School of Computing Science

On the Mechanisation of the Logic of Partial Functions

Thesis by

Matthew James Lovert

In partial fulfillment of the requirements
for the Degree of Doctor of Philosophy.

July 2013

Abstract

It is well known that partial functions arise frequently in formal reasoning about programs. A partial function may not yield a value for every member of its domain. Terms that apply partial functions thus may not denote, and coping with such terms is problematic in two-valued classical logic. A question is raised: how can reasoning about logical formulae that can contain references to terms that may fail to denote (partial terms) be conducted formally? Over the years a number of approaches to coping with partial terms have been documented. Some of these approaches attempt to stay within the realm of two-valued classical logic, while others are based on non-classical logics. However, as yet there is no consensus on which approach is the best one to use. A comparison of numerous approaches to coping with partial terms is presented based upon formal semantic definitions.

One approach to coping with partial terms that has received attention over the years is the Logic of Partial Functions (LPF), which is the logic underlying the Vienna Development Method. LPF is a non-classical three-valued logic designed to cope with partial terms, where both terms and propositions may fail to denote. As opposed to using concrete undefined values, undefinedness is treated as a "gap", that is, the absence of a defined value. LPF is based upon Strong Kleene logic, where the interpretations of the logical operators are extended to cope with truth value "gaps".

Over the years a large body of research and engineering has gone into the development of proof based tool support for two-valued classical logic. This has created a major obstacle that affects the adoption of LPF, since such proof support cannot be carried over directly to LPF. Presently, there is a lack of direct proof support for LPF.

An aim of this work is to investigate the applicability of mechanised (automated) proof support for reasoning about logical formulae that can contain references to partial terms in LPF. The focus of the investigation is on the basic but fundamental two-valued classical logic proof procedure: resolution and the associated technique proof by contradiction. Advanced proof techniques are built on the foundation that is provided by these basic fundamental proof techniques. Looking at the impact of these basic fundamental proof techniques in LPF is thus the essential and obvious starting point for investigating proof support for LPF. The work highlights the issues that arise when applying these basic techniques in LPF, and investigates the extent of the modifications needed to carry them over to LPF. This work provides the essential foundation on which to facilitate research into the modification of advanced proof techniques for LPF.

Acknowledgements

I would like to thank both of my supervisors Cliff Jones and Jason Steggles for their advice and support throughout.

I would like to thank Leo Frietas and Andrius Velykis for discussions that took place. I would also like to thank Gudmund Grov for helping me to get started with using Isabelle.

Additionally, I also want to thank the numerous anonymous referees of papers that have been submitted on some of this work for their comments.

This work was supported by an EPSRC Studentship.

Publications

Some parts of this thesis have already been published:

- Some of the content on semantic definitions presented in Sections 3.1, 3.2, and 3.3.2 is presented in [Lov10], but has been extended for this thesis.
- Some of the content on semantic definitions presented in Sections 3.3.2 (similar to the content in [Lov10]) and 3.4 is presented in [JL11]. The content has again been thoroughly extended and modified in this thesis. This paper also presents proofs about a partial predicate done by Cliff Jones which forms no part of this thesis.
- A preliminary draft of a small subset of Section 4.1 appeared in [JLS12b]. Such semantic definitions have been overhauled with significant modifications and extensions in this thesis. Numerous other semantic definitions alongside comparisons on the semantic definitions are presented in this thesis.
- Some of the results from Chapter 6 are published in [JLS12a]. The work has been extended for this thesis.

These papers are referenced and discussed further in the relevant chapters of this thesis.

Contents

1 Introduction **1**
1.1 Setting the Scene 1
1.1.1 Mathematical Logic 1
1.1.2 Formal Methods 2
1.1.3 Proofs 3
1.1.4 The Issues that Arise When Reasoning About Partial Functions 4
1.2 Aims 11
1.3 Contributions 12
1.4 Structure of the Thesis 13

2 Background **15**
2.1 Categorising the Different Approaches to Coping with Partial Terms 16
2.2 Approaches to Coping with Partial Terms 18
2.2.1 Reformulating Expressions and Function Definitions . . . 18
2.2.2 Classical Approaches 20
2.2.3 Semi-Classical Approaches 25
2.2.4 Non-Classical Logic Approaches 30
2.2.5 Sequent Interpretations 42
2.3 Summary of the Justifications for LPF 44
2.4 Previous Attempts at Mechanisation Support for LPF 49
2.4.1 The VDM Toolset and the Overture Toolset 49
2.4.2 mural 50
2.4.3 Utilising Existing Theorem Provers 51
2.5 Conclusions 52

3 Semantic Definitions for LPF **54**
3.1 Expression Constructs 56
3.2 Context Conditions 58
3.3 Operational Semantics 63
3.3.1 Big-Step Structural Operational Semantics Definition . . 65
3.3.2 Small-Step Structural Operational Semantics Definition . 67
3.4 Denotational Semantics 80
3.5 Relationships between the Semantic Definitions 84

