract properties about a system consisting of numerous components, it is sufficient to refer to the machine specification of each component and not its refinement or implementation. The latter are usually more complex in mathematical terms. The proof on the other hand that some refinement respects the behaviour of its abstract model (or previous refinements in a chain of intermediate models) is carried out as a separate independent obligation.

What allows us to conduct the correctness proof of a system in such a manner as described is the transitivity of the refinement relation and the monotonicity of all specification language constructs with respect to it. Strictly there are two guises of refinement which have to be differentiated: one is operational refinement and defined on generalised substitutions in general outside the context of a B operation, and the other one is data refinement which only has meaning in the context of some data type (or model) which we encapsulate in a B machine.

In this chapter we limit our attention in essence to operational refinement when examining the suitability of the classical refinement approach and thus aiming to adapt it for our purposes. This is so since for the case study conducted in this chapter we don’t require data refinement in particular because none of the B components we will develop has any state, and under such conditions the data refinement proof obligations of B exactly guarantee operational refinement of the underlying B operations (or more precisely their bodies). Future work however could entail the application of results in this chapter to data refinement in general, and with particular focus on how the refinement proof obligations of B may have to be altered. We think that such an investigation would naturally follow from this chapter’s contribution and is certainly in scope regarding the wider context of our research in formally developing reversible computations in B, albeit not central to the ideas discussed in this chapter.
7.2 The Reversible Virtual Machine

The Reversible Virtual Machine (RVM) is a stack-based virtual machine and associated compiler and interpreter for a simple postfix language; its core instruction set is taken from the ISO Forth Standard. The stack architecture inherited from Forth is augmented with an additional history stack, which is used to store information normally discarded during a computation and to perform the book keeping required for backtracking in order to promote *reverse* program execution. In addition to the normal control structures of a sequential programming language, the RVM supports guard and choice commands which are implemented according to their operational interpretation in terms of backtracking. Thus if during program execution the RVM encounters a guard statement whose condition evaluates to false, it engages in reverse execution until it meets a choice statement which has some unexplored branch from whereon execution again proceeds in a forward direction.

In its characteristic feature to carry out backtracking by means of a history stack, the RVM also performs garbage collection upon reverse execution. Its ability to handle garbage in this simple way helps us to provide an efficient and complete implementation of finite sets as immutable references, which has been mostly developed by the author of the thesis (the RVM itself was developed by Bill Stoddart within the Formal Methods and Programming Research Group at the University of Teesside). Support for bunches is not provided at present, the RVM does, however, support prospective-value constructs which generally represent bunches, though they must occur within the context of a set construction if they yield a non-elementary bunch. The two main objectives when developing the RVM were first to provide a platform and experimental target language for reversible programming, and secondly to make more abstract (mathematical) data structures such as sets and functions directly available to the implementor.

The purpose of the RVM in the context of research for this PhD is that it acts as an execution environment and target language for translating B implementations that utilise guards and choice constructs in their operations e.g. in form of AMN SELECT statements, as well as data types such as sequences which are conventionally not considered implementable by the B tools framework. The language we thus obtain we will call RB0 (*Reversible B0*), and although commercial B development tools such as the Atelier B can be configured to accept, analyse and process RB0 implementations, there is no translation support into executable code. Our aim in this section is not to explain the implementation details and the many features of the RVM in its current state of development, but to make the reader aware of its existence and rôle as outlined above with regards to the contributions of this chapter. Further aspects concerning the exact definition of RB0, and how the translation of RB0 into RVM Forth, the native language of the RVM, is to be conducted are also not addressed in this PhD investigation\(^1\).

\(^1\)Another PhD thesis by Robert Lynas (Formal Methods and Programming Research Group,
Note Further information regarding the RVM can be found in the following publication and technical report [Sto03, Sto06]; the software itself may be downloaded from the website of the Formal Methods and Programming Research Group at the University of Teesside (under Research Activities/Virtual Machines and Forth):

http://www.scm.tees.ac.uk/formalmethods/

7.3 Refinement vs. *-refinement

Conventionally, one computation $S$ is refined by another computation $T$ if any behaviour $T$ might exhibit is also a possible behaviour of $S$. The refinement relation $\sqsubseteq$ between computations may be formalised by the following higher-order definition:

Definition 53.

$$S \sqsubseteq T =_{df} \forall Q \cdot \wp(S, Q) \Rightarrow \wp(T, Q)$$

with $S$ and $T$ being generalised substitutions, and $Q$ ranging over predicates.

It is easy to show that $\sqsubseteq$ is reflexive, anti-symmetric and transitive. Furthermore, GSL computations form a complete lattice under $\sqsubseteq$, with abort being its bottom, and magic being its top. In particular, we can use Def. 53 to show that $x := 1 \parallel x := 2 \sqsubseteq x := 1$, thus if $x := 1 \parallel x := 2$ occurs as the specification of some operation within a B development, $x := 1$ would be one of its legitimate refinements. This example illustrates how non-determinism can be reduced through the refinement process.

In what follows we will investigate whether this conventional notion of refinement is still appropriate in the context of reversible computations, e.g. those admitting infeasible operation as executable constructs.

A well-known fact is that magic is a refinement of any GSL computation as shown by the following proof:

$$S \sqsubseteq \text{magic}$$

$$= \text{"Defn of } \sqsubseteq \text{ (Def. 53)"}$$

$$\forall Q \cdot \wp(S, Q) \Rightarrow \wp(\text{magic, } Q)$$

$$= \text{"Defn of magic"}$$

$$\forall Q \cdot \wp(S, Q) \Rightarrow \wp(\text{false } \implies \text{skip, } Q)$$

$$= \text{"Defn of guarded substitution"}$$

University of Teesside) is currently in preparation which will aim to do justice to this issue, in the current work however we bear the reader’s excuse that notions considering the particulars of RB0 and its translation have to remain vague.

2Other first-order formulations are possible too.
\[ \forall Q \bullet \text{wp}(S, Q) \Rightarrow \text{false} \Rightarrow \text{wp}(\text{skip}, Q) \]

\[ \Rightarrow \text{“Logic”} \]

\[ \forall Q \bullet \text{wp}(S, Q) \Rightarrow \text{true} \]

\[ \Rightarrow \text{“Logic”} \]

\[ \text{true} \]

In classical B this is not a problem as it is impossible to express \textit{magic} within the implementation level sub-language of the AMN known as B0. This is clearly so since all permissible constructs in B0 have to be \textit{conventionally} implementable in order to be translated into some conventional target language, thus feasible.

In our work, however, we extend B0 to give the reversible language RB0 which unlike B0 includes stand-alone guards and choice statements. Consequently this would enable the implementor to employ \textit{magic} \equiv \text{false} \Rightarrow \text{skip} as a valid but vacuous refinement for any abstract B operation. Although our execution environment, the Reversible Virtual Machine, supports invoking \textit{magic} by reporting “\textit{ko}” rather than “\textit{ok}” to the user if no feasible path of execution can be found through the program, this is not an acceptable response if the specification of an operation indeed possesses some behaviour. Reporting infeasibility to the user upon running a software application, we assert, is \textit{almost} as bad as non-termination; “almost” since detecting the attempted execution of a miracle doesn’t conceptually break the system as non-termination does, so at least the program may be cleanly shut down providing some feedback to the user.

The previous line of thought suggests that we have to protect ourselves somewhat from what may be called “over-refinement” — the reduction of non-determinism to a point where no behaviour is left. Clearly we don’t want to completely abandon the idea of refinement, but we have to tame its effect in situations as the one previously described. In an attempt to do so we introduce a \textit{stronger} refinement relation \( \sqsubseteq^* \) which we call \( ^* \)-refinement (pronounced “star-refinement”):

\textbf{Definition 54.}

\[ S \sqsubseteq^* T \overset{=}{=}_{df} S \sqsubseteq T \land (\text{fis}(S) \Rightarrow \text{fis}(T)) \]

where \( S \) and \( T \) are generalised substitutions, and \( \text{fis}(S) \) is the feasibility of \( S \) defined in the usual way as \( \neg \text{wp}(S, \text{false}) \).

Informally \( S \sqsubseteq^* T \) states that \( T \) is a conventional refinement of \( S \), and that \( T \) cannot be less feasible than \( S \). This enables the developer to reduce non-determinism through refinement, but not to a point where behaviour is entirely removed. We illustrate this by means of “blob diagrams”\(^3\). In Fig. 7.1 and Fig. 7.2

\(^3\)The idea of blob diagrams to visual GSL computation is due to Steve Dunne and Bill Stoddart who used them as a successful teaching tool in their Formal Aspects of Computer Science module at the University of Teesside. In fact it is also possible to characterise termination within these diagrams by adding a “fringe” over the top border of the box in states where the initial computation doesn’t terminate.
CHAPTER 7. REFINEMENT OF REVERSIBLE COMPUTATIONS

Figure 7.1: Blob diagram illustrating proper feasibility-preserving refinement.

each box represents a generalised substitution acting on only one state variable $x$. The x-axis of each box is associated with the before state $x$, and the y-axis with the after state $x'$. The blobs themselves characterise the non-deterministic update to the state variable, hence visualise the prd of the underlying computations. Note that outside the blob the computation is infeasible.

The first diagram Fig. 7.1 shows a proper *-refinement. Here the blob in the left-hand box is refined by a line in the right-hand box which passes through the area of the of the initial blob. Whereas the left-hand computation is non-deterministic, the right-hand one is deterministic. The fact that the points in the right box are a subset of the points in the left box guarantees a conventional refinement, the fact that the projection of the points in the right box on the x-axis are a subset of the projection of the points in the left box on the x-axis guarantees a *-refinement.

In Fig. 7.2 we illustrate the case of a proper refinement which is not a *-refinement. Here the points of the blob in the left box are as before a subset of the points of the blob in the right box, however if we look at their projection on the x-axis, only the second half of the blob and therefore projected points are preserved. We would achieve this scenario by guarding the initial computation with $x > x_0$ where $x_0$ lies somewhere in the middle of the blob (i.e. the dividing line between the two halves). In initial states corresponding to the first half we have thus introduced infeasibility that wasn’t present in the initially refined computation — this violates the requirement for *-refinement.

As a more concrete example relating back to the previously described problem of considering magic as an implementable but vacuous refinement of any specification, for *-refinement we can prove, as for conventional refinement, that $x := 1 \parallel x := 2 \sqsupset^* x := 1$ since both computations are everywhere feasible. On the other hand it is not true that $x := 1 \parallel x := 2 \sqsupset^* \text{magic}$ because

$$\text{fis}(x := 1 \parallel x := 2) = \text{true} \not\Rightarrow \text{false} = \text{fis(magic)}.$$
CHAPTER 7. REFINEMENT OF REVERSIBLE COMPUTATIONS

Having thus provided a formal definition for *-refinement, the logical next step is now to examine if it can be used as a basis for piecewise and stepwise development as they are fundamental to the B approach. In other words we have to check whether all constructs of the underlying command language, i.e. the GSL, are monotonic with respect to $\sqsubseteq^*$.

7.3.1 Monotonicity Properties

In this section we examine the monotonicity properties the Generalised Substitution Language constructors with respect to *-refinement. As we will see, unfortunately not all operators of the GSL are monotonic w.r.t. $\sqsubseteq^*$, since monotonicity is lost in the first operand of sequential composition. We first will give proofs for those operators which are monotonic, and then present a counter-example in the case of sequential composition’s first operand.

Theorem 55. $\sqsubseteq^*$ preserves monotonicity of pre-conditioned substitution, guarded substitution, choice, unbounded choice, and sequential composition in its second operand.

Proof. By saying that some GSL connective $op$ is monotonic with respect to $\sqsubseteq^*$ we mean $op$ is monotonic in each of its operands in turn\textsuperscript{4}. To show monotonicity of some n-ary $op$ in its k-th operand for $1 \leq k \leq n$ we have to prove that under the assumption $S_k \sqsubseteq^* T_k$, $\quad op(S_1 \ldots S_{k-1}, S_k, S_{k+1} \ldots S_n) \sqsubseteq^* op(S_1 \ldots S_{k-1}, T_k, S_{k+1} \ldots S_n)$.

\textsuperscript{4}An alternative interpretation for $op$ to be monotonic is to require monotonicity in all its operands at once, formally under the assumption $S_1 \sqsubseteq^* T_1 \land S_2 \sqsubseteq^* T_2 \land \ldots \land S_n \sqsubseteq^* T_n$ we have $op(S_1, \ldots, S_n) \sqsubseteq^* op(T_1, \ldots, T_n)$. Note that these two notions of monotonicity are equivalent providing the underlying ordering is reflexive and transitive, as $\sqsubseteq^*$ indeed is.
Since all GSL connectives are monotonic with respect to ordinary refinement ⊑, it is sufficient to prove the second conjunct of Def. 54:

\[ \text{fis}(\text{op}(S_1 \ldots S_{k-1}, S_k, S_{k+1} \ldots S_n)) \Rightarrow \text{fis}(\text{op}(S_1 \ldots S_{k-1}, T_k, S_{k+1} \ldots S_n)). \]

**Monotonicity of Pre-conditioned Substitution.** Under the assumption \( S \sqsubseteq^* T \) we have to show that fis\((P \mid S) \Rightarrow fis(P \mid T)\) for an arbitrary predicate \( P \).

\[
\text{fis}(P \mid S) \Rightarrow fis(P \mid T)
\]

\[ = \text{“Defn of fis”} \]

\[ \neg wp(P \mid S, \text{false}) \Rightarrow \neg wp(P \mid T, \text{false}) \]

\[ = \text{“wp of pre-conditioned substitution”} \]

\[ \neg (P \land wp(S, \text{false})) \Rightarrow \neg (P \land wp(T, \text{false})) \]

\[ = \text{“Logic (de Morgan)”} \]

\[ \neg P \lor \neg wp(S, \text{false}) \Rightarrow \neg P \lor \neg wp(T, \text{false}) \]

\[ = \text{“Defn of fis”} \]

\[ \neg P \lor fis(S) \Rightarrow \neg P \lor fis(T) \]

Proof by cases: \( P \) and \( \neg P \).

Assume \( P \) (case 1),

\[ \neg P \lor fis(S) \Rightarrow \neg P \lor fis(T) \]

\[ \iff \text{“Assumption: } P \text{”} \]

\[ \neg \text{true} \lor fis(S) \Rightarrow \neg \text{true} \lor fis(T) \]

\[ = \text{“Logic”} \]

\[ fis(S) \Rightarrow fis(T) \]

\[ \iff \text{“Assumption } S \sqsubseteq^* T \text{”} \]

Assume \( \neg P \) (case 2),

\[ \neg P \lor fis(S) \Rightarrow \neg P \lor fis(T) \]

\[ \iff \text{“Assumption } \neg P \text{”} \]

\[ \text{true} \lor fis(S) \Rightarrow \text{true} \lor fis(T) \]

\[ = \text{“Logic”} \]

\[ \text{true} \quad \Box. \]

**Monotonicity of Guarded Substitution.** Under the assumption \( S \sqsubseteq^* T \) we have to show that fis\((P \equiv S) \Rightarrow fis(P \equiv T)\) for an arbitrary predicate \( P \).

\[ \text{fis}(P \equiv S) \Rightarrow fis(P \equiv T) \]
CHAPTER 7. REFINEMENT OF REVERSIBLE COMPUTATIONS

= "Defn of fis"
  ¬wp(P \implies S, false) \implies ¬wp(P \implies T, false)

= "wp of guarded substitution"
  ¬(P \implies wp(S, false)) \implies ¬(P \implies wp(T, false))

= "Logic"
  ¬(¬P \lor wp(S, false)) \implies ¬(¬P \lor wp(T, false))

= "Logic (de Morgan)"
  P \land ¬wp(S, false) \implies P \land wp(T, false)

= "Defn of fis"
  P \land fis(S) \implies P \land fis(T)

Proof by cases: P and ¬P.

Assume P (case 1),
  P \land fis(S) \implies P \land fis(T)

\iff "Assumption P"
  true \land fis(S) \implies true \land fis(T)

= "Logic"
  fis(S) \implies fis(T)

\iff "Assumption S \subseteq^* T"

Assume ¬P (case 2),
  P \land fis(S) \implies P \land fis(T)

\iff "Assumption ¬P"
  false \land fis(S) \implies false \land fis(T)

= "Logic"
  true \square.

Monotonicity of Choice. Since choice is commutative it is sufficient to show that under the assumption S \subseteq^* T we also have fis(S \parallel R) \implies fis(T \parallel R).
  fis(S \parallel R) \implies fis(T \parallel R)

= "Defn of fis"
  ¬wp(S \parallel R, false) \implies ¬wp(T \parallel R, false)

= "wp of choice"
  ¬(wp(S, false) \land wp(R, false)) \implies ¬(wp(T, false) \land wp(R, false))

= "Logic (de Morgan)"
\[ \neg \wp(S, \text{false}) \lor \neg \wp(R, \text{false}) \Rightarrow \neg \wp(T, \text{false}) \lor \neg \wp(R, \text{false}) \]

\[ = \text{“Defn of fis”} \]

\[ \text{fis}(S) \lor \text{fis}(R) \Rightarrow \text{fis}(T) \lor \text{fis}(R) \]

\[ = \text{“Logic”} \]

\[ \text{fis}(S) \Rightarrow \text{fis}(T) \lor \text{fis}(R) \]

\[ = \text{“Logic”} \]

\[ \text{fis}(S) \Rightarrow \text{fis}(T) \]

\[ \Leftarrow \text{“Assumption } S \sqsubseteq^* T \text{”} \Box. \]

**Monotonicity of Sequential Composition in its 2nd Operand.** Under the assumption \( S \sqsubseteq^* T \) we have to show that \( \text{fis}(R ; S) \Rightarrow \text{fis}(R ; T) \).

\[ \text{fis}(R ; S) \Rightarrow \text{fis}(R ; T) \]

\[ = \text{“Defn of fis”} \]

\[ \neg \wp(R ; S, \text{false}) \Rightarrow \neg \wp(R ; T, \text{false}) \]

\[ = \text{“wp of sequential composition”} \]

\[ \neg \wp(R, \wp(S, \text{false})) \Rightarrow \neg \wp(R, \wp(T, \text{false})) \]

\[ = \text{“Logic”} \]

\[ \wp(R, \wp(T, \text{false})) \Rightarrow \wp(R, \wp(S, \text{false})) \]

\[ \Leftarrow \text{“Monotonicity of wp w.r.t. implication”} \]

\[ \wp(T, \text{false}) \Rightarrow \wp(S, \text{false}) \]

\[ = \text{“Logic”} \]

\[ \neg \wp(S, \text{false}) \Rightarrow \neg \wp(T, \text{false}) \]

\[ = \text{“Defn of fis”} \]

\[ \text{fis}(S) \Rightarrow \text{fis}(T) \]

\[ \Leftarrow \text{“Assumption } S \sqsubseteq^* T \text{”} \Box. \]

**Monotonicity of Unbounded Choice.** Under the assumption \( S \sqsubseteq^* T \) we have to show that \( \text{fis}(\@ z \cdot S) \Rightarrow \text{fis}(\@ z \cdot T) \).

\[ \text{fis}(\@ z \cdot S) \Rightarrow \text{fis}(\@ z \cdot T) \]

\[ = \text{“Defn of fis”} \]

\[ \neg \wp(\@ z \cdot S, \text{false}) \Rightarrow \neg \wp(\@ z \cdot T, \text{false}) \]

\[ = \text{“wp of unbounded choice”} \]

\[ \neg(\forall z \cdot \wp(S, \text{false})) \Rightarrow \neg(\forall z \cdot \wp(T, \text{false})) \]

\[ = \text{“Logic”} \]
As mentioned earlier sequential composition is not monotonic in its first operand with respect to $\sqsubseteq^*$. We illustrate this by a counter-example. Consider the following computation $S$:

\[
S \equiv x := 1 \; \parallel x := 2 ; \; x = 1 \implies \text{skip}
\]

Upon closer inspection we ought to convince ourselves, either operationally or by proof, that $S$ is equivalent to the assignment $x := 1$. Therefore the program specification $S$ is everywhere feasible. If we subject $S$ now to piecewise development under the $\sqsubseteq^*$ refinement regime, we may replace the left hand of the sequential composition by $x := 2$; as pointed out earlier such would constitute a correct $^*$-refinement of $x := 1 \; \parallel x := 2$.

Doing so on the other hand renders the overall computation infeasible since $x := 2 ; \; x = 1 \implies \text{skip}$ reduces to magic. The resulting computation is hence less feasible than the original one $S$, thus cannot be a valid $^*$-refinement of it. In other words, maintaining feasibility “in the small” does not serve us here to preserve feasibility “in the large”: despite maintaining the feasibility of $x := 1 \; \parallel x := 2$ we lost overall feasibility of the refinement of $S$ during piecewise development.

From the previous elaboration we immediately conclude that sequential composition in general cannot be monotonic with respect to $\sqsubseteq^*$ in its first operand. This means that at least naively we cannot use $\sqsubseteq^*$ as a basis for piecewise and stepwise formal software development as in supported in B. The implications of this are profound because at first glance it seems that either we have to prohibit the use of sequential composition in refinements and implementations, or otherwise give up on the idea of piecewise development in the context of reversible computations. A less drastic solution however is to demand that generalised substitutions appearing as the first operand of a sequential composition should never be subject to refinement. We will further discuss this in Section 7.7 where we aim to partially restore a piecewise approach. For the following case study it should be noted computations occurring within the first operand of sequential composition are effectively not refined.
7.4 The Knight’s Tour Case Study

In this part of the thesis our focus is to present a case study illustrating through a substantial B development how a standard backtracking algorithm can be developed which utilises guards and choice statements within its implementation to carry out the respective search. In Chapter 3 we first examined the interaction of guards and choice by pointing out their operational interpretation in terms of backtracking. The examples given in the chapter were deliberately kept simple to emphasis the essential points made. In this chapter however we like to take things further by exploiting this property in the context of the formal B development of a practically relevant application performing some non-trivial task, and thereby examining and discussing issues surrounding refinement, piecewise and stepwise development, proof obligations and translation. This ties in with objectives 6 and 7 of the thesis (see Section 1.2) showing how our paradigm of specifying reversible computations through a sequential programming language can be exploited in practical software development, and how the B tool support, namely the Atelier B, can be used to assist in such cases.

Note that for the purpose of the case study developed here we don’t draw upon the material presented in Chapter 4 to 6, namely Improper Bunch Theory and prospective-value semantics. A case study which involves those parts of the thesis is still pending and subject to future work.

The Knight’s Tour is a classical chess board problem illustrating algorithmic search by means of backtracking. We will develop it here in a reversible manner taking advantage of the B Method and its tool support, in particular the Atelier B and the Click’n Prove interactive prover interface driving the openly available B4free distribution of the Atelier B tool set[AC03]. Although the Knight’s Tour problem in itself may not be of singular practical interest, the development we will carry out here can nevertheless be generalised to apply to a whole class of practically relevant problems such as Artificial Intelligence planning algorithms, finding solutions to time-tabling problems, etc.

Placing the chess piece of a knight on some arbitrary square of an empty chess board, the problem of the Knight’s Tour is to find a sequence of moves for the knight to visit each square of the chess board exactly once. We assume at this point the reader is sufficiently acquainted with the rules of chess to know what constitutes a valid knight move, otherwise ample material can be found, for example, on the World Wide Web.

Traditionally we implement a solver for this kind of problem using a backtracking approach. Roughly, such would perform the following steps:

1. Place the knight on some designated initial square on the chess board.

2. Enumerate the set of valid moves for the knight from its current position. Only include those squares which are not already part of the current tour.

3. Choose a square from the set that hasn’t already been chosen upon
previous path explorations, and move the knight to it. If the set is empty, or no choices are permissible, backtrack to the most recent choice.

4. If all squares have been visited, terminate the algorithm while reporting the current tour as a solution, otherwise continue with step 2.

The reversible algorithm we will develop here very closely resembles these steps. The major difference is that instead of performing backtracking when reaching a dead end, we provoke reverse execution through guards and choice. The choices made by the knight at any point advancing it to the next square will formally be modelled as non-deterministic choices, although conceptually they may be more adequately termed “provisional choices”\(^5\).

### 7.4.1 The Abstract Development

A new Atelier B project KnightsTour is introduced to subsequently host the B components for the Knight’s Tour case study. For the abstract specification of the Knight’s Tour solver we create a B machine KnightsTourSolver encapsulating the solve operation which non-deterministically delivers a solution for the Knight’s Tour problem. To specify this operation in the most concise manner, we first introduce several abstract and concrete constants as well as two DEFINITIONS as shown in Fig. 7.3. Their informal interpretation is given as follows.

#### Concrete Constants

- **SQUARE** represents the set of squares on the chess board. They are numbered continuously from 0 to 63 traversing the board first from left to right, and secondly from bottom to top. Accordingly, square 0 is located near the bottom-left corner, and square 63 near the top-right corner.

#### Abstract Constants

- **valid_move** is a relation containing all combinations of pairs of squares on the chess board which constitute valid knight moves.

- **valid_tour** is a set containing all sequences of squares which represent valid, but not necessarily complete, tours for the knight. An incomplete (or partial) tour fulfils the requirements of a knight’s tour\(^6\), but may have any length \(\leq 64\).

---

\(^5\)The subtle difference between non-deterministic and provisional choice is that the latter are expected to be retained through refinement as they are not “implementor’s choices”. Reducing them for example in our development here could result in the implementation possibly not being able to find a solution despite one existing, and consequently the component failing to fulfil its contract of feasibility-preserving refinement.

\(^6\)Namely these are that consecutive squares visited have to represent a valid knights move, and that no square is visited more than once through the tour.
MACHINE KnightsTourSolver

CONCRETE_CONSTANTS

SQUARE

ABSTRACT_CONSTANTS

valid_move, valid_tour, valid_solution

... 

DEFINITIONS

row(ss) == ss / 8;
col(ss) == ss mod 8

Figure 7.3: CONSTANTS clauses of the KnightsTourSolver B machine.

• valid_solution is the set of all solutions for the Knight’s Tour problem, i.e. exactly those members of valid_tour which have the length 64 and thus traverse all the squares of the chess board.

Definitions

• row is a function (introduced as a definition) which maps elements from SQUARE to their respective row index \(\in \{0, 1, \ldots, 7\}\) on the chess board. The bottom row has the index 0.

• col is a function (introduced as a definition) which maps elements from SQUARE to their respective column index \(\in \{0, 1, \ldots, 7\}\) on the chess board. The left-most column has the index 0.

The constants row and col enable us to determine the physical row or column of a square on the chess board (given through an element of SQUARE). They are trivially defined using integer division and modulus. Note that we could equally well have introduced these two functions by suitable constants. The reason for not doing so is pragmatic: having them as definitions effectively simplifies respective proofs associated with the component, mainly because definitions are substituted before any theorem proving takes place. Functions as constants on the other hand require a posteriori particularisation if their application is to be eliminated within proofs.

In the PROPERTIES clause of the B machine we formalise the above descriptions in terms of suitable defining predicates. In the remainder of this section we will give extracts of the KnightsTourSolver specification in order to explain
each of its features separately; moreover Appendix D.1 contains the complete
mark-up of the B specification for the reader’s inspection.

We first define the set \( \text{SQUARE} \) to be the subset of natural numbers from 0
to 63 — we remind the reader that there are \( 64 = 8 \times 8 \) squares on a chess
board. For simplicity we start counting squares from 0, this also makes it easier to infer
the row and column of a square as previously explained.

\[
\text{PROPERTIES}
\]
\[
\text{SQUARE} = 0 \ldots 63 \land \ldots
\]

The \textit{valid\_moves} constant being a relation between squares records all possible
knight moves between any two squares of the chess board.

\[
\begin{align*}
\text{valid\_move} & \subseteq \text{SQUARE} \times \text{SQUARE} \land \\
\forall (ss, tt).&(ss \in \text{SQUARE} \land tt \in \text{SQUARE} \Rightarrow) \\
&(ss \mapsto tt \in \text{valid\_move} \\
\iff \\
&((\text{row}(ss) \geq 2 \land \text{col}(ss) \geq 1 \land tt = ss - 17) \lor \\
&((\text{row}(ss) \geq 2 \land \text{col}(ss) \leq 6 \land tt = ss - 15) \lor \\
&((\text{row}(ss) \geq 1 \land \text{col}(ss) \geq 2 \land tt = ss - 10) \lor \\
&((\text{row}(ss) \geq 1 \land \text{col}(ss) \leq 5 \land tt = ss - 6) \lor \\
&((\text{row}(ss) \leq 6 \land \text{col}(ss) \geq 2 \land tt = ss + 6) \lor \\
&((\text{row}(ss) \leq 6 \land \text{col}(ss) \leq 5 \land tt = ss + 10) \lor \\
&((\text{row}(ss) \leq 5 \land \text{col}(ss) \geq 1 \land tt = ss + 15) \lor \\
&((\text{row}(ss) \leq 5 \land \text{col}(ss) \leq 6 \land tt = ss + 17)))) \land \ldots
\end{align*}
\]

Each disjunct within the universal quantification describes one possibility of
moving the knights from square \( ss \). At maximum, there are 8 distinct knight
moves available e.g. from the centre squares of the chess board. From squares
closer to the borders, however, there are usually less (from a corner square the
knight has only two fields to move to). Examine, for example, the first disjunct

\[
\text{row}(ss) \geq 2 \land \text{col}(ss) \geq 1 \land tt = ss - 17
\]

Such accommodates the case of moving the knight two squares below, and one
square to the left. We express knight moves here in terms of displacement of
\textit{squares} for supposed simplicity. Alternatively it is conceivable to specify knight
moves in terms of row and column displacements instead making use of \textit{row}(tt)
and \textit{col}(tt), however this as well would complicate the proof obligations.
The valid\_tour constant is defined as a subset of injective sequences of squares. Injectivity here ensures that in a valid knight’s tour, no square is visited more than once. Additionally, we clearly only want to include those sequences in which consecutive elements represent valid knight moves on the chess board. In defining valid\_tour we therefore rely upon the previous definition of the constant valid\_moves.

$$\ldots$$

$$\text{valid\_tour} \subseteq \text{iseq}(\text{SQUARE}) \land$$

$$\forall tt.(tt \in \text{iseq}(\text{SQUARE}) \Rightarrow (tt \in \text{valid\_tour} \iff \forall ii.(ii \in 1 \ldots \text{size}(tt)-1 \Rightarrow tt(ii) \mapsto tt(ii+1) \in \text{valid\_move})))) \land \ldots$$

Finally the valid\_solution constant constituting the set of all solutions to the Knight’s Tour problem is specified. It includes exactly the members (sequences) from valid\_tour which cover all squares of the chess board, thus have the entire set SQUARE as their range:

$$\ldots$$

$$\text{valid\_solution} \subseteq \text{valid\_tour} \land$$

$$\forall tt.(tt \in \text{valid\_tour} \Rightarrow (tt \in \text{valid\_solution} \iff (\text{ran}(tt) = \text{SQUARE}))$$

Equipped with the valid\_solution constant it is not difficult now to express the solve operation, which as previously explained non-deterministically returns a solution to the Knight’s Tour problem starting from some specific square:

\begin{verbatim}
OPERATIONS
  tour ← solve(start) =
  PRE
    start ∈ SQUARE
  THEN
    ANY tt WHERE
      tt ∈ valid_solution ∧
      tt(1) = start
  THEN
    tour := tt
  END
END
END
\end{verbatim}
CHAPTER 7. REFINEMENT OF REVERSIBLE COMPUTATIONS

The solution is selected by means of an **ANY** construct. The purpose of the parameter **start** is to choose the initial square on which the knight is placed, therefore **start** must also be the first element of the sequence returned. Notice that if no solution exists from some given initial square, the above operation becomes infeasible.

From a mathematical point of view it is often not trivial to decide whether a particular instance of a search problem with some initial conditions has a solution, or not. Our development therefore deliberately entails the case ‘no solution found’. Therefore it actually lies outside the obligation (or contract) of the **KnightsTourSolver** component to guarantee the existence of a solution, or equivalently the feasibility of the **solve** operation. We could of course precondition the operation in a suitable way restricting its use to cases where a result can be expected, however this would only shift the problem to the caller of the operation who is now faced with the difficult task of establishing whether a solution exists to fulfill its contract in turn. If nothing else, the previous discussion illustrates that for reversible programming to be useful we have to anticipate infeasibility not just at the level of generalised substitution, but B operations, too.

As can be seen there is no state present in the **KnightsTourSolver** machine component, hence neither **VARIABLES**, **INVARIANT** or **INITIALISATION** clause are required. The solution is instantiated in “one shot” by appealing to (mathematical) constants which describe it purely in terms of properties, leaving aside the issue of how algorithmically it may be constructed. Whether the **solve** operation is feasible for a particular starting square depends on the contents of the set **valid_tour**. Clearly the component’s proof obligations won’t ensure that for example **valid_solution** might not be empty; indeed proving this could in any case anything but trivial.

In Appendix D.5 we include an alternative **generic** variant of the **KnightsTourSolver** specification which can handle arbitrary board sizes. This is not further discussed here, although the reader may be interested to examine it for comparison.

### 7.4.2 The Concrete Development

The concrete development is factored into the implementation **KnightsTourSolverI** refining the **KnightsTourSolver** specification, and the utility component **KnightsTourSolverUtil** imported by **KnightsTourSolverI**.

The additional utility machine **KnightsTourSolverUtil** serves two purposes: first it improves the readability and compactness of the (R)B0 code implementation of the **solve** operation, and secondly it structurally isolates within its operations the choice-making and guarded statements which are central to the reversible implementation of the algorithm. Indeed this particular design allows us to factor out into a separate component exactly those code elements which aren’t B0, but strictly RB0, and more importantly will give us a specimen and test bed for discussions in Section 7.7 where we investigate how a piecewise and
stepwise development methodology may be restored in the light of reversible computations and the feasibility-preserving, but non-monotonic, refinement relation $\sqsubseteq^\ast$.

Before examining the implementation $\text{KnightsTourSolverI}$ let us first examine the utility machine and its implementation whose mark-ups are included in Appendices D.3 and D.4.

**KnightsTourSolverUtil and KnightsTourSolverUtilI**

The B machine $\text{KnightsTourSolverUtil}$ essentially only provides two operations, namely $\text{select\_move}$ and $\text{check\_unvisited}$. It SEES the $\text{KnightsTourSolver}$ machine so it can refer to its concrete constant $\text{SQUARE}$. The first of these operations non-deterministically selects a move for the knight from a given location on the chess board. The second one is used to engage the run-time system in reverse execution if a selected move leads to a square already visited by the knight in its current tour.

The $\text{select\_move}$ operation is included in Fig. 7.4 for closer inspection. Note that the current position of the knight is passed to the operation with the argument $ss$. Although guarded and choice constructs are not explicitly mentioned in the operation’s body, they of course implicitly form part of it underlying the definition of the $\text{SELECT}$ construct in B-AMN [Abr96b]. Each individual predicate within the collection of $\text{WHEN} \ldots \text{THEN} \ldots$ fragments evaluates whether a particular move of the eight possible knight moves is feasible from the square $ss$. To do so we make use of the $\text{row}$ and $\text{col}$ functions which are defined as in $\text{KnightsTourSolver}$ to deduce the column and row of $ss$.

The second operation $\text{check\_unvisited}$ presented by Fig. 7.5 provokes reverse execution if the square $ss$ passed to the operation is already visited through the current tour $tt$, also passed to the operation. For a precondition the argument $tt$ must be a valid tour, hence an injective sequence of elements from $\text{SQUARE}$. The body of the operation similarly takes advantage of a $\text{SELECT}$ construct which here unfolds into a guarded substitution whose guard is true exactly if $ss \notin \text{ran}(tt)$, i.e. $ss$ is not an element of the tour sequence $tt$.

The implementation $\text{KnightsTourSolverUtilI}$ is in fact identical to its specifying machine, hence we won’t re-iterate its operations here. The only difference is that we use upper-case identifiers for the definitions of $\text{col}$ and $\text{row}$, this is merely to indicate that in the implementation they play the part of executable constructs to be translated into code, and not just as mathematical functions introduced for the sake of reasoning about the specification.

It should be mentioned here that without further adjustments we couldn’t have submitted the $\text{KnightsTourSolverUtilI}$ implementation to the B tools we use. This is certainly so since $\text{SELECT}$ constructs are not translatable by any of the tools, and hence not permitted in implementations in the classical theory of B. Consequently our utility component fails the B0 checks of the Atelier B, and would also have been rejected in the analysis phase if we had decided to
\[
rr \leftarrow \text{select\_move}(ss) =
\]

\[
\begin{align*}
\text{PRE} \\
ss \in SQUARE \\
\text{THEN} \\
\text{SELECT} \quad & \quad \text{row}(ss) \geq 2 \land \text{col}(ss) \geq 1 \quad \text{THEN} \\
rr := & \quad ss - 17 \\
\text{WHEN} \quad & \quad \text{row}(ss) \geq 2 \land \text{col}(ss) \leq 6 \quad \text{THEN} \\
rr := & \quad ss - 15 \\
\text{WHEN} \quad & \quad \text{row}(ss) \geq 1 \land \text{col}(ss) \geq 2 \quad \text{THEN} \\
rr := & \quad ss - 10 \\
\text{WHEN} \quad & \quad \text{row}(ss) \geq 1 \land \text{col}(ss) \leq 5 \quad \text{THEN} \\
rr := & \quad ss - 6 \\
\text{WHEN} \quad & \quad \text{row}(ss) \leq 6 \land \text{col}(ss) \geq 2 \quad \text{THEN} \\
rr := & \quad ss + 6 \\
\text{WHEN} \quad & \quad \text{row}(ss) \leq 6 \land \text{col}(ss) \leq 5 \quad \text{THEN} \\
rr := & \quad ss + 10 \\
\text{WHEN} \quad & \quad \text{row}(ss) \leq 5 \land \text{col}(ss) \geq 1 \quad \text{THEN} \\
rr := & \quad ss + 15 \\
\text{WHEN} \quad & \quad \text{row}(ss) \leq 5 \land \text{col}(ss) \leq 6 \quad \text{THEN} \\
rr := & \quad ss + 17 \\
\text{END} \\
\text{END}
\end{align*}
\]

Figure 7.4: Operation \text{select\_move} in \text{KnightsTourSolverUtil}.

\[
\begin{align*}
\text{check\_unvisited}(ss, tt) = 
\text{PRE} \\
ss \in SQUARE \land tt \in \text{iseq}(SQUARE) \\
\text{THEN} \\
\text{SELECT} \quad ss \notin \text{ran}(tt) \text{ THEN skip END} \\
\text{END}
\end{align*}
\]

Figure 7.5: Operation \text{check\_unvisited} in \text{KnightsTourSolverUtil}.
use the B-Toolkit instead. Despite this, we managed to “coax” the Atelier B, which is the tool we used in preference to the B-Toolkit for its more sophisticated theorem prover, into accepting our development by disabling certain B0 and implementability checks in the corresponding resource file of the Knight’s Tour project. Although this still wouldn’t allow us to translate the Knight’s Tour development into any of the target languages supported by the tool, it at least enabled us to generate the refinement proof obligations, and thereby conduct a proof that KnightsTourSolverUtilI is a conventional refinement of specification machine. The previous investigation suggests that this is still valuable knowledge, although apparently such a proof can as well be obtained by restricting the non-determinism of the select_move operation. In such a case it also seems plausible that a solution to the problem may not be found anymore; this issue is not addressed by the conventional proof obligations.

The KnightsTourSolverI Implementation

The implementation KnightsTourSolverI refines the KnightsTourSolver specification, and imports the already discussed utility component KnightsTourSolverUtil. Its complete document can be found in Appendix D.2.

At first, values have to be given in KnightsTourSolverI to the concrete constants introduced in the specification. This only includes the set SQUARE; note that since parameters and return values of operations refer to it, it makes sense to make it a concrete (thus implementable) constant. The valuation of SQUARE is in fact not different from its defining predicate in the PROPERTIES clause of the abstract machine, i.e. the set of numbers from 0 to 63. The VALUES clause of the implementation therefore reads as

\[
\text{VALUES} \\
\text{SQUARE} = 0 \ldots 63
\]

The only remaining part of KnightsTourSolverI left for discussion now is the (R)B0 implementation of the solve operation. It is presented by the following AMN fragment:
Two local variables \( ss \) and \( length \) are declared within the operation. The first one \( ss \) is used to store the next subsequent square the knight moves to, and the second one \( length \) to record the length of the current tour under construction; moreover the return variable \( tour \) is used for holding the constructed tour — note that it is not necessary to introduce a another local variable for this purpose. Conventionally \( tour \) being of type \( \text{seq}(\text{SQUARE}) \) is not implementable, in RB0 however we extend the permissible types of concrete variables to include sets and relations as they are supported by our run-time system [SZ02, Sto06].

Once having initialised \( tour \) with a singleton sequence containing only the starting square \( start \), the solution is constructed iteratively through a while loop in which each iteration adds another square (or knight’s move). The statements executed in the loop body are the following:

1. \( ss \leftarrow \text{select\_move}(\text{last}(tour)) \)
2. \( \text{check\_unvisited}(ss, tour) \)
3. \( tour := tour \leftarrow ss \)
4. \( length := length + 1 \)

The first statement selects the next square the knight potentially can move to from its current position. The current position of the knight is the last element
of the tour sequence under construction. As can be seen, the selection process is propagated to the select_move operation of the KnightsTourSolverUtil component which performs it in a non-deterministic fashion. The second statement, also relying on KnightsTourSolverUtil by invoking the check_unvisited operation, provokes reverse execution if the previously selected square ss has already been visited through the current tour. Having reached the third statement we know for sure that this is not the case, and thus append ss to the tour sequence. Finally the forth statement increments the length variable which is useful for the loop to determine when iteration must terminate, and consequently the constructed tour sequence covers all the squares of the chess board. Besides length enables us to provide a variant for the loop to prove termination. Strictly speaking, however, we could infer the length of the current tour directly from the tour sequence.

To be able to prove that the loop in fact constructs a valid Knight’s Tour starting from the square start, we have to state in the INVARIANT clause of the loop that

\[ \text{tour}(1) = \text{start} \land \text{tour} \in \text{valid\_tour}. \]

Upon termination of the loop moreover we have \( \neg (\text{length} \leq 64) \) from the loop guard being false, which gives us along with the invariant conjunct

\[ \text{length} \in 1..64 \land \text{length} = \text{size}(\text{tour}) \]

that \( \text{size}(\text{tour}) = 64 \), thus the constructed tour is complete and indeed a solution to the Knight’s Tour problem. The other conjunct \( \text{tour} \in \text{iseq}(\text{SQUARE}) \) aids the invariant preservation proof.

**Final Remark.** The reader’s attention may be directed to how concisely the solution algorithm here can be specified making use of guards and choice, and their operational interpretation in terms of reversible computation. This not only simplifies the task of the implementor, but also very likely reduces the proof effort in discharging refinement proof obligations associated with the component’s implementation. A comparative development using a classical backtracking approach is a potential subject for future work, and not included in this PhD thesis.

### 7.4.3 Conventional Refinement Proof

In this section we limit our attention to the classical B refinement proof obligations generated by the Atelier B for the Knight’s Tour development. These guaranteed that if the solve operation delivers a solution it will be a valid knight’s tour according to the specification in the abstract component. Clearly this also implies that the implementation of solve terminates. What we don’t prove here is that the solve operation doesn’t behave miraculously where a solution could
possibly be generated.

To ensure the latter we would have to go a step further in the light of what has been explained in Section 7.3, namely guaranteeing that the implementation of the `solve` operation is not merely a conventional refinement, but also a valid \*refinement of its specification. Intuitively, by inspecting the B components of the development, we should be able to convince ourselves that this in fact ought to be the case. In Section 7.6 we will moreover investigate how this may be proved in practice, but this doesn’t concern us in this section.

In attempting a complete proof of the Knight’s Tour development, after the generation of proof obligations we first made use of the automatic theorem prover of the Atelier B (force 0 & 1) in order to discharge mechanically as many proof obligations as possible.

It showed that the only component which produced proof obligations that couldn’t be automatically discharged was `KnightsTourSolverI`. The following tables document the status of all components with non-obvious proof obligations after the automatic prover has been invoked:

**Component Status of KnightsTourSolverUtilI**

<table>
<thead>
<tr>
<th></th>
<th>NbOv</th>
<th>NbPO</th>
<th>NbPri</th>
<th>NbPRA</th>
<th>NbUn</th>
<th>%Pr</th>
</tr>
</thead>
<tbody>
<tr>
<td>ValuesLemmas</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>100</td>
</tr>
<tr>
<td>Initialisation</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>100</td>
</tr>
<tr>
<td>select_move</td>
<td>18</td>
<td>56</td>
<td>0</td>
<td>56</td>
<td>0</td>
<td>100</td>
</tr>
<tr>
<td>check_unvisited</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>100</td>
</tr>
<tr>
<td>KnightsTourSolverUtilI</td>
<td>24</td>
<td>56</td>
<td>0</td>
<td>56</td>
<td>0</td>
<td>100</td>
</tr>
</tbody>
</table>

**Component Status of KnightsTourSolverI**

<table>
<thead>
<tr>
<th></th>
<th>NbOv</th>
<th>NbPO</th>
<th>NbPri</th>
<th>NbPRA</th>
<th>NbUn</th>
<th>%Pr</th>
</tr>
</thead>
<tbody>
<tr>
<td>ValuesLemmas</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>...</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>100</td>
</tr>
<tr>
<td>Initialisation</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>100</td>
</tr>
<tr>
<td>solve</td>
<td>29</td>
<td>121</td>
<td>0</td>
<td>77</td>
<td>44</td>
<td>63</td>
</tr>
<tr>
<td>KnightsTourSolverI</td>
<td>32</td>
<td>122</td>
<td>0</td>
<td>77</td>
<td>45</td>
<td>63</td>
</tr>
</tbody>
</table>

Note that for readability we here changed the name of the table row labelled `InstanciatedConstraintsLemmas` to “...”, since no proof obligations fell in
CHAPTER 7. REFINEMENT OF REVERSIBLE COMPUTATIONS

127

this category it unnecessarily widened the table.

These two tables illustrated that the only component leaving undischarged proof obligations was KnightsTourSolverI. 45 of its 122 non-obvious proof obligations couldn’t be discharged automatically.

We subsequently managed to discharge 43 of the 45 remaining proof obligations using the interactive prover interface Click’n Prove developed by Cansell and Abrial[AC03]. Click’n Prove is part of Clearsy’s freely available B4free distribution of the Atelier B which provides a stripped-down version of the bbatch tool — the underlying command-line interface of the Atelier B. This could all be done without supplying any additional user proof rules7, merely exploiting the features and capabilities of Click’n Prove. The only two remaining undischarged proof obligations were one ValuesLemma ensuring that valuation of concrete constants doesn’t contradict the conditions specified in the PROPERTIES clause of the abstract specification, and a left-over PO resulting from the refinement proof of the solve operation. The latter ensured that the value returned by the refinement of solve is indeed one that may also be delivered by abstraction specification of the operation.

In both cases we convinced ourselves that a proof should potentially be possible. For the first remaining PO (machine context) this seems fairly obvious, hence we didn’t put any more effort into discharging it mechanically. The second one was a little trickier since neither the Atelier B, nor the Click’n Prove tool seemed to have a sufficient rule base to establish the truth of a sub-goal within the proof, despite that, intuitively it may have appeared trivial. Although we managed to define and incorporate a custom rule which would have filled this gap, there has been a further problem with applying modus ponens on some hypotheses resulting from particularisation. This was apparently a bug in the Atelier B, having however informally established the truth of both proof obligations we didn’t investigated this technical problem any further i.e. with Clearsy — the company who developed the Atelier B.

Lastly, we give the the component status of KnightsTourSolverI after having carried out the interactive proof:

| | NbObv | NbPO | NbPri | NbPra | NbUn | %Pr |
| | | | | | | |
| ValuesLemmas | 2 | 1 | 0 | 0 | 1 | 0 |
| Initialisation | 1 | 0 | 0 | 0 | 0 | 100 |
| solve | 29 | 121 | 43 | 77 | 1 | 99 |
| KnightsTourSolverI | 32 | 122 | 43 | 77 | 2 | 98 |

7Before Click’n Prove was developed and openly available, we managed to construct a partial proof using the interactive theorem prover of the Atelier B. For this we had to add various custom proof rules in order to aid the automatic proof steps. The corresponding theory file is included for reference in Appendix E.1.
In the next section of the Knight’s Tour case study we will provide a few general comments alongside our specific account of experiences using the Click’n Prove interface. Since Click’n Prove is still a fairly recent development, and supposedly not used to any major extent at present even in academic circles, our documented first-hand experience here may be of value. What we won’t attempt is to give a detailed account of every interactive proof performed, this is outside the objective of the thesis and besides is unlikely to bear much interest for the reader.

7.4.4 The Click’n Prove Tool

Unlike the interactive theorem prover of the Atelier B which is essentially text-based, Click’n Prove provides a graphical user interface integrated into XEmacs, a widely-known text editor in the Unix community.

Thus instead of having to type in instructions for the prover manually, formulæ can be marked within filtered lists hypotheses, and associated commands may be executed by means of a mouse click from a menu of options. The commands supported by Click’n Prove are fundamentally all those supplied by the Atelier B interactive theorem prover (the only one we missed was the \texttt{pc} command for loading and compiling user theories associated with a component). In this way the invocation of commands showed to be much quicker since the parameters of any command can be conveniently selected as sub-expressions of hypotheses, and copied into a designated text field. Other commands such as equal hypothesis, expression and predicate abstraction, or particularisation are directly executable through buttons displayed next to the respective hypothesis, i.e. the hypothesis itself doesn’t have to be typed in as it is the case with the interactive theorem prover of the Atelier B. Listing of all hypotheses containing a certain sub-expression may as well be done via a single mouse click.

What ultimately renders Click’n Prove more efficient than the Atelier B interactive prover is that the user can exercise subtle control over the hypotheses included in the assumptions for each intermediate invocation of the automatic predicate prover on a sub-goal.

It seems that often failure to prove a certain theorem automatically is caused by a large number of sometimes irrelevant initial hypotheses, and subsequent “noise” generated from these disrupting the mechanical search process of finding a proof.

Click’n Prove counteracts this problem by allowing the user to selectively filter hypotheses for specific proof steps. Our experience obtained from the Knight’s Tour case study showed the the performance of the predicate prover considerably increases if only confronted with a small number of initial hypothesis which nevertheless are sufficient to establish the sub-goal in question. There where only rare occasions on which this strategy didn’t lead to immediate success, this was more likely due to a short-coming in the default rule base.

When using the B-Toolkit or Atelier B to formally develop software it is fairly
common to overcome a poor performance of the automatic provers in discharging proof obligations by manually extending the rule base through user theories. Our experience using Click’n Prove however showed that such extensions in most cases are redundant from a proof-feasibility point of view. This is underpinned by our first attempts to obtain a complete proof for the Knight’s Tour development which involved the manually addition of 16 custom proof rules included in Appendix E.1. Still we had to cheat a little by means of a theory Lemma which discharge a difficult prove step vacuously that otherwise would be provable from the hypotheses. With Click’n Prove no additional proof rules at all where necessary to discharge 95% of the remaining proof obligations (43 out of 45).

We must admit that owing to Click’n Prove being more recently released, its default rule base may be superior to that of the Atelier B 3.6; this was also confirmed by our experience encountering proofs which succeeded immediately using the pr command8 in Click’n Prove, but failed with pr in the Atelier B. To conclude the classical refinement proof of the Knight’s Tour case study, a few examples of interactively discharged proof obligations via Click’n Prove are given next.

Examples of Interactive Proofs with Click’n Prove

Probably the most interesting PO emerging from the KnightsTourSolverI component was the following:

$$tour$1 \leftarrow \text{last}(tour$1) − 17 \in \text{valid}_\text{tour}$$

Here tour$1$ is the tour sequence constructed through the iterations of the while loop of the solve operation. The proof obligation results form the third condition of the loop invariant stating that tour$1$ extended with last(tour$1) − 17 has to remain a valid knights tour (assuming it was one before). This reflects the new value of the tour sequence after execution of the loop body with a particular choice of move being taken. Here the knight’s move with a displacement of $−17$ squares is considered, or in other words moving the knight two rows below and one column to the left.

The proof tree generated by Click’n Prove when performing the interactive proof of this PO is included in Appendix E.2, unfortunately we cannot present the proof in a marked-up linear form since neither the Atelier B, nor Click’n Prove supports the generation of proof documents e.g. similar to the B-Toolkit. Hence the account given of the interactive proof is the content of the ‘proof tree’ buffer after successful completion within Click’n Prove. A few features of Click’n Prove which we had to use for this proof were particularisation of universal quantifications, modus ponens on hypotheses, selective filtering of hypotheses for intermediate proof steps, proof by cases and expression abstraction.

---

8 The pr command invokes the standard automatic prover of the Atelier B or B4free within interactive proves.
A further PO worth examining (which was indeed one of the two we didn’t discharge mechanically) is the following:

\[ \text{tourz}$^7777 \in \text{valid\_solution} \]

The purpose of this PO is to conciliate the return value of the concrete \texttt{solve} operation with the one of the abstract specification of this operation. Implied by the loop invariant we immediately have that \texttt{tourz}$^7777 \in \text{valid\_tour}. What is left to show for \texttt{tourz}$^7777 to be also a member of \texttt{valid\_solution} is that its range must be equal to \texttt{SQUARE}. The loop guard being false after execution of the loop implies \(^\neg (\text{length} < 64)\) which along with the invariant \texttt{length} \in 1..64 gives us \texttt{length} = 64. Since the invariant also mentions \texttt{length} = \texttt{size(tourz}$^7777) we subsequently can show that \texttt{size(tourz}$^7777) = 64.

This together with \texttt{tourz}$^7777 being an injective sequence of elements from \texttt{SQUARE} should allow us to conclude that \texttt{ran(tourz}$^7777) = \texttt{SQUARE} because \texttt{card(SQUARE)} = 64. The mechanical proof step however couldn’t establish this, despite this the evidence given here will hopefully convince the reader that this PO in fact is provable.

We will leave it with these two examples of proof obligations illustrating the interactive proof of the Knight’s Tour development using Click’n Prove.

### 7.4.5 Translation and Execution

The thus obtained implementation of the \texttt{KnightsTourSolver} component clearly is not translatable into executable code with the current set of B tools, although we managed configure the Atelier B to generate proof obligations and succeeded in discharging them using the commonly available tools.

This is where the practical side of this PhD investigation ties in. Here the focus was to develop appropriate tools to make such a translation possible. There are two major software applications which this PhD investigation contributed to, one is the Reversible Virtual Machine\cite{Sto06} (RVM), and the other the OpenB Java tool library.

The RVM was primarily developed by Bill Stoddart with the author of thesis providing the core implementation for the set kernel, i.e. an efficient, general and reversible implementation of mathematical set, and moreover interactive debugging facilities for the RVM through the Heapwatch utility. OpenB was developed solely by the author, however is still in a proto-type stage. Eventually these tools are meant to be linked together providing a coherent framework for automatic translation of RBO implementations such as, for example, the \texttt{solve} operation of \texttt{KnightsTourSolverI}. For the current work we had to resolve to translate the RB0 implementation of the \texttt{solve} operation manually by hand.
7.5 Loop Feasibility

In this section we present and prove theorems which allow us to generally infer the feasibility of a while loop by means of a fixed point. The Knight’s Tour case study should make clear why there is need to do this in particular if we wanted to show that the concrete implementation of the solve operation is not just a conventional, but also a *-refinement of its abstract specification.

The WHILE \( G \) DO \( S \) END iteration construct is part of the AMN used only in B implementations. Its operational interpretation is that it repeatedly executes the loop body \( S \) as long as the guard \( G \) is true. If \( G \) is false the construct successfully terminates.

When we utilise while-loop constructs in B implementations we generally don’t have to worry about their feasibility property. By restricting ourselves to an executable subset of AMN we can rely on the feasibility of the loop body \( S \). Moreover, if the loop body \( S \) is always feasible then the loop statement as a whole remains feasible. Note, that feasibility shouldn’t be confused with the potential of non-termination, e.g. a non-terminating while loop is yet feasible.

The situation becomes more difficult if we can’t rely on the feasibility of the loop body and since in our theory of refinement we extend the implementation language with constructs that may lead to partial feasibility, it becomes essential to re-investigate loops under those “more relaxed” conditions. In the following we will derive the feasibility of a B while loop based on its fixed-point representation.

Let us refer to the following B loop construct as LOOP:

\[
\text{LOOP} = \text{WHILE } G \text{ DO } S \text{ END}
\]

According to Abrial’s account of while loops in [Abr96b] we can represent LOOP by the following construction involving the transitive opening \(^9\):

\[
\text{LOOP} = (G \implies S)^\wedge; \neg G \implies \text{skip}
\]

The operational interpretation of this formula is that first \( (G \implies S)^\wedge \) leaves

\(^9\)The transitive opening \( S^\wedge \) of a substitution \( S \) is the weakest (least refined) substitution satisfying the fixed-point equation \( X = (S; X) \] skip. It corresponds to Back and von Wright’s [BW98] strong iteration \( S^\omega \) of a monotonic predicate transformer \( S \). Furthermore, Abrial’s characterisation of a while loop here is equivalent to the one given by Back and von Wright in their Lemma 21.8 on p354 of [BW98].
us the choice of how many times we want to execute the loop body guarded by $G$, and secondly $G \rightarrow \text{skip}$ filters out all those cases (by making continuation infeasible) where we terminated the loop prematurely, e.g. where $G$ still holds after executing $S$ a certain number of times, respectively. The transitive opening also embraces the possibility of infinite repetition, that is in cases where execution of the substitution concerned always remains feasible. Thus is $G$ holds after executing $S$ indefinitely we have non-termination of $(G \rightarrow S)\rightarrow$ and thus the loop.

Before we derive the feasibility of our loop, we establish two useful lemmas.

**Lemma 56.** $\text{fis}(S ; T) = \neg \wp(S, \neg \text{fis}(T))$

**Proof.**

$\text{fis}(S ; T)$

$= \text{"Defn of fis"}$

$\neg \wp(S ; T, \neg \text{false})$

$= \text{"Sequential composition axiom"}$

$\neg \wp(S, \wp(T, \neg \text{false}))$

$= \text{"Logic"}$

$\neg \wp(S, \neg \neg \wp(T, \neg \text{false}))$

$= \text{"Defn of fis"}$

$\neg \wp(S, \neg \text{fis}(T))$ \hfill $\square$.

**Lemma 57.** $\wp(S^\wedge, Q) = \nu X \bullet \wp(S, X) \land Q$ for $S$ being a generalised substitution. Here, $\nu X \bullet f(X)$ designates the strongest fixed point of a function $f$ on predicates, which is the strongest predicate $X$ such that $f(X) = X$ holds.

We won’t include a proof but a discussion and formal justification can be found in Lemma 21.1 on p348 of [BW98].

Equipped with these two lemmas we can now prove the following theorem about the feasibility of $\text{LOOP}$.

**Theorem 58.** $\text{fis}(\text{LOOP}) = \neg \nu X \bullet G \land \wp(S, X)$

**Proof.**

$\text{fis}(\text{LOOP})$

$= \text{"Fixed-point characterisation"}$

$\text{fis}((G \rightarrow S)^\wedge ; \neg G \rightarrow \text{skip})$

$= \text{"Lemma 56"}$

$\neg \wp((G \rightarrow S)^\wedge, \neg \text{fis}(\neg G \rightarrow \text{skip}))$
"Defn of fis"
\[ \neg \text{wp}((G \implies S)^\land, \neg \text{wp}(\neg G \implies \text{skip, false})) \]

"Guarded substitution axiom, logic"
\[ \neg \text{wp}((G \implies S)^\land, \neg G \implies \text{wp}(\text{skip, false})) \]

"Axiom of skip"
\[ \neg \text{wp}((G \implies S)^\land, \neg G \implies \text{false}) \]

"Logic"
\[ \neg \text{wp}((G \implies S)^\land, G) \]

"Lemma 57"
\[ \nu X \bullet \text{wp}(G \implies S, X) \land G \]

"Guarded substitution axiom"
\[ \nu X \bullet (G \implies \text{wp}(S, X)) \land G \]

"Logic"
\[ \nu X \bullet G \land \text{wp}(S, X) \quad \square. \]

Alternatively, we may also represent the \text{fis(LOOP)} as the weakest fixed point of some function \( f \) by appealing to the duality property of the implication ordering. We need the following lemma in order to proceed:

\textbf{Lemma 59.} \( \nu X \bullet f(X) = \neg \mu X \bullet \neg f(\neg X) \)

This equality is justified in section 2.7 of [HJ98], so we omit a proof here.

It leads to the following corollary expressing the feasibility of \text{LOOP} as the weakest fixed point:

\textbf{Corollary 60.} \( \text{fis(LOOP)} = \mu X \bullet G \implies \neg \text{wp}(S, \neg X) \)

\textit{Proof.} \( \text{fis(LOOP)} \)
\[ \nu X \bullet G \land \text{wp}(S, X) \]

\textbf{Lemma 59}"
\[ \neg \mu X \bullet \neg (G \land \text{wp}(S, \neg X)) \]

"Logic"
\[ \mu X \bullet G \implies \neg \text{wp}(S, \neg X) \quad \square. \]

Note that the loop body of the \texttt{find_tour} implementation is infeasible whenever the knight cannot move to a new square. Proving the feasibility of the \texttt{find_tour} loop is equivalent to proving a Knight’s Tour exists. We would not expect this to be an easy task. On the other hand the approach we have used does enable us to demonstrate the feasibility of some simple loop structures.
7.6 Proving *-refinement of the Knight’s Tour

With Theorem 58 we can in theory calculate the feasibility of any given while loop by means of a fixed-point equation in predicates. This is very valuable from a mathematical viewpoint, however in practice when reasoning about programs we usually don’t refer to a loop through its fixed-point semantics, and clearly we don’t do so when performing formal software development using the B Method. Instead the familiar fundamental invariance theorem \cite{Dij76} is employed enabling us to reason about programs employing loops \textbf{without} having to appeal to the fixed-point semantics of iteration. This is achieved by manually providing a suitable loop invariant and variant. The invariant has to hold upon entry of the loop, and crucially must be preserved by the iterations of the loop, namely the execution of the loop body \textit{when the guard holds}. It further must be strong enough to imply the post-condition \( Q \) which we desire to verify to be true after execution of the loop construct \textit{when the guard doesn’t hold}. The variant on the other hand is critical to proving termination. This is done by showing it strictly decreases with each iteration of the loop body, and termination is implied (i.e. the loop guard becomes false) when it reaches 0. Additionally, as moreover is the case in B, we have to establish (from the invariant) that the loop variant always remains a positive natural number as long as the loop body continues to executes to make the previous argument work.

The question we shall ask ourselves before tackling the problem of *-refinement of the Knight’s Tour implementation is whether a similar approach could be employed in order to prove the feasibility of any given loop constructs. We first suggest and informally justify a general method which taken by itself can lead to success in proving feasibility of the while loop statement. In more interesting scenarios, however, such as the Knight’s Tour implementation, we shall argue that this method makes too strong assumptions, and thus cannot directly be used in this similar cases. Instead we suggest a different method which in specific cases allows us to establish overall loop feasibility especially when backtracking through reverse execution is expected to occur across the syntactic boundaries of the loop body. Such is the case when iterations of the loop itself would have to be reversed (compare this to the case where we anticipate ‘local’ backtracking within the loop body, however in each iteration would nevertheless succeed in finding a feasible path through it). Then the general method is not applicable anymore, and we have to resolve to a “trick” which will be explained in due course in this section. The overall achievement here is to lay the foundations for proving feasibility-preserving refinement of the Knight’s Tour solver implementation, and thus finalise the correctness proof of the Knight’s Tour case study.
7.6.1 The General Method

Consider the following general loop construct as part of B’s Abstract Machine Notation:

\[
\text{LOOP} \triangleq \begin{cases} \\
\text{WHILE } G \text{ DO } \\
S \\
\text{INVARIANT} \\
I \\
\text{VARIANT} \\
v \\
\text{END} \\
\end{cases}
\]

Our first attempt to show that \( \text{LOOP} \) is feasible, hence not behaving in a miraculous way prompting “\textbf{ko}” when executed on our reversible virtual machine, is to require the following as an additional PO for the while loop alongside the ones which are typically generated in B.

**Proof Obligation 1.** To establish the overall feasibility of a while loop with its loop guard being \( G \), body \( S \), and loop invariant being \( I \), it is sufficient to discharge the following proof obligation in addition to the standard B proof obligations for the loop construct:

\[
G \land I \Rightarrow \text{fis}(S)
\]

Note that \( \text{fis}(S) \) may be inferred in the usual way, i.e. \( \text{fis}(S) \triangleq \neg \text{wp}(S, \text{false}) \).

Informally, we then argue along the same lines as the fundamental invariance theorem: Since \( I \) holds initially, and moreover after each iteration of the loop, it implies that moreover \( \text{fis}(S) \) must hold initially and after each repetition of the loop body \( S \). By strengthening the antecedent with \( G \) we drop the requirement of loop body feasibility in cases where the loop body is not entered, an extreme example of such a case is given by \textbf{WHILE} false DO \textbf{magic} \textbf{END} which, as a statement, is clearly feasible despite its infeasible body, simply for the loop body never being entered. The inductive argument goes along the lines of asserting that \( I \) (and thus \( \text{fis}(S) \)) holds initially, after each iteration of the body \( S \), and consequently must hold until the loop construct terminates. It follows that the while loop statement as a whole, too, must be feasible from any initial state satisfying \( I \), and we may generally conclude that \( \text{fis}(\text{LOOP}) \) is true — providing all standard proof obligations of the loop including Proof Obligation 1 are discharged; a rigorous proof of this argument is omitted here.

As concise and appealing the condition given by Proof Obligation 1 looks, it is unfortunately in many cases too strong. More precisely, it will not always be possible to find an invariant which implies feasibility of the loop body \( S \). If we’d manage to find such an invariant, the operational consequence of this would
necessarily be that no backtracking through reversing (i.e. via guards and choice) could take place across the iterations of the loop, such would clearly contradict the proof of $\text{fis}(S)$ under the assumption $I$. Looking at the implementation of the `solve` operation in the Knight’s Tour case study (see Section 7.4.2), we should be able to convince ourselves that reversing across iterations of the loop is nevertheless essential here, namely when emerging from a dead end during the search process. Then, clearly iterations of the loop have to be undone in order to revise previous choices of moves taken at earlier points in the search.

The conclusions we may draw so far are constitute that the method proposed in this section would be directly applicable in cases where no reverse execution is anticipated across the boundaries of the loop body. In other cases, which are stipulated to be probably the more relevant and interesting ones when it comes to developing software using the paradigm of reversible computation presented in this thesis, we are bound to explore alternative solutions as Proof Obligation 1 cannot lead to success.

Unfortunately there doesn’t seem to be a simple and uniform approach to tackle such cases e.g. by suitably modifying the fundamental invariance theorem as we previously attempted. What, however, we will illustrate in what follows in the next section is a specific method and its mathematical justification which, in the case of the Knight’s Tour and presumably many similar-type applications, *pragmatically* enables us to prove feasibility of the implementing loop providing that the abstraction of the respective operation is feasible. It will emerge that the attempts towards proving loop feasibility in this section moreover haven’t been futile, and thus can be reused by the specific method within a modified context to the ends of actually proving feasibility of the Knight’s Tour `solve` operation implementation. Note that the endeavour of proving loop feasibility is primarily directed by discharging the additional conjunct within the *refinement condition* Def. 54, namely that $\text{fis}(O_A) \Rightarrow \text{fis}(O_C)$ for some abstract operation $O_A$ and concrete **reversible** implementation of it $O_C$ within some B development. (Consequently we are not concerned about the `solve` operation in `KnightsTourSolverI` behaving miraculously if no solution to the problem existed, and thus its abstraction of the `solve` operation would be infeasible.)

### 7.6.2 The Specific Method

The idea behind the specific method we suggest for the sake of proving feasibility in the particular case of the Knight’s Tour solver implementation is to provide another B operation e.g. within its own component which we call a **super-concrete** refinement of the abstract operation. This operation is syntactically derived from the concrete operation in such a way as allowing us to infer in purely syntactic (or algebraic) terms that indeed it is a conventional refinement of the concrete operation; the important point here is that no additional proof effort ought to be required in order to determine that the super-concrete operation is a **conventional** refinement of the abstract operation, providing we have have already
proved that the concrete operation is a conventional refinement of the abstract operation. Note that the latter we have established already for the Knight’s Tour case study by discharging the conventional set of proof obligations for the \texttt{KnightsTourSolverI} refinement.

The crucial point is that whereas the loop body of the concrete solve operation makes choices \textit{blindly} and hence potentially may stumble into dead ends over the search process, the super-concrete operation exploits the \textit{a priori} existence of a solution and makes use of it for always making the “right” choices when executing the iterations of the loop that successively construct the knight’s tour solution sequence. It is obvious that in such a manner the super-concrete operation would never have to exercise any backtracking within the iterations of the loop, its loop body would always remain feasible. This argument suggests that for the super-concrete operation, the previously developed general method would indeed be applicable and sufficient despite its failing for the intermediate concrete operation.

Before we explain by example how the super-concrete operation is to be constructed, we first give a general proof argument in form of a theorem that the previously described approach would give us in the end what we desire, namely that the concrete operation is a $^*$-refinement of the abstract operation.

\textbf{Theorem 61.} Let $O_{A}$, $O_{C}$ and $O_{S}$ be generalised substitutions representing $B$ operations. We call $O_{A}$ the abstract operation, $O_{C}$ the concrete operation, and $O_{S}$ the super-concrete operation. Moreover assume that $O_{A} \subseteq O_{C} \subseteq O_{S}$, and additional that $\text{fis}(O_{A}) \Rightarrow \text{fis}(O_{S})$. From these assumptions follows that

$$O_{A} \sqsubseteq^* O_{C}$$

\textbf{Proof.} From $O_{C} \subseteq O_{S}$ we conclude that $\text{fis}(O_{S}) \Rightarrow \text{fis}(O_{C})$, i.e.

$$O_{C} \subseteq O_{S}$$

$\Rightarrow$ “Defn of $\subseteq$ (Def. 53)”

$$\forall \ Q \ \bullet \ \text{wp}(O_{C},\ Q) \Rightarrow \text{wp}(O_{S},\ Q)$$

$\Rightarrow$ “Particularisation with $Q = \text{false}$”

$$\text{wp}(O_{C},\ \text{false}) \Rightarrow \text{wp}(O_{S},\ \text{false})$$

$\Rightarrow$ “Logic”

$$\neg \text{wp}(O_{S},\ \text{false}) \Rightarrow \neg \text{wp}(O_{C},\ \text{false})$$

$\Rightarrow$ “Defn of fis”

$$\text{fis}(O_{S}) \Rightarrow \text{fis}(O_{C})$$

Along with the assumption $\text{fis}(O_{A}) \Rightarrow \text{fis}(O_{S})$ we have by transitivity of implication $\text{fis}(O_{A}) \Rightarrow \text{fis}(O_{C})$. From the assumption $O_{A} \sqsubseteq O_{C}$ and the definition of $^*$-refinement (Def. 54) we immediately conclude that $O_{A} \sqsubseteq^* O_{C}$. □

It thus shows that the only real effort we have to invest in addition to discharging the conventional refinement proof obligations is to prove that $\text{fis}(O_{A}) \Rightarrow$
fis(\(O_p_S\)) to obtain the desired result \(O_p_A \sqsubseteq^* O_p_C\). Remember that \(O_p_S\) is accordingly constructed so that \(O_p_C \sqsubseteq O_p_S\) is established more or less for free. It remains to examine how the construction of the super-concrete operation specifically is carried out.

In practice, we derive \(O_p_S\) from \(O_p_C\) by replacing the non-deterministic choices performed in the loop body of \(O_p_C\) with (deterministic) conditional constructs always selecting the ‘right’ choice to ensure always-feasible continuation of the computation. An instance of this strategy exemplified by the simplest of cases is given by

\[
BODY_C \triangleq S \parallel T \sqsubseteq \text{IF } b \text{ THEN } S \text{ ELSE } T \text{ END} \triangleq BODY_S
\]

where \(BODY_C\) is the loop body of the concrete operation, and \(BODY_S\) the loop body of the super-concrete operation. The refinement \(BODY_C \sqsubseteq BODY_S\) here is easy to verify. In real-life situations however \(BODY_C\) could be more complex containing multiple choices (8 for example as in the body of the implementation of the solve in KnightsTourSolverI), sequentially composed statements, etc. Nevertheless the general argument for \(BODY_C \sqsubseteq BODY_S\) still holds in the presence of such complexities, mainly due to monotonicity of B-AMN constructs with respect to conventional refinement. Hence if we construct the super-concrete operation in the prescribed way above we should first have \(BODY_C \sqsubseteq BODY_S\), consequently

\[
\text{WHILE } G \text{ DO } BODY_C \text{ END } \sqsubseteq \text{WHILE } G \text{ DO } BODY_S \text{ END}
\]

and ultimately \(O_p_S \sqsubseteq O_p_C\). How we specify the condition \(b\) doesn’t influence the previous justification for \(O_p_C \sqsubseteq O_p_S\), however it is crucial to the remainder of the proof which requires us to show that fis(\(O_p_A\)) \(\Rightarrow\) fis(\(O_p_S\)).

In the abstract development of the Knight’s Tour solver, fis(\(O_p_A\)) is indeed equivalent to postulate the existence of a solution for the knight’s tour problem from a particular starting square. To refer to any such solution in the super-concrete refinement, one possibility is to introduce another abstract constant, let’s call it called solution, into the KnightsTourSolver machine:

```
\text{ABSTRACT\_CONSTANTS}
\text{. . . , solution}
\text{PROPERTIES}
\text{. . . }
\text{solution} \in \text{valid\_solution}
```

The abstract constant solution is subsequently used in the super-concrete operation to determine the next move for the knight after each iteration of the loop. In
consequence, the super-concrete operation effectively constructs the knight’s tour specified by the abstract constant \textit{solution}. It moreover does so without incurring of any backtracking. This is an improved situation since now we can be hopeful to apply the previously outline general method in Section 7.6.1 for proving loop feasibility to the loop of the super-concrete operation.

A possible outline of the super-concrete version of the Knight’s Tour \texttt{solve} operation is presented in the following:

\begin{verbatim}
OPERATIONS
  tour ← solve_super =
  VAR ss, length IN
  tour := [solution(1)];
  length := 1;
  WHILE length < 64 DO
    ss ← select_right_move(last(tour), length, solution);
    check_unvisited(ss, tour);
    tour := tour ← ss;
    length := length + 1
  INVARIANT
    tour(1) = start ∧
    tour ∈ iseq(SQUARE) ∧
    tour ∈ valid_tour ∧
    tour ⊆ solution ∧
    length ∈ 1 .. 64 ∧
    length = size(tour)
  VARIANT
    64 - length
END
END
\end{verbatim}

There are two major differences comparing the concrete \texttt{solve} operation implemented in \texttt{KnightsTourSolverI}, these have been highlighted by surrounding boxes\textsuperscript{10}. First, instead of calling the \texttt{selectMove} operation, the super-concrete operation instead calls an operation \texttt{select_right_move} (see Fig. 7.6) which is parameterised in terms of the current square of the knight, but also the current length of the tour sequence and postulated solution.

A second crucial alteration can be observed in the \texttt{INVARIANT} clause of the loop. The additional conjunct \texttt{tour ⊆ solution} strengthening the loop invariant asserts that the current tour sequence must be an initial segment of the \textit{solution} sequence. This is in particular important to prove the feasibility of the loop body in each iteration. Note that infeasibility in the super-concrete operation

\textsuperscript{10}Note that the super-concrete operation is also not parameterised with a starting square \textit{start}, this is for the sake of simplifying the presentation of the example.
can potentially still arise from the second statement of the loop body, namely the operation call to `check_unvisited(ss, tour).

Although no formal proof is included in the thesis showing that from the invariant of the super-concrete operation the feasibility of its loop body follows, let us at least convince ourselves informally that this is true. Infeasibility of the loop body can only emerge from the invocation of `check_unvisited` i.e. if the square `ss` returned by the former call to `select_right_move` was already contained in the tour sequence. We may refute this by appealing to the properties of `solution` being an element of `valid_solution`. The invariant tells us that the current tour sequence is an initial segment of the model solution `solution` including the first `length` elements, and the first statement of the `solve_super` together with the definition of the `select_right_move` (Fig. 7.6) operation yields that \( ss = solution(length + 1) \). This is sufficient to prove that `ss` is not already contained in the range of the current tour sequence, therefore the guard of the `check_unvisited` operation call will always evaluate to true.

The `select_right_move` is almost exactly derived in the way we previously described, meaning that non-deterministic choices are replaced by conditional statements. The formulation of the conditions is slightly awkward since we have to “reverse engineer” the choice taken to obtain the tour according to the `solution` argument, however the general principle should in essence become clear here. The fact that from the start we isolated the non-deterministic selection process of the solver implementation into the operation of a designated utility component did, as illustrated here, simplify the definition of the super-concrete operation, in particular only minor changes were required to the solve operation.

**Concluding Remarks**

A slight cosmetic flaw in the development we sketched out is that by adding the defining property `solution \in valid_solution` we effectively demand for overall consistency that at least one solution exists, in other words `valid_solution \neq \emptyset`. We previously explained that this would be too strong a claim which we wouldn’t want to burden the specifiers with proving. To avoid the respective proof obligation, alternatively we may define the `solution` constant in another B component particularly designated for holding it. An even more sophisticated alternative would be overall to do without the additional abstract constant, but introduce it instead locally in the super-concrete operation by encapsulating it in an `ANY` construct. Note that this would still allow us to infer syntactically that the super-concrete operation is a refinement of the concrete one.
rr ← select_right_move(ss, length, solution) =

PRE
ss ∈ SQUARE ∧ 
length ∈ 0 .. 63 ∧ 
solution ∈ valid_solution ∧ 
ss = solution(length)

THEN
IF solution(length+1) = ss - 17 THEN
    SELECT row(ss) ≥ 2 ∧ col(ss) ≥ 1 THEN
        rr := ss - 17
    END
ELSIF solution(length+1) = ss - 15 THEN
    SELECT row(ss) ≥ 2 ∧ col(ss) ≤ 6 THEN
        rr := ss - 15
    END
ELSIF solution(length+1) = ss - 10 THEN
    SELECT row(ss) ≥ 1 ∧ col(ss) ≥ 2 THEN
        rr := ss - 10
    END
ELSIF solution(length+1) = ss - 6 THEN
    SELECT row(ss) ≥ 1 ∧ col(ss) ≤ 5 THEN
        rr := ss - 6
    END
ELSIF solution(length+1) = ss + 6 THEN
    SELECT row(ss) ≤ 6 ∧ col(ss) ≥ 2 THEN
        rr := ss + 6
    END
ELSIF solution(length+1) = ss + 10 THEN
    SELECT row(ss) ≤ 6 ∧ col(ss) ≤ 5 THEN
        rr := ss + 10
    END
ELSIF solution(length+1) = ss + 15 THEN
    SELECT row(ss) ≤ 5 ∧ col(ss) ≥ 1 THEN
        rr := ss + 15
    END
ELSE /* By default solution(length+1) = ss + 17 here. */
    SELECT row(ss) ≤ 5 ∧ col(ss) ≤ 6 THEN
        rr := ss + 17
    END
END
END

Figure 7.6: Super-concrete refinement of the select_move operation.
7.7 Piecewise and Stepwise Development

The main issue we are faced with when developing software using reversible computations that exploit the backtracking nature of guards and choice is the problem of vacuous refinement. The feasibility-preserving *-refinement relation solves the problem of “over-refinement”, however is unfortunately not monotonic in all GSL constructors, i.e. in the in first argument of sequential composition.

The Knight’s Tour case study illustrates this in the following way: since we broke down the development into two parts, namely the main solver component KnightsTourSolver and the utility component KnightsTourSolverUtil, both these machines could be implemented separately, even possibly by different people. One person could decide to remove one or more of the 8 choices inherent in the select_move operation in KnightsTourSolverUtil, such would locally still fulfil the contract of *-refinement providing there is at least one choice left. The prospect however of the solve operation implementation in KnightsTourSolver finding a solution if exercising only a limited number of choices of moves would likely vanish. Note that this problem only occurs since the choice of move happens before a decision is made whether to continue or reverse execution, i.e. in the second statement of the solve operation.

A solution we suggest to this problem is that certain operations should be annotated in a B development as containing “provisional choices”. The non-determinism inherent in such operations we require to be retained in the refinement process. Not all operations in a possibly larger-scale B development clearly need to be subject to this; in some places it would certainly be appropriate and desirable to specify non-determinism with the intention of giving the implementor the opportunity to resolve it. The question of which operations are critical in the sense that they compromise feasibility-preserving refinement of other operations we claim could even be answered by some mechanical procedure inspecting the syntax of implementations, and determining operation calls occurring as the first argument of a sequential composition. Components could be marked as “safe” meaning that the implementor is at liberty to resolve non-determinism of their operations, or otherwise “unsafe”. The latter thus contains operations which at some point are called within the first argument of a sequential composition in some other B operation. In principle, it could still be possible to allow the implementor to refine operations in these components while genuinely reducing non-determinism. The consequence however is that the ‘hiding of the implementation’ methodology of the B Method has to be given up in such cases. This means that when importing an unsafe component genuinely refining one of its operations, the importing component actually has to look at the refinement rather than just the abstraction when generating the residual proof obligations for feasibility-preserving refinement.
7.8 Conclusions

In this chapter of the thesis illustrated in a substantial B development how our approach of reversible computation can be used to solve the classical chessboard problem of the Knight’s Tour. While doing so we investigated the implications of admitting infeasible constructs within implementations of B operations on refinement. This lead us to propose a stronger refinement relation in order to prevent vacuous refinement. The stronger refinement which we call $\ast$-refinement ensures that the concrete operation while being a classical refinement of the abstract operation cannot be less feasible than it. Thereby it allows us to reduce non-determinism inherent in the abstract operation, but not entirely remove it. We show that all constructs of the Generalised Substitution Language are monotonic with respect to feasibility-preserving refinement except for sequential composition in its first operand. We explain how the loss of monotonicity becomes a problem when following the piecewise and stepwise approach of B, and moreover how this problem may be counteracted by managing the refinement of operations whose non-determinism is interpreted as provisional choice rather than implementor’s choice in a controlled way.
Chapter 8

Conclusions and Future Work

8.1 Conclusions

In this PhD thesis we have investigated how the B Method for rigorous software construction can be adopted to specify and implement computations which exploit reversibility and thus can effectively take advantage of a reversible execution environment or platform. To make reversible programming more expressive and practically useful, we have developed a novel theory of prospective values for the Generalised Substitution Language which allows us to overcome the fundamental problem of the involuntary discarding of information during the intermediate reverse steps of a computation.

We began our investigation by first examining the concept of logical reversibility which enabled us to establish the important thermodynamic relationship between the minimum energy requirements of a computation, and the irrecoverable loss of information in its logical steps. We illustrated that by making the computation stepwise reversible, the required energy can at least theoretically be reduced to a one-off cost to be paid to initialise its memory to a known state; the reversible computation itself may then in theory be carried out with zero energy consumption. This insight is moreover of practical significance realising that the dissipation of energy is a major obstacle in increasing the efficiency of modern computing hardware, reversible architectures may offer some possibilities to address this problem in the future.

The fundamental method we proposed to integrating reversible computing into the B Method was to revoke the ‘law of the excluded miracle’, as Dijkstra called it, by allowing stand-alone guards and choice statements to occur in B implementations. So far infeasible computations have played an analogous rôle in theories of programming to imaginary numbers in analytical mathematics: although they are sometimes useful and even necessary to perform calculations, we would like to rid ourselves of them in the final answer. Where Dijkstra decided to altogether exclude them from his Guarded Command Language (GCL) and associated wp calculus [Dij76], Nelson first realised their importance as rendering the theory of the GCL in a more complete way [Nel89].
In this PhD investigation we have gone one step further by admitting them as executable statements and taking advantage of their operational interpretation in terms of backtracking. For the execution of such programs we have developed a designated software platform, the ‘Reversible Virtual Machine’ or short RVM.

To overcome a short-coming of the reversible approach implying that intermediate results of a computation are necessarily discarded prior to reversing, we proposed a new operator $S \diamond E$ yielding the prospective value of an expression $E$ after executing a computation $S$, however without incurring any side-effects. We showed how this operator can be used to define an alternative semantics for the Generalised Substitution Languages which is isomorphic to the one being based on weakest preconditions. In order to represent the outcome of all computations within the generalised substitution lattice at the level of expressions, we adopted bunch theory in our work but extended it to server our particular purpose. This endeavour resulted in the development of Improper Bunch Theory which moreover justified itself as a useful and comprehensive theory in its own right, tailored for future integration into the B Method and tools.

By means of a case study solving the Knight’s Tour problem we illustrated the implications on refinement of incorporating infeasible computations into $B^0\ddagger$. To prohibit vacuous refinement we proposed a stronger, feasibility-preserving refinement relation, however it turned out sequential composition in its first operand is not monotonic with respect to it. Thus we addressed how despite the loss of monotonicity a piecewise and stepwise development approach may be pursued. Importantly, we also exemplified how feasibility-preserving refinement can be proved practically in the context of this non-trivial B development in which the implementing operation utilises a loop construct which potentially may backtrack across the syntactic boundaries of the loop body. The method we have developed for constructing the respective proof can easily be adopted for analogue B developments of applications which encapsulate backtracking search algorithms.

**Applicability of Results**

Several contributions and results obtained in this PhD thesis may be relevant or applicable to other areas of computer science too. Reversibility in general has many more applications than just the one we outlined in Chapter 2, namely to lower the minimum theoretic energy requirements of a computation. Examples are systems that due to some condition or failure have to undo a series of steps or transactions, word processors which allow the user to revert to previous versions of a document or undo recent modifications, software debuggers, etc. In Chapter 7 we illustrated that in principle it is possible to use the B Method and tool support to develop applications specifying reversible behaviour, and we think that the results obtained here may be to a certain degree applicable (or otherwise

---

\(^1\)Note that $B^0$ is the subset of B-AMN conventionally permitted in implementations
could be generalised) to the more general field of safety-critical systems development using rigorous approaches, in particular formalisms which support some form of backtracking. Improper Bunch Theory in Chapter 5 was initially developed to formulate prospective-value semantics in the most concise way, however we moreover realised that it constitutes a powerful theory in its own right. For example, the presence of the improper bunch $\bot$ allows us to characterise the behaviour of any computation expressed in the Generalised Substitution Language purely as a function and thereby gives rise to a reconciliation between functional programming and sequential programming i.e. every GSL computation could be representable by a corresponding function and vice versa. Last but not least in Chapter 6 we developed a formal theory of prospective values which in this PhD investigation was primarily motivated to overcome a short-coming of our backtracking approach of not being able to construct all the results of a backtracking search as being defeated by the oblivious nature of reverse execution. It seems however that the thus introduced $S \diamond E$ operator is in fact a rather versatile animal as it could serve a dual purpose: on one hand we can use it to reason about programs, on another it fulfils useful functions if integrated into the syntax of the language itself. This realisation could be of interest in the area of programming languages and the construction of algorithms. There may be further applications of our work which haven’t occurred to us yet.

8.2 Future Work

From the theoretic and practical work carried out as part of this PhD investigation a series of possible future work projects have emerged which we want to outline in this section. Before doing so it should be mentioned that not all the work the author has invested over the period of time for this PhD, in particular the development of tools, is reflected and done justice in the thesis document. This in particular included the OpenB Java tool library which despite being in a publishable state we decided not to incorporate into the write-up of the thesis, partly for space and time considerations, but also since the contribution here was to primarily focus on theoretical aspects whereas the practical work might subsequently better be published as a succession of technical reports, hopefully sparking further interest in our work and tools within the formal methods research community. Indeed, a few such reports can already be found on our web-site at http://www.scm.tees.ac.uk/formalmethods/.

The need for future work with regards to the contribution of this PhD is in particular given since in many places the feeling that surfaced when developing the ideas of this PhD thesis has been that of opening a “Pandora’s Box”. For instance, admitting infeasible computations in implementations could in itself be taking as quite a radical step which one would expect not to integrate smoothly with the current set of methods for refinement and development. Where in classical theories of programming the guarantee of termination is an important issue to
focus on, in our theory of reversible programming with the GSL we have the addi-
tional complication of infeasibility at the top level to worry about, and whether
*-refinement can address this problem suitable in large-size developments still
remains to be practically explored. Another profound extension we propose is
to incorporate bunches into the theory of B. Bunches might have so far lead a
shadow existence in mathematics, possibly due to some criticism of the absence
of a model and the implied danger of inconsistency; however in our work we make
a strong case for their usefulness and hope that gradually they will become a
more established theory in computer science research.

We will now enumerate and elaborate on a few particular aspects which we
consider fruitful for follow-up research and future work:

**Tool Support: OpenB**

Despite already a considerable amount of work having gone into the development
of tools, namely the RVM and OpenB, a lot still remains to be done on this side
to support the development of reversible programs in a comprehensive method
and tools framework.

Whereas the RVM is currently in a reasonably mature state, OpenB has just
reached a proto-typical point in development where we can parse, type check and
translate a subset of the AMN into RVM Forth (the language processed by the
RVM). What eventually we would like to achieve is the automatic translation of
entire Atelier B projects with a minimum level of user interaction. This suggests
a graphical user interface for OpenB offering similar facilities as the Atelier B.
The effort which has gone into the OpenB tool library up to this point was
aimed at making it a generic set of components for developing any kind of B-
related tool, and thereby supporting the quick development and proto-typing of
tools supporting extensions to the B Method. In several aspects OpenB is even
superior to commercial tools such as the Atelier B and B-Toolkit e.g. as its type
checker extracts more information from the syntax, and hence depends less on
the inclusion of type-constraining predicates.

We eventually hope that by making OpenB commonly available under the
GPL at http://openb.scm.tess.ac.uk/openb/ other academics and formal
software developers will take an interest in it and actively contribute to its future
development. The author realised that the scale of this project would be too vast
to be undertaken by just one person, effectively the future vision for OpenB is
to re-implement most parts of the commercial B tools in a coherently designed
and well-documented open source software application being freely available for
academic and non-commercial use. The author has used his knowledge and ex-
perience in object-oriented programming and software engineering to produce a
set of components which may nicely act as a starting point for such an endeav-
our. However, the more extensions we integrated into OpenB (most recently we
added a rewrite mechanism which was intended to transform AMN operations
into the corresponding generalised substitutions, but indeed also showed to be
useful to perform simple proofs by rewrite), the more possibilities seem to open up, and conducting such a project within the scope of this PhD thesis was clearly infeasible.

**Integration of OpenB and the RVM**

As previously explained, it was decided that to achieve the goal of translating reversible B into executable code two applications were to be developed, namely the RVM, which acts as the target language, and OpenB, which provides translation support for RB0 into RVM Forth. These components have separately reached a state of maturity in which they are essentially fulfilling their role, however what hasn’t been entirely satisfactory is the implementation of an integrating environment that ideally hides from the user the underlying details of translation. We currently provide an “RVM Console” which links up OpenB with the Reversible Virtual Machine, and allows either to type in commands and pass them directly to the RVM, or switching into a mode which instead expects some fragment of AMN to be entered which is then parsed, type checked, translated, and propagated to the subordinate RVM process in its translated form. To investigate how this integration is best performed, how we can achieve platform independence, how errors are reported to the user, how applications translated in this way are deployed, etc. is still subject to future development.

**Build a Theory of Prospective-Value Semantics *ab initio***

Discussion with colleagues in the Teesside Formal Methods and Programming Research Group have raised the interesting issue whether we could approach pv semantics from a different angle not providing a closed-form definition, as done at the beginning of Chapter 6, but instead formulate certain healthiness conditions which characterise the subset of expression transformers that represent prospective-value transformers of GSL computations. Just as, for example, Hoare and He characterise the subclass of relations representing valid designs by means of 4 healthiness conditions H1 to H4 [HJ98], we believe we can do something similar in pv semantics by providing only one healthiness condition, namely

\[ S \circ E = (\lambda s \cdot E)(S \circ s) \]  

where \( s \) is the frame of \( S \) \((\text{ZHC})\) \hspace{0.5cm} (8.1)

Intuitively this condition states that the effect \( S \circ \_ \) may have on \( E \) is limited to changing the value of its frame variables, i.e. no specific assumptions about the syntax of \( E \) can be made.

Since the ideas regarding healthiness conditions in pv semantics haven’t sufficiently matured when writing up the thesis, we didn’t elaborate on them in Chapter 6. Nevertheless it would be beneficial prospective work to see how far we can go building a theory of prospective values using ZHC and what properties about prospective values can be derived from it. Moreover in what way does
ZHHC relates to the healthiness conditions typically encountered in wp semantics such as conjunctivity and frame circumscription\(^2\), is there an analogy?

**A Fuller Integration of Bunches into the B Method and Tool Support**

In Chapter 5 we have developed a theory of bunches which we claim potentially lends itself for integration into the B Method and tools. Ultimately, however, we haven’t performed such an integration. In our current work this is not anticipated since we require all bunch expressions to occur only in packed form as the contents of a set. A more ambitious goal would be to support bunches as mathematical objects within a B machine specification, or directly within the target execution platform by allowing statements of the form \(x := S \diamond E\) alongside the presently supported \(x := \{S \diamond E\}\).

One of the implications of this seems that we require a more liberalised definition of assignment, and moreover might have to expect the possibility of non-termination and infeasibility of the assignment statement. The way we interpret such an assignment could be either to incorporate genuine bunch variables into our target language, or instead regard it as the daemonic choice of values delivered by the bunch expression \(\{S \diamond E\}\). One of the crucial difference between wp and pv semantics is that prospective values not just constitute of a formalism for reasoning about computations, but \(S \diamond E\) itself is an executable statement in our target language, thus an implementable expression (in the context of a set). Examining how this affect formal development of software and reasoning about correctness is a further worthy area of subsequent research that could be embarked upon.

In particular it would be desirable to conduct a case study which made more extensive use of pv transformers within a substantial B development, for example generalising the Knight’s Tour to incorporate heuristics. This could serve as a fruitful future exercise from which we expect much valuable experience and knowledge could be derived.

**Integrating PV Semantics and Probability**

In recent years methods for reasoning about probabilistic systems and programs have gained a considerable interest, and the issues arising from interaction between non-deterministic and probabilistic choice seem in particular challenging for the design of new theories and methods which entail the specification and verification of probabilistic systems and algorithms. Our aim for future investigation here would be to see how the current methods of reasoning about probabilistic programs fit in with the kind of backtracking emerging from guards and choice.

\(^2\)This healthiness condition is part of Dunne’s theory of generalised substitutions [Dun02].
The questions arising here in particular are whether we can formally define a probabilistic choice which moreover backtracks when encountering infeasibility (such would be very useful for example in the implementation of heuristic methods), and what properties such a construct would have at the semantics level.

Our RVM does already support the executing or probabilistic programs, however it does this in a very specific way i.e. restoring the random seed upon reversing used in determining subsequent probabilistic choice by means of some deterministic pseudo-random process. Investigating in what way probability could sensibly be integrated into the semantics of prospective values, and whether in such we are able to reason directly about the implementation of probabilistic behaviour in our RVM will be another major research project which might also potentially offer some new insights into the development and verification of probabilistic applications of a particular kind, e.g. using probabilistic behaviour to implement heuristic methods for optimising backtracking search.
Appendix A

Symbol and Notation Glossary

A.1 Predicate Logic

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>∧</td>
<td>Logical And</td>
<td>infix operator</td>
</tr>
<tr>
<td>∨</td>
<td>Logical Or</td>
<td>infix operator</td>
</tr>
<tr>
<td>¬</td>
<td>Logical Not</td>
<td>prefix operator</td>
</tr>
<tr>
<td>⇒</td>
<td>Logical Implication</td>
<td>infix operator</td>
</tr>
<tr>
<td>⇔</td>
<td>Logical Equivalence</td>
<td>infix operator</td>
</tr>
<tr>
<td>∀</td>
<td>Universal Quantification</td>
<td>quantifier</td>
</tr>
<tr>
<td>∃</td>
<td>Existential Quantification</td>
<td>quantifier</td>
</tr>
<tr>
<td>true</td>
<td>True Predicate</td>
<td>constant</td>
</tr>
<tr>
<td>false</td>
<td>False Predicate</td>
<td>constant</td>
</tr>
</tbody>
</table>

A.1.1 Predicate Logic (B-AMN)

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>BOOL</td>
<td>Type of Booleans (B-AMN)</td>
<td>constant</td>
</tr>
<tr>
<td>TRUE</td>
<td>Boolean value for True (member of BOOL)</td>
<td>constant</td>
</tr>
<tr>
<td>FALSE</td>
<td>Boolean value for False (member of BOOL)</td>
<td>constant</td>
</tr>
<tr>
<td>bool</td>
<td>Maps predicates to elements of BOOL</td>
<td>prefix operator</td>
</tr>
</tbody>
</table>
A.2 Set Theory

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>{e_1, e_2, \ldots}</td>
<td>Set Enumeration</td>
<td></td>
</tr>
<tr>
<td>\cup</td>
<td>Set Union</td>
<td>infix operator</td>
</tr>
<tr>
<td>\cap</td>
<td>Set Intersection</td>
<td>infix operator</td>
</tr>
<tr>
<td>\setminus</td>
<td>Set Subtraction</td>
<td>infix operator</td>
</tr>
<tr>
<td>\in</td>
<td>Set Membership</td>
<td>infix operator</td>
</tr>
<tr>
<td>\notin</td>
<td>Not in Set (negation of \in)</td>
<td>infix operator</td>
</tr>
<tr>
<td>\subseteq</td>
<td>Subset Inclusion</td>
<td>infix operator</td>
</tr>
<tr>
<td>\text{card}</td>
<td>Cardinality of a Set</td>
<td>prefix operator</td>
</tr>
<tr>
<td>\mapsto</td>
<td>Maplet Construction</td>
<td>infix operator</td>
</tr>
<tr>
<td>\mathbb{P}</td>
<td>Power Set</td>
<td>prefix operator</td>
</tr>
<tr>
<td>\times</td>
<td>Cartesian Product</td>
<td>infix operator</td>
</tr>
<tr>
<td>n..m</td>
<td>Interval of Naturals from n to m</td>
<td>infix operator</td>
</tr>
</tbody>
</table>

A.3 Functions and Relations

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>\lambda</td>
<td>Functional Abstraction</td>
<td>quantifier</td>
</tr>
<tr>
<td>f(\ldots)</td>
<td>Function Application</td>
<td>juxtaposition</td>
</tr>
<tr>
<td>f(\mid \ldots \mid)</td>
<td>Relational Image</td>
<td></td>
</tr>
<tr>
<td>\leftrightarrow</td>
<td>Set of Relations</td>
<td>infix operator</td>
</tr>
<tr>
<td>\rightarrow</td>
<td>Set of Functions</td>
<td>infix operator</td>
</tr>
<tr>
<td>\rightarrow</td>
<td>Set of Partial Functions</td>
<td>infix operator</td>
</tr>
<tr>
<td>\Rightarrow</td>
<td>Set of Partial Injections</td>
<td>infix operator</td>
</tr>
<tr>
<td>\text{dom}</td>
<td>Domain of a Function or Relation</td>
<td>prefix operator</td>
</tr>
<tr>
<td>\text{ran}</td>
<td>Range of a Function or Relation</td>
<td>prefix operator</td>
</tr>
<tr>
<td>\circ</td>
<td>Relational Composition</td>
<td>infix operator</td>
</tr>
</tbody>
</table>
## A.4 Sequences

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>iseq $T$</td>
<td>Injective Sequences over elements from $T$</td>
<td>prefix operator</td>
</tr>
<tr>
<td>size</td>
<td>Length of a Sequence</td>
<td>prefix operator</td>
</tr>
<tr>
<td>last</td>
<td>Last element of a Sequence</td>
<td>prefix operator</td>
</tr>
<tr>
<td>$[x]$</td>
<td>Sequence consisting of just one element $x$</td>
<td></td>
</tr>
</tbody>
</table>

## A.5 Bunch Theory

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\cdot$</td>
<td>Bunch Union</td>
<td>infix operator</td>
</tr>
<tr>
<td>$\colon$</td>
<td>Bunch Intersection</td>
<td>infix operator</td>
</tr>
<tr>
<td>$\backslash$</td>
<td>Bunch Difference</td>
<td>infix operator</td>
</tr>
<tr>
<td>$:$</td>
<td>Subbunch Inclusion</td>
<td>infix operator</td>
</tr>
<tr>
<td>$\equiv, =)$</td>
<td>Bunch Equality</td>
<td>infix operator</td>
</tr>
<tr>
<td>$\neq$</td>
<td>Inequality (negation of $=$)</td>
<td>infix operator</td>
</tr>
<tr>
<td>$\notin$</td>
<td>Bunch Cardinality</td>
<td>prefix operator</td>
</tr>
<tr>
<td>$\Delta$</td>
<td>Elementary Property</td>
<td>prefix operator</td>
</tr>
<tr>
<td>$\mathbf{null}_T$</td>
<td>Empty Bunch of type $T$</td>
<td>constant</td>
</tr>
<tr>
<td>$\mathbf{⊥}_T$</td>
<td>Improper Bunch of type $T$</td>
<td>constant</td>
</tr>
<tr>
<td>${\ldots}$</td>
<td>Packaging (of a bunch)</td>
<td></td>
</tr>
<tr>
<td>$\sim$</td>
<td>Unpackaging (of a set)</td>
<td>prefix operator</td>
</tr>
<tr>
<td>$\rightarrow$</td>
<td>Guarded Bunch</td>
<td>infix operator</td>
</tr>
<tr>
<td>$I$</td>
<td>Preconditioned Bunch</td>
<td>infix operator</td>
</tr>
<tr>
<td>$\S$</td>
<td>Bunch Comprehension</td>
<td>quantifier</td>
</tr>
<tr>
<td>$o^*$</td>
<td>Alternative lifting of operator $o$ via existential quantification</td>
<td>superscript</td>
</tr>
<tr>
<td>$\subseteq$</td>
<td>Refinement (of expressions)</td>
<td>infix operator</td>
</tr>
<tr>
<td>$T_b$</td>
<td>Bunch of all elements of type $T$</td>
<td>subscript</td>
</tr>
</tbody>
</table>
A.6 Generalised Substitutions Language

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>skip</td>
<td>Skip</td>
<td>constant</td>
</tr>
<tr>
<td>:=</td>
<td>Assignment</td>
<td>infix operator</td>
</tr>
<tr>
<td></td>
<td>Precondition</td>
<td>infix operator</td>
</tr>
<tr>
<td>==&gt;</td>
<td>Guard</td>
<td>infix operator</td>
</tr>
<tr>
<td>[]</td>
<td>Choice</td>
<td>infix operator</td>
</tr>
<tr>
<td>;</td>
<td>Sequential Composition</td>
<td>infix operator</td>
</tr>
<tr>
<td>@</td>
<td>Unbounded Choice</td>
<td>quantifier</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>S^</td>
<td>Transitive opening of S</td>
<td>superscript</td>
</tr>
<tr>
<td>Sy</td>
<td>Frame Extension</td>
<td>subscript</td>
</tr>
<tr>
<td>abort</td>
<td>Abort (≡ false</td>
<td>skip)</td>
</tr>
<tr>
<td>magic</td>
<td>Magic (≡ false ==&gt; skip)</td>
<td>constant</td>
</tr>
</tbody>
</table>

A.7 Program Semantics

<table>
<thead>
<tr>
<th>Notation</th>
<th>Description</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>[S]Q</td>
<td>Weakest precondition (Abrial) for S to establish Q</td>
<td></td>
</tr>
<tr>
<td>wp(S, Q)</td>
<td>Weakest precondition for S to establish Q</td>
<td></td>
</tr>
<tr>
<td>wp(S, Q)</td>
<td>Conjugate weakest precondition (≡ ¬wp(S, ¬Q))</td>
<td></td>
</tr>
<tr>
<td>w : [Q]</td>
<td>Morgan Specification Statement</td>
<td></td>
</tr>
<tr>
<td>S ⊙ E</td>
<td>Prospective value of E after running S</td>
<td>infix operator</td>
</tr>
<tr>
<td>⊑</td>
<td>Refinement (of GSL constructs)</td>
<td>infix operator</td>
</tr>
<tr>
<td>⊑*</td>
<td>Feasibility-preserving (Star) Refinement</td>
<td>infix operator</td>
</tr>
<tr>
<td>frame(S)</td>
<td>Frame of a generalised substitutions S</td>
<td></td>
</tr>
<tr>
<td>trm(S)</td>
<td>Char. predicate describing termination of S</td>
<td></td>
</tr>
<tr>
<td>fis(S)</td>
<td>Char. predicate describing feasibility of S</td>
<td></td>
</tr>
<tr>
<td>prd(S)</td>
<td>Char. predicate describing the behaviour of S</td>
<td></td>
</tr>
</tbody>
</table>
### A.8 Arithmetics

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \mathbb{N} )</td>
<td>Natural Numbers</td>
<td>constant</td>
</tr>
<tr>
<td>( \mathbb{N}_1 )</td>
<td>Natural Numbers excluding 0</td>
<td>constant</td>
</tr>
<tr>
<td>+</td>
<td>Arithmetic Addition</td>
<td>infix operator</td>
</tr>
<tr>
<td>−</td>
<td>Arithmetic Subtraction</td>
<td>infix operator</td>
</tr>
<tr>
<td>*</td>
<td>Arithmetic Multiplication</td>
<td>infix operator</td>
</tr>
<tr>
<td>/</td>
<td>Arithmetic Division</td>
<td>infix operator</td>
</tr>
<tr>
<td>( \text{mod} )</td>
<td>Integer Modulus</td>
<td>infix operator</td>
</tr>
<tr>
<td>&lt;, &gt;, ( \leq ), ( \geq )</td>
<td>Relational Operators on Numbers</td>
<td>infix operators</td>
</tr>
<tr>
<td>( \ln )</td>
<td>Natural Logarithm</td>
<td>function</td>
</tr>
<tr>
<td>\text{min}</td>
<td>Min element of a set of numbers</td>
<td>prefix operator</td>
</tr>
<tr>
<td>\text{max}</td>
<td>Max element of a set of numbers</td>
<td>prefix operator</td>
</tr>
</tbody>
</table>

### A.9 Miscellaneous

<table>
<thead>
<tr>
<th>Notation</th>
<th>Description</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>=_{df}</td>
<td>Introducing a global definition</td>
<td></td>
</tr>
<tr>
<td>≡</td>
<td>Introducing a local definition</td>
<td></td>
</tr>
<tr>
<td>( \emptyset )</td>
<td>Empty List (of variables or expressions)</td>
<td>constant</td>
</tr>
<tr>
<td>( E[x/F] )</td>
<td>Syntactic (( \beta )) Substitution (substitute ( F ) for ( x ) in ( E ))</td>
<td></td>
</tr>
<tr>
<td>if ( P ) then ( E_1 ) else ( E_2 ) end</td>
<td>Conditional Expression (evaluates to ( E_1 ) if ( P ) and ( E_2 ) if ( \neg P ))</td>
<td></td>
</tr>
<tr>
<td>( \mu Y \bullet f(Y) )</td>
<td>Weakest fix point of ( f )</td>
<td>quantifier</td>
</tr>
<tr>
<td>( \nu Y \bullet f(Y) )</td>
<td>Strongest fix point of ( f )</td>
<td>quantifier</td>
</tr>
<tr>
<td>( x : \in A )</td>
<td>Choice form a Set</td>
<td>infix operator</td>
</tr>
<tr>
<td>if \ldots fi</td>
<td>Dijkstra’s conditional statement</td>
<td></td>
</tr>
<tr>
<td>( x \setminus E )</td>
<td>Non-freeness of variable ( x ) in ( E )</td>
<td></td>
</tr>
</tbody>
</table>
Appendix B

Hehner’s Bunch Theory Axioms and Laws

The following bunch theory axioms taken from the 2007 online edition of Hehner’s book “A practical Theory of programming” [Heh07] are also fundamental in Improper Bunch Theory and thus included as an Appendix of the thesis.

B.1 Hehner’s Bunch Theory Axioms

<table>
<thead>
<tr>
<th>Axiom</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x : y = x = y$</td>
<td>Elementary Axiom</td>
</tr>
<tr>
<td>$x : A, B = x : A \lor x : B$</td>
<td>Compound Axiom</td>
</tr>
<tr>
<td>$A, A = A$</td>
<td>Idempotence</td>
</tr>
<tr>
<td>$A, B = B, A$</td>
<td>Symmetry</td>
</tr>
<tr>
<td>$A, (B, C) = (A, B), C$</td>
<td>Associativity</td>
</tr>
<tr>
<td>$A : A = A$</td>
<td>Idempotence</td>
</tr>
<tr>
<td>$A : B = B : A$</td>
<td>Symmetry</td>
</tr>
<tr>
<td>$A : (B : C) = (A : B) : C$</td>
<td>Associativity</td>
</tr>
<tr>
<td>$A, B : C = A : C \land B : C$</td>
<td>Antidistributivity</td>
</tr>
<tr>
<td>$A : B : C = A : B \land A : C$</td>
<td>Distributivity</td>
</tr>
<tr>
<td>$A : A, B$</td>
<td>Generalisation</td>
</tr>
<tr>
<td>$A : B : A$</td>
<td>Specialisation</td>
</tr>
</tbody>
</table>
### APPENDIX B. HEHNER’S BUNCH THEORY AXIOMS AND LAWS

<table>
<thead>
<tr>
<th>Law</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A : A$</td>
<td>Reflexivity</td>
</tr>
<tr>
<td>$A : B \land B : A \Rightarrow A = B$</td>
<td>Antisymmetry</td>
</tr>
<tr>
<td>$A : B \land B : C \Rightarrow A : C$</td>
<td>Transitivity</td>
</tr>
<tr>
<td>$\not x = 1$</td>
<td>Size</td>
</tr>
<tr>
<td>$\not (A, B) + \not (A \cdot B) = \not A + \not B$</td>
<td>Size</td>
</tr>
<tr>
<td>$\not x : A \Rightarrow \not (A \cdot x) = 0$</td>
<td>Size</td>
</tr>
<tr>
<td>$A : B \Rightarrow \not A \leq \not B$</td>
<td>Size</td>
</tr>
<tr>
<td>null : A</td>
<td>Induction</td>
</tr>
<tr>
<td>$\not A = 0 = A = \text{null}$</td>
<td>Size</td>
</tr>
</tbody>
</table>

#### B.2 Laws following from these Axioms

<table>
<thead>
<tr>
<th>Law</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A, (A \cdot B) = A$</td>
<td>Absorption</td>
</tr>
<tr>
<td>$A \cdot (A, B) = A$</td>
<td>Absorption</td>
</tr>
<tr>
<td>$A : B \Rightarrow C, A : C, B$</td>
<td>Monotonicity</td>
</tr>
<tr>
<td>$A : B \Rightarrow C \cdot A : C \cdot B$</td>
<td>Monotonicity</td>
</tr>
<tr>
<td>$A : B = A, B = B = A = A \cdot B$</td>
<td>Inclusion</td>
</tr>
<tr>
<td>$A, (B, C) = (A, B), (A, C)$</td>
<td>Distributivity</td>
</tr>
<tr>
<td>$A, (B \cdot C) = (A, B) \cdot (A, C)$</td>
<td>Distributivity</td>
</tr>
<tr>
<td>$A \cdot (B, C) = (A \cdot B), (A \cdot C)$</td>
<td>Distributivity</td>
</tr>
<tr>
<td>$A \cdot (B \cdot C) = (A \cdot B) \cdot (A \cdot C)$</td>
<td>Distributivity</td>
</tr>
<tr>
<td>$A : B \land C : D \Rightarrow A, C : B, D$</td>
<td>Conflation</td>
</tr>
<tr>
<td>$A : B \land C : D \Rightarrow A \cdot C : B \cdot D$</td>
<td>Conflation</td>
</tr>
<tr>
<td>$A, \text{null} = A$</td>
<td>Identity</td>
</tr>
<tr>
<td>$A \cdot \text{null} = \text{null}$</td>
<td>Base</td>
</tr>
<tr>
<td>$\not \text{null} = 0$</td>
<td>Size</td>
</tr>
</tbody>
</table>
Appendix C

Alternative Proof of the Seq. Comp. Law

The following alternative proof of the sequential composition pv rewrite law (Prop. 40) exploits Theorem 49 given in Section 6.6.

Proof. \( S \circ T \circ E \)

\[ \begin{align*}
&= \text{“Alt. wp characterisation of } S \circ E \text{ (Theorem 49)"} \\
&= \text{“Alt. wp characterisation of } S \circ E \text{ (Theorem 49)"} \\
&= \text{“Bunch law: } P \mid E \neq \bot = P \text{ providing } E \neq \bot, \text{ note that } (\mathcal{L} x \bullet P \rightarrow x) \text{ is always } \neq \bot” \\
&= \text{“Bunch law: } x : P \mid E = P \Rightarrow x : E” \\
&= \text{“Logic (de Morgan)”} \\
&= \text{“Conjunctivity of wp”}
\end{align*} \]
APPENDIX C. ALTERNATIVE PROOF OF THE SEQ. COMP. LAW

\[\text{wp}(S; T, E \neq \bot) \mid \exists x \bullet \\
\neg(\text{wp}(S, \text{wp}(T, E \neq \bot)) \land \text{wp}(S, \neg x : \exists x \bullet \neg \text{wp}(T, \neg x : E) \rightarrow x)) \rightarrow x\]

\begin{align*}
\text{=} & \quad \text{Logic (de Morgan)} \\
\text{wp}(S; T, E \neq \bot) \mid \exists x \bullet \\
\neg\text{wp}(S, \text{wp}(T, E \neq \bot)) \lor \neg\text{wp}(S, \neg x : \exists x \bullet \neg \text{wp}(T, \neg x : E) \rightarrow x) \rightarrow x
\end{align*}

\begin{align*}
\text{=} & \quad \text{Sequential composition rule for wp} \\
\text{wp}(S; T, E \neq \bot) \mid \exists x \bullet \\
\neg\text{wp}(S; T, E \neq \bot) \lor \neg\text{wp}(S, \neg x : \exists x \bullet \neg \text{wp}(T, \neg x : E) \rightarrow x) \rightarrow x
\end{align*}

\begin{align*}
\text{=} & \quad \text{Splitting bunch comprehension (Prop. 28)} \\
\text{wp}(S; T, E \neq \bot) \mid (\exists x \bullet \neg \text{wp}(S; T, E \neq \bot) \rightarrow x), \\
\exists x \bullet \neg \text{wp}(S, \neg x : \exists x \bullet \neg \text{wp}(T, \neg x : E) \rightarrow x) \rightarrow x
\end{align*}

\begin{align*}
\text{=} & \quad \text{Bunch comprehension slide rule (Prop. 26), note that } x \setminus \text{wp}(S; T, E \neq \bot) \\
\text{wp}(S; T, E \neq \bot) \mid \exists x \bullet \\
\neg\text{wp}(S; T, E \neq \bot) \leftrightarrow \neg\text{wp}(S, \neg x : \exists x \bullet \neg \text{wp}(T, \neg x : E) \rightarrow x) \rightarrow x
\end{align*}

\begin{align*}
\text{=} & \quad \text{Bunch identity: } P \mid \neg P \rightarrow E, F \equiv P \mid F \\
\text{wp}(S; T, E \neq \bot) \mid \exists x \bullet \neg\text{wp}(S, \neg x : \exists x \bullet \neg \text{wp}(T, \neg x : E) \rightarrow x) \rightarrow x
\end{align*}

\begin{align*}
\text{=} & \quad \text{Bunch law: } x : \exists x \bullet P \rightarrow x \equiv P \\
\text{wp}(S; T, E \neq \bot) \mid \exists x \bullet \neg\text{wp}(S, \neg\text{wp}(T, \neg x : E)) \rightarrow x
\end{align*}

\begin{align*}
\text{=} & \quad \text{Logic} \\
\text{wp}(S; T, E \neq \bot) \mid \exists x \bullet \neg\text{wp}(S, \text{wp}(T, \neg x : E)) \rightarrow x
\end{align*}

\begin{align*}
\text{=} & \quad \text{Sequential composition rule for wp} \\
\text{wp}(S; T, E \neq \bot) \mid \exists x \bullet \neg\text{wp}(S; T, \neg x : E) \rightarrow x
\end{align*}

\begin{align*}
\text{=} & \quad \text{Alt. wp characterisation of } S \diamond E \text{ (Theorem 49)} \\
S; T \diamond E \quad \square.
\end{align*}
Appendix D

B Development of the Knight’s Tour

D.1 Abstract Specification: KnightsTourSolver

MACHINE KnightsTourSolver

ABSTRACT_CONSTANTS
  valid_move, valid_tour, valid_solution

CONCRETE_CONSTANTS
  SQUARE

PROPERTIES
  SQUARE = 0 .. 63 ∧
  valid_move ⊆ SQUARE × SQUARE ∧
  ∀ (ss, tt).(ss ∈ SQUARE ∧ tt ∈ SQUARE ⇒
             (ss →→ tt ∈ valid_move
             ⇔
             /* There are 8 potential ways of moving a knight on a chessboard. */
             ((row(ss) ≥ 2 ∧ col(ss) ≥ 1 ∧ tt = ss - 17) ∨
              (row(ss) ≥ 2 ∧ col(ss) ≤ 6 ∧ tt = ss - 15) ∨
              (row(ss) ≥ 1 ∧ col(ss) ≥ 2 ∧ tt = ss - 10) ∨
              (row(ss) ≥ 1 ∧ col(ss) ≤ 5 ∧ tt = ss - 6) ∨
              (row(ss) ≤ 6 ∧ col(ss) ≥ 2 ∧ tt = ss + 6) ∨
              (row(ss) ≤ 6 ∧ col(ss) ≤ 5 ∧ tt = ss + 10) ∨
              (row(ss) ≤ 5 ∧ col(ss) ≥ 1 ∧ tt = ss + 15) ∨

  

160
\[(\text{row}(ss) \leq 5 \land \text{col}(ss) \leq 6 \land tt = ss + 17))) \land\]

\[\text{valid}_\text{tour} \subseteq \text{iseq}(\text{SQUARE}) \land\]

\[\forall tt. (tt \in \text{iseq}(\text{SQUARE}) \Rightarrow (tt \in \text{valid}_\text{tour} \iff\]

\[\forall ii. (ii \in 1 \ldots \text{size}(tt)-1 \Rightarrow tl(ii) \rightarrow tl(ii+1) \in \text{valid}_\text{move})) \land\]

\[\text{valid}_\text{solution} \subseteq \text{valid}_\text{tour} \land\]

\[\forall tt. (tt \in \text{valid}_\text{tour} \Rightarrow (tt \in \text{valid}_\text{solution} \iff (\text{ran}(tt) = \text{SQUARE})))\]

\textbf{DEFINITIONS}
\[
\begin{align*}
\text{row}(ss) &= ss / 8; \\
\text{col}(ss) &= ss \mod 8
\end{align*}
\]

\textbf{OPERATIONS}
\[
\begin{align*}
tour &\leftarrow \text{solve}(\text{start}) = \\
\text{PRE} &
\begin{align*}
\text{start} &\in \text{SQUARE}
\end{align*}
\text{THEN} &
\begin{align*}
\text{ANY} \ tt \ WHERE &
\begin{align*}
\text{tt} &\in \text{valid}_\text{solution} \land \\
\text{tt}(1) &\text{=} \text{start}
\end{align*}
\text{THEN} &
\begin{align*}
tour &:= tt
\end{align*}
\text{END} &
\text{END}
\end{align*}
\text{END}
\]
D.2 Implementation: KnightsTourSolverI

IMPLEMENTATION KnightsTourSolverI

REFINES KnightsTourSolver

IMPORTS KnightsTourSolverUtil

VALUES

\[ SQUARE = 0 \ldots 63 \]

OPERATIONS

\[
\text{tour} \leftarrow \text{solve}(\text{start}) =
\]

VAR \( ss, \text{length} \) IN

\[
\text{tour} := [\text{start}];
\]

\[
\text{length} := 1;
\]

\[
\text{WHILE} \ \text{length} < 64 \ \text{DO}
\]

\[
ss \leftarrow \text{select}_\text{move} (\text{last}(\text{tour}));
\]

\[
\text{check}_\text{unvisited}(ss, \text{tour});
\]

\[
\text{tour} := \text{tour} \leftarrow ss;
\]

\[
\text{length} := \text{length} + 1
\]

INVARIANT

\[
\text{tour}(1) = \text{start} \land \\
\text{tour} \in \text{iseq}(SQUARE) \land \\
\text{tour} \in \text{valid}_\text{tour} \land \\
\text{length} \in 1 \ldots 64 \land \\
\text{length} = \text{size}(\text{tour})
\]

VARIANT

\[
64 - \text{length}
\]

END

END

END
## D.3 Abstract Specification: KnightsTourSolverUtil

**MACHINE** KnightsTourSolverUtil

**SEES** KnightsTourSolver

**DEFINITIONS**

\[
\begin{align*}
row(ss) & = ss / 8; \\
col(ss) & = ss \mod 8
\end{align*}
\]

**OPERATIONS**

\[
rr \leftarrow \text{select}_\text{move}(ss) =
\]

**PRE**

\[
ss \in SQUARE
\]

**THEN**

\[
\begin{align*}
\text{SELECT} \ & \ row(ss) \geq 2 \land col(ss) \geq 1 \text{ THEN} \\
rr & := ss - 17 \\
\text{WHEN} \ & \ row(ss) \geq 2 \land col(ss) \leq 6 \text{ THEN} \\
rr & := ss - 15 \\
\text{WHEN} \ & \ row(ss) \geq 1 \land col(ss) \geq 2 \text{ THEN} \\
rr & := ss - 10 \\
\text{WHEN} \ & \ row(ss) \geq 1 \land col(ss) \leq 5 \text{ THEN} \\
rr & := ss - 6 \\
\text{WHEN} \ & \ row(ss) \leq 6 \land col(ss) \geq 2 \text{ THEN} \\
rr & := ss + 6 \\
\text{WHEN} \ & \ row(ss) \leq 6 \land col(ss) \leq 5 \text{ THEN} \\
rr & := ss + 10 \\
\text{WHEN} \ & \ row(ss) \leq 5 \land col(ss) \geq 1 \text{ THEN} \\
rr & := ss + 15 \\
\text{WHEN} \ & \ row(ss) \leq 5 \land col(ss) \leq 6 \text{ THEN} \\
rr & := ss + 17
\end{align*}
\]

**END**

**END**

**check_unvisited**(ss, tt) =

**PRE**

\[
\begin{align*}
ss & \in SQUARE \land \\
tt & \in \text{iseq}(SQUARE)
\end{align*}
\]

**THEN**

\[
\begin{align*}
\text{SELECT} \ & \ ss \not\in \text{ran}(tt) \text{ THEN} \text{skip END}
\end{align*}
\]

**END**

**END**
D.4 Implementation: KnightsTourSolverUtilI

IMPLEMENTATION KnightsTourSolverUtilI

REFINES KnightsTourSolverUtilI

SEES KnightsTourSolver

DEFINITIONS
ROW(\textit{ss}) == \textit{ss} / 8;
COL(\textit{ss}) == \textit{ss} mod 8

OPERATIONS
\texttt{rr} \leftarrow \texttt{select\_move} (\textit{ss}) =
BEGIN
\texttt{SELECT \textit{ROW}(\textit{ss}) \geq 2 \land \textit{COL}(\textit{ss}) \geq 1 \ \texttt{THEN}}
\hspace{1em} \texttt{rr := ss - 17}
\texttt{WHEN \textit{ROW}(\textit{ss}) \geq 2 \land \textit{COL}(\textit{ss}) \leq 6 \ \texttt{THEN}}
\hspace{1em} \texttt{rr := ss - 15}
\texttt{WHEN \textit{ROW}(\textit{ss}) \geq 1 \land \textit{COL}(\textit{ss}) \geq 2 \ \texttt{THEN}}
\hspace{1em} \texttt{rr := ss - 10}
\texttt{WHEN \textit{ROW}(\textit{ss}) \geq 1 \land \textit{COL}(\textit{ss}) \leq 5 \ \texttt{THEN}}
\hspace{1em} \texttt{rr := ss - 6}
\texttt{WHEN \textit{ROW}(\textit{ss}) \leq 6 \land \textit{COL}(\textit{ss}) \geq 2 \ \texttt{THEN}}
\hspace{1em} \texttt{rr := ss + 6}
\texttt{WHEN \textit{ROW}(\textit{ss}) \leq 6 \land \textit{COL}(\textit{ss}) \leq 5 \ \texttt{THEN}}
\hspace{1em} \texttt{rr := ss + 10}
\texttt{WHEN \textit{ROW}(\textit{ss}) \leq 5 \land \textit{COL}(\textit{ss}) \geq 1 \ \texttt{THEN}}
\hspace{1em} \texttt{rr := ss + 15}
\texttt{WHEN \textit{ROW}(\textit{ss}) \leq 5 \land \textit{COL}(\textit{ss}) \leq 6 \ \texttt{THEN}}
\hspace{1em} \texttt{rr := ss + 17}
\texttt{END}
\texttt{END;

\texttt{check\_unvisited(\textit{ss}, \texttt{tt}) =}
\texttt{BEGIN}
\hspace{1em} \texttt{SELECT \textit{ss} \not\in \texttt{ran(tt)} \ \texttt{THEN skip \ END}}
\texttt{END}
\texttt{END}
D.5 Abstract Specification: KnightsTourSolverGeneric

/* It would be more elegant to parameterise the machine instead of introducing
the concrete constants ROWS and COLS to determine the board size. Such
however results in a visibility problem accessing the machine parameters in the
PROPERTIES clause. */

MACHINE KnightsTourSolverGeneric

ABSTRACT_CONSTANTS
   valid_move, valid_tour, valid_solution

CONCRETE_CONSTANTS
   SQUARE, ROWS, COLS

PROPERTIES
   ROWS ∈ NAT1 ∧
   COLS ∈ NAT1 ∧
   SQUARE = 0 .. (COLS × ROWS)-1 ∧
   valid_move ⊆ SQUARE × SQUARE ∧
   ∀ (ss, tt).(ss ∈ SQUARE ∧ tt ∈ SQUARE ⇒
   (ss ↦→ tt ∈ valid_move

   /* There are 8 potential ways of moving a knight on a chessboard. */
   ((row(ss) ≥ 2 ∧ col(ss) ≥ 1 ∧ tt = ss - 17) ∨
   (row(ss) ≥ 2 ∧ col(ss) ≤ LAST_COL - 1 ∧ tt = ss - 15) ∨
   (row(ss) ≥ 1 ∧ col(ss) ≥ 2 ∧ tt = ss - 10) ∨
   (row(ss) ≥ 1 ∧ col(ss) ≤ LAST_COL - 2 ∧ tt = ss - 6) ∨
   (row(ss) ≤ LAST_ROW - 1 ∧ col(ss) ≥ 2 ∧ tt = ss + 6) ∨
   (row(ss) ≤ LAST_ROW - 1 ∧ col(ss) ≤ LAST_COL - 2 ∧ tt = ss + 10) ∨
   (row(ss) ≤ LAST_ROW - 2 ∧ col(ss) ≥ 1 ∧ tt = ss + 15) ∨
   (row(ss) ≤ LAST_ROW - 2 ∧ col(ss) ≤ LAST_COL - 1 ∧ tt = ss + 17))) ∧

   valid_tour ⊆ iseq(SQUARE) ∧
   ∀ tt.(tt ∈ iseq(SQUARE) ⇒
   (tt ∈ valid_tour ↔ ∀ ii.(ii ∈ 1 .. size(tt)-1 ⇒ tt(ii) ↦→ tt(ii+1) ∈ valid_move)))
\[
valid\_solution \subseteq valid\_tour \land \\
\forall tt.(tt \in valid\_tour \Rightarrow (tt \in valid\_solution \iff (\text{ran}(tt) = SQUARE)))
\]

**DEFINITIONS**

\[
\begin{align*}
\text{row}(ss) &= ss / 8; \\
\text{col}(ss) &= ss \mod 8; \\
\text{LAST\_ROW} &= ROWS - 1; \\
\text{LAST\_COL} &= COLS - 1
\end{align*}
\]

**OPERATIONS**

\[
tour \leftarrow \text{solve}(start) = \\
\text{PRE} \\
\text{start} \in SQUARE \\
\text{THEN} \\
\text{ANY} tt \text{ WHERE} \\
\text{tt} \in \text{valid\_solution} \land \\
\text{tt}(1) = start \\
\text{THEN} \\
\text{tour} := tt \\
\text{END} \\
\text{END}
\]
Appendix E

Correctness Proof of the Knight’s Tour

E.1 User Theory for Discharging POs in KnightsTourSolverI

THEORY Sequences IS
  a : s
  =>
  [a]^ : s +-> NATURAL;

  binhyp(q : seq(s)) &
  q : iseq(s) &
  ran(q) <: 0..n-1
  =>
  size(q) <= n ;

  [a](1) == a ;

  binhyp(q : seq(s)) &
  size(q) >= 1
  =>
  (q ^ t)(1) == q(1) ;

  binhyp(q : seq(s)) &
  binhyp(q(1) = e)
  =>
  size(q) >= 1 ;

  binhyp(q : seq(s)) &
APPENDIX E. CORRECTNESS PROOF OF THE KNIGHT’S TOUR

\[ \text{binhyp}(q : s \rightarrow \text{NATURAL}) \quad /\!* \text{q injective} \quad */ \]
\[ \text{binhyp}(\text{ran}(q) \subseteq a) \]
\[ \Rightarrow \]
\[ \text{size}(q) \subseteq \text{card}(a) ; \]

\[ q : \text{iseq}(s) \quad /\]
\[ a /: \text{ran}(q) \]
\[ \Rightarrow \]
\[ (q \upharpoonright [a]) : s \rightarrow \text{NATURAL} ; \quad /\!* q \upharpoonright [a] \text{ injective} \quad */ \]

\[ \text{binhyp}(q : \text{seq}(s)) \]
\[ \Rightarrow \]
\[ \text{last}(q \upharpoonright [a]) = a \]

END

&

THEORY Arithmetic IS

\[ \text{binhyp}(n \leq x / k) \quad /\]
\[ \text{binhyp}(m \leq x \mod k) \]
\[ \Rightarrow \]
\[ x \geq (n \times k) + m ; \]

\[ \text{binhyp}(n \leq x / k) \]
\[ \Rightarrow \]
\[ x \geq (n \times k) ; \]

\[ \text{binhyp}(x / k \leq n) \quad /\]
\[ \text{binhyp}(x \mod k \leq m) \]
\[ \Rightarrow \]
\[ x \leq n \times k + m ; \]

\[ \text{binhyp}(x / k \leq n) \]
\[ \Rightarrow \]
\[ x \leq (n+1) \times k - 1 ; \]

\[ \text{binhyp}(m \leq n) \quad /\]
\[ \text{not}(m \leq n-1) \]
\[ \Rightarrow \]
\[ m = n \]

END

&
THEORY Logic IS
(Q => R) =>
((P & Q) => R) ;

P =>
((P => Q) => Q)
END

&

THEORY Lemma IS
binhyp(q : seq(chessboard)) &
a : chessboard &
is_valid_tour_prefix(q) = TRUE &
is_valid_move(last(q), a) = TRUE &
a /: ran(q)
=>
is_valid_tour_prefix(q ^ [a]) = TRUE
END
E.2 Example of an Interactive Proof using Click’n Proof

Proof obligation PO69 in KnightsTourSolverI (solve operation):

\[ \text{tour}^1 \leftarrow \text{last(tour}^1) - 17 \in \text{valid_tour} \]

Corresponding proof tree generated with Click’n Prove V2.0.

```
[mk] cl
[zm] /*tour$1<-last(tour$1)-17 : valid_tour*/
[mk] xp
[mk] ah(last(tour$1) : SQUARE)
[zm] /*last(tour$1) : SQUARE*/
[mk] pr
[mk] ah(last(tour$1)-17 : SQUARE)
[zm] /*last(tour$1)-17 : SQUARE*/
[mk] cls
[mk] ah(1<=last(tour$1) mod 8)
[mk] ah(2<=last(tour$1)/8)
[mk] ah(last(tour$1) : SQUARE)
[mk] ah(SQUARE = 0..63)
[zm] /*last(tour$1)-17 : 0..63*/
[mk] pp(rp.0)
[mk] ax(last(tour$1)-17)
[zm] /*tour$1<-xx : valid_tour*/
[mk] ph(tour$1<-xx,
   !tt.(tt : seq(0..63) & tt~ : 0..63 ++> NATURAL =>
   (tt : valid_tour => !ii.(ii : 1..size(tt)-1
   =>
   tt(ii)|->tt(ii+1) : valid_move)) &
   (!ii.(ii : 1..size(tt)-1 => tt(ii)|->tt(ii+1) : valid_move)
   =>
   tt : valid_tour)))
[zm] /*tour$1<-xx : seq(0..63)*/
[mk] cls
[mk] ah(tour$1 : seq(SQUARE))
[mk] ah(xx : SQUARE)
[mk] ah(SQUARE = 0..63)
[mk] pp(rp.0)
[zm] /*(tour$1<-xx)~ : 0..63 ++> NATURAL*/
[mk] cls
[mk] ah(tour$1 : seq(SQUARE))
```
APPENDIX E. CORRECTNESS PROOF OF THE KNIGHT’S TOUR

\[
\begin{align*}
[\text{mk}] & \quad \text{ah}(\text{tour}^1 \mapsto \text{SQUARE} \mapsto \text{NATURAL}) \\
[\text{mk}] & \quad \text{ah}(\text{xx} \mapsto \text{SQUARE}) \\
[\text{mk}] & \quad \text{ah}(\text{not}(\text{xx} \mapsto \text{ran}(\text{tour}^1))) \\
[\text{mk}] & \quad \text{ah}(\text{SQUARE} = 0..63) \\
[\text{mk}] & \quad \text{pp}(\text{rp.0}) \\
[\text{mk}] & \quad \text{mh}(\text{ii} \mapsto \text{1..size}(\text{tour}^1 \mapsto \text{xx}) - 1 \Rightarrow \\
& \quad (\text{tour}^1 \mapsto \text{xx})(\text{ii}) \Rightarrow (\text{tour}^1 \mapsto \text{xx})(\text{ii} + 1) : \text{valid_move}) \Rightarrow \\
& \quad \text{tour}^1 \mapsto \text{xx} : \text{valid_tour}) \\
[\text{zm}] & \quad /*\text{ii} : \text{1..size}(\text{tour}^1 \mapsto \text{xx}) - 1 \Rightarrow \\
& \quad (\text{tour}^1 \mapsto \text{xx})(\text{ii}) \Rightarrow (\text{tour}^1 \mapsto \text{xx})(\text{ii} + 1) : \text{valid_move}*/ \\
[\text{zm}] & \quad /*(\text{tour}^1 \mapsto \text{xx})(\text{ii}) \Rightarrow (\text{tour}^1 \mapsto \text{xx})(\text{ii} + 1) : \text{valid_move}*/ \\
[\text{mk}] & \quad \text{ah}((\text{tour}^1 \mapsto \text{xx})(\text{ii}) = \text{tour}^1(\text{ii})) \\
[\text{zm}] & \quad /*(\text{tour}^1 \mapsto \text{xx})(\text{ii}) = \text{tour}^1(\text{ii})*/ \\
[\text{mk}] & \quad \text{pr} \\
[\text{mk}] & \quad \text{eh}((\text{tour}^1 \mapsto \text{xx})(\text{ii}), \text{tour}^1(\text{ii}), \text{Goal}) \\
[\text{zm}] & \quad /*\text{tour}^1(\text{ii}) \Rightarrow (\text{tour}^1 \mapsto \text{xx})(\text{ii} + 1) : \text{valid_move}*/ \\
[\text{mk}] & \quad \text{dc}(\text{ii} \leq \text{size}(\text{tour}^1) - 1) \\
[\text{mk}] & \quad \text{ah}((\text{tour}^1 \mapsto \text{xx})(\text{ii} + 1) = \text{tour}^1(\text{ii} + 1)) \\
[\text{zm}] & \quad /*(\text{tour}^1 \mapsto \text{xx})(\text{ii} + 1) = \text{tour}^1(\text{ii} + 1)*/ \\
[\text{mk}] & \quad \text{cls} \\
[\text{mk}] & \quad \text{ah}(\text{tour}^1 \mapsto \text{seq}(\text{SQUARE})) \\
[\text{mk}] & \quad \text{ah}(\text{xx} \mapsto \text{SQUARE}) \\
[\text{mk}] & \quad \text{ah}(\text{ii} : \text{1..size}(\text{tour}^1 \mapsto \text{xx}) - 1) \\
[\text{mk}] & \quad \text{ah}(\text{ii} \leq \text{size}(\text{tour}^1) - 1) \\
[\text{mk}] & \quad \text{pp}(\text{rp.0}) \\
[\text{mk}] & \quad \text{eh}((\text{tour}^1 \mapsto \text{xx})(\text{ii} + 1), \text{tour}^1(\text{ii} + 1), \text{Goal}) \\
[\text{zm}] & \quad /*\text{tour}^1(\text{ii}) \Rightarrow \text{tour}^1(\text{ii} + 1) : \text{valid_move}*/ \\
[\text{mk}] & \quad \text{ph}(\text{tour}^1, \\
& \quad !\text{tt.(tt} : \text{seq}(0..63) \& \text{tt} \mapsto 0..63 \mapsto \text{NATURAL} \Rightarrow \\
& \quad (\text{tt : valid_tour} \Rightarrow !\text{ii.(ii} : \text{1..size(tt)} - 1 \\
& \Rightarrow \\
& \quad \text{tt}(\text{ii}) \Rightarrow \text{tt}(\text{ii} + 1) : \text{valid_move}) \& \\
& \quad (!\text{ii.(ii} : \text{1..size(tt)} - 1 \Rightarrow \text{tt}(\text{ii}) \Rightarrow \text{tt}(\text{ii} + 1) : \text{valid_move}) \\
& \Rightarrow \\
& \quad \text{tt : valid_tour)))) \\
[\text{zm}] & \quad /*\text{tour}^1 : \text{seq}(0..63)*/ \\
[\text{mk}] & \quad \text{pr} \\
[\text{zm}] & \quad /*\text{tour}^1 \mapsto 0..63 \mapsto \text{NATURAL}*/ \\
[\text{mk}] & \quad \text{pr} \\
[\text{mk}] & \quad \text{ph}(\text{ii}, !\text{ii.(ii} : \text{1..size(\text{tour}^1)} - 1 \Rightarrow \\
& \quad \text{tour}^1(\text{ii}) \Rightarrow \text{tour}^1(\text{ii} + 1) : \text{valid_move})) \\
[\text{zm}] & \quad /*\text{ii} : 1..size(\text{tour}^1) - 1*/ \\
[\text{mk}] & \quad \text{cls}
APPENDIX E. CORRECTNESS PROOF OF THE KNIGHT’S TOUR

[113x758] ah(ii : 1..size(tour$1<-xx)-1)  
[113x800] ah(ii<=size(tour$1)-1)  
[113x842] pp(rp.0)  
[113x894] ah(ii = size(tour$1))  
[113x946] /*ii = size(tour$1)*/  
""""""  
[113x988] //  
[113x1040] ah(ii : 1..size(tour$1<-xx)-1)  
[113x1092] ah(not(ii<=size(tour$1)-1))  
[113x1144] pp(rp.0)  
[113x1196] /*tour$1(size(tour$1))|->
(tour$1<-xx)(size(tour$1)+1) : valid_move*/  
[113x1248] ah(tour$1(size(tour$1)) = last(tour$1))  
[113x1300] /*last(tour$1)|->xx : valid_move*/  
[113x1352] pr  
[113x1404] eh((tour$1<-xx)(size(tour$1)+1),xx,Goal)  
[113x1456] /*(tour$1<-xx)(size(tour$1)+1) = xx*/  
[113x1508] ph(last(tour$1),xx,  
""""""  
!(ss,tt).(ss : 0..63 & tt : 0..63 => (ss|->tt : valid_move =
2<=ss/8 & 2<=ss/8 & (1<=ss mod 8 & 1<=ss mod 8) &
(0 = -17+ss-tt & tt = ss-17)
or
(2<=ss/8 & 2<=ss/8 & (ss mod 8<=6 & ss mod 8<=6) &
(0 = -15+ss-tt & tt = ss-15))
or
(1<=ss/8 & 1<=ss/8 & (2<=ss mod 8 & 2<=ss mod 8) &
(0 = -10+ss-tt & tt = ss-10))
or
(1<=ss/8 & 1<=ss/8 & (ss mod 8<=5 & ss mod 8<=5) &
(0 = -6+ss-tt & tt = ss-6))
or
(ss/8<=6 & ss/8<=6 & (2<=ss mod 8 & 2<=ss mod 8) &
tt = 6+ss)
or
(ss/8<=6 & ss/8<=6 & (ss mod 8<=5 & ss mod 8<=5) &
tt = 10+ss)
or
(ss/8<=5 & ss/8<=5 & (1<=ss mod 8 & 1<=ss mod 8) &
""""""
\[ tt = 15 + ss \]
\[ \text{or} \]
\[ (ss/8 \leq 5 \text{ and } ss/8 \leq 5 \text{ and } (ss \text{ mod } 8 \leq 6 \text{ and } ss \text{ mod } 8 \leq 6) \text{ and } tt = 17 + ss) \]
\[ \& \]
\[ (2 \leq ss/8 \text{ and } 2 \leq ss/8 \text{ and } (1 \leq ss \text{ mod } 8 \text{ and } 1 \leq ss \text{ mod } 8) \text{ and } (0 = -17 + ss-tt \text{ and } tt = ss-17) \]
\[ \text{or} \]
\[ (2 \leq ss/8 \text{ and } 2 \leq ss/8 \text{ and } (ss \text{ mod } 8 \leq 6 \text{ and } ss \text{ mod } 8 \leq 6) \text{ and } (0 = -15 + ss-tt \text{ and } tt = ss-15) \]
\[ \text{or} \]
\[ (1 \leq ss/8 \text{ and } 1 \leq ss/8 \text{ and } (2 \leq ss \text{ mod } 8 \text{ and } 2 \leq ss \text{ mod } 8) \text{ and } (0 = -10 + ss-tt \text{ and } tt = ss-10) \]
\[ \text{or} \]
\[ (1 \leq ss/8 \text{ and } 1 \leq ss/8 \text{ and } (ss \text{ mod } 8 \leq 5 \text{ and } ss \text{ mod } 8 \leq 5) \text{ and } (0 = -6 + ss-tt \text{ and } tt = ss-6) \]
\[ \text{or} \]
\[ (ss/8 \leq 6 \text{ and } ss/8 \leq 6 \text{ and } (2 \leq ss \text{ mod } 8 \text{ and } 2 \leq ss \text{ mod } 8) \text{ and } tt = 6 + ss) \]
\[ \text{or} \]
\[ (ss/8 \leq 6 \text{ and } ss/8 \leq 6 \text{ and } (ss \text{ mod } 8 \leq 5 \text{ and } ss \text{ mod } 8 \leq 5) \text{ and } tt = 10 + ss) \]
\[ \text{or} \]
\[ (ss/8 \leq 5 \text{ and } ss/8 \leq 5 \text{ and } (1 \leq ss \text{ mod } 8 \text{ and } 1 \leq ss \text{ mod } 8) \text{ and } tt = 15 + ss) \]
\[ \text{or} \]
\[ (ss/8 \leq 5 \text{ and } ss/8 \leq 5 \text{ and } (ss \text{ mod } 8 \leq 6 \text{ and } ss \text{ mod } 8 \leq 6) \text{ and } tt = 17 + ss) \]
\[ \Rightarrow \]
\[ ss \rightarrow tt : \text{valid_move}) \]
(0 = -10+last(tour$1)-xx & xx = last(tour$1)-10))
or
(1<=last(tour$1)/8 & 1<=last(tour$1)/8 &
(last(tour$1) mod 8<=5 & last(tour$1) mod 8<=5) &
(0 = -6+last(tour$1)-xx & xx = last(tour$1)-6))
or
(last(tour$1)/8<=6 & last(tour$1)/8<=6 &
(2<=last(tour$1) mod 8 & 2<=last(tour$1) mod 8) &
xx = 6+last(tour$1))
or
(last(tour$1)/8<=6 & last(tour$1)/8<=6 &
(last(tour$1) mod 8<=5 & last(tour$1) mod 8<=5) &
xx = 10+last(tour$1))
or
(last(tour$1)/8<=5 & last(tour$1)/8<=5 &
(1<=last(tour$1) mod 8 & 1<=last(tour$1) mod 8) &
xx = 15+last(tour$1))
or
(last(tour$1)/8<=5 & last(tour$1)/8<=5 &
(last(tour$1) mod 8<=6 & last(tour$1) mod 8<=6) &
xx = 17+last(tour$1)) =>$last(tour$1)|->xx : valid_move

[zm] /*
2<=last(tour$1)/8 & 2<=last(tour$1)/8 &
(1<=last(tour$1) mod 8 & 1<=last(tour$1) mod 8) &
(0 = -17+last(tour$1)-xx & xx = last(tour$1)-17)
or
(2<=last(tour$1)/8 & 2<=last(tour$1)/8 &
(last(tour$1) mod 8<=6 & last(tour$1) mod 8<=6) &
(0 = -15+last(tour$1)-xx & xx = last(tour$1)-15))
or
(1<=last(tour$1)/8 & 1<=last(tour$1)/8 &
(2<=last(tour$1) mod 8 & 2<=last(tour$1) mod 8) &
(0 = -10+last(tour$1)-xx & xx = last(tour$1)-10))
or
(1<=last(tour$1)/8 & 1<=last(tour$1)/8 &
(last(tour$1) mod 8<=5 & last(tour$1) mod 8<=5) &
(0 = -6+last(tour$1)-xx & xx = last(tour$1)-6))
or
(last(tour$1)/8<=6 & last(tour$1)/8<=6 &
(2<=last(tour$1) mod 8 & 2<=last(tour$1) mod 8) &
xx = 6+last(tour$1))
or
(last(tour$1)/8<=6 & last(tour$1)/8<=6 &
APPENDIX E. CORRECTNESS PROOF OF THE KNIGHT’S TOUR

(last(tour$1) mod 8 <= 5 & last(tour$1) mod 8 <= 5) &
xx = 10 + last(tour$1))
or
(last(tour$1)/8 <= 5 & last(tour$1)/8 <= 5 &
(1 <= last(tour$1) mod 8 & 1 <= last(tour$1) mod 8) &
xx = 15 + last(tour$1))
or
(last(tour$1)/8 <= 5 & last(tour$1)/8 <= 5 &
(last(tour$1) mod 8 <= 6 & last(tour$1) mod 8 <= 6) &
xx = 17 + last(tour$1))*/

[mk] cls
[mk] ah(2 <= last(tour$1)/8)
[mk] ah(1 <= last(tour$1) mod 8)
[mk] ah(last(tour$1) - 17 = xx)
[mk] pp(rp.0)

/*SUCCESS*/
Bibliography