3.6 Conclusions 89

4 Comparison of Approaches to Coping with Partial Terms 91

4.1 Alternative Semantic Definitions 93

4.1.1 Relations 94

4.1.2 Forcing all Terms to Denote 97

4.1.3 Semi-Classical Approaches 105

4.1.4 Weak Kleene Logic 112

4.1.5 McCarthy's Conditional Operators 113

4.1.6 Łukasiewicz's Logic 115

4.1.7 Bochvar's External Logic 115

4.1.8 Concluding Remarks 120

4.2 Comparing the Sequent Interpretations 123

4.2.1 SS 124

4.2.2 SW 124

4.2.3 WW 124

4.2.4 WS 125

4.3 Comparisons between the Different Approaches to Coping with Partial Terms 125

4.3.1 Comparison 1: The Values Denoted in the Different Approaches 125

4.3.2 Comparison 2: Non-classical Logic Comparisons 128

4.4 Relationships between the Different Approaches to Coping with Partial Terms and LPF 130

4.4.1 Relationship with Two-Valued Classical Logic using Overspecification 131

4.4.2 Relationship with the Well-Definedness Approach 132

4.4.3 Relationship with Two-Valued Classical Logic using Strong Equality 132

4.4.4 Relationship with the Weak Kleene Approach 134

4.4.5 Relationship with the McCarthy Conditional Operator Approach 134

4.4.6 Relationship with Łukasiewicz's Approach 136

4.4.7 Relationship with Bochvar's External Approach 136

4.4.8 Concluding Remarks 137

4.5 Conclusions 139

5 Mechanising LPF Semantic Definitions 141

5.1 Maude Mechanisation 141

5.1.1 An Introduction to Maude 142

5.1.2 Mechanising the SOS Definitions . . . 144
5.2 Isabelle Mechanisation . . . 169
5.3 Conclusions . . . 182

6 Investigating Proof Procedures in LPF 184
6.1 Validity and Satisfiability . . . 185
6.1.1 Two-Valued Classical Logic Recap . . . 185
6.1.2 LPF . . . 186
6.2 The Method of Truth Tables . . . 187
6.3 Clausal Form . . . 188
6.3.1 Two-Valued Classical Logic Recap . . . 188
6.3.2 LPF . . . 189
6.4 Unification . . . 198
6.5 Resolution . . . 199
6.5.1 Two-Valued Classical Logic Recap . . . 199
6.5.2 LPF . . . 201
6.6 Refutation Procedures . . . 204
6.6.1 Two-Valued Classical Logic Recap . . . 204
6.6.2 LPF . . . 205
6.7 Conclusions . . . 215

7 Concluding Remarks 220
7.1 Summary and Conclusions . . . 220
7.2 Future Work . . . 224
7.2.1 Further Comparison Results . . . 225
7.2.2 Further Performance Results . . . 226
7.2.3 Implementing the Proof Techniques Considered . . . 226
7.2.4 Investigating the Applicability of Further Proof Techniques for LPF . . . 227

Bibliography 231

A Full LPF Semantic Definitions 240
A.1 Abstract Syntax . . . 240
A.1.1 Expression Constructs . . . 240
A.1.2 Syntactic Definitions . . . 241
A.1.3 Function Definitions and Predicate Definitions . . . 241
A.2 Context Conditions . . . 242
A.2.1 Type Map . . . 242
A.2.2 Expressions . . . 242

A.2.3 Function Definitions and Predicate Definitions 244
A.3 Structural Operational Semantics 244
A.3.1 Semantic Object 244
A.3.2 Big-Step Structural Operational Semantics Definition . . 245
A.3.3 Small-Step Structural Operational Semantics Definition . 249
A.4 Denotational Semantics 254
A.4.1 Expressions 254
A.4.2 Context Conditions 254
A.4.3 Semantic Object 255
A.4.4 Denotational Semantic Definition 1 255
A.4.5 Denotational Semantic Definition 2 257

B Selected Mathematical VDM-SL Notation 259
B.1 Type Definitions 259
B.1.1 Record Type 259
B.1.2 The Boolean Data Type 259
B.1.3 The Numeric Data Types 260
B.1.4 The Set Type 260
B.1.5 The Sequence Type 261
B.1.6 The Map Type 262
B.1.7 The Union Type 262
B.1.8 Pairs 262
B.2 Expressions 263
B.2.1 Quantified Expressions 263
B.2.2 The Conditional Expression 263
B.2.3 The Cases Expression 263
B.2.4 The Let Expression 263
B.3 Function Definitions 264
B.4 Inference Rules 264
B.5 Proofs 264

C Glossary 266

List of Figures

2.1 The truth table for weak equality 26
2.2 The truth table for existential equality 26
2.3 The truth table for strong equality 28
2.4 The weak Kleene truth tables for disjunction, conjunction, implication and negation 31
2.5 The McCarthy truth tables for disjunction, conjunction, implication and negation 33
2.6 An ordering on truth values 33
2.7 The LPF truth tables for disjunction, conjunction, implication and negation 36
2.8 The LPF truth table for the definedness operators 37
2.9 The Łukasiewicz truth tables for disjunction, conjunction, implication and negation 40
2.10 The Bochvar external truth tables for disjunction, conjunction, implication and negation 41
2.11 The Bochvar external truth tables for the assertion operator and for the denial operator 42
2.12 A proof of a zero function property 45
2.13 A proof of a zero function property 46

3.1 The expression context conditions (part 1) 61
3.2 The expression context conditions (part 2) 62
3.3 The function definitions context conditions 62
3.4 The predicate definitions context conditions 62
3.5 A sample small-step SOS expression evaluation 70
3.6 A sample small-step SOS "gap" expression evaluation 71
3.7 A further sample small-step SOS expression evaluation 79
3.8 The $\mathcal{E}$ semantic function definition 85

4.1 The $\mathcal{E}^R$ semantic function definition 96
4.2 The $\mathcal{E}^C$ semantic function definition 100
4.3 The $\mathcal{E}^D$ semantic function definition 104
4.4 The $\mathcal{E}^\exists$ semantic function definition (part 1) 107
4.5 The $\mathcal{E}^\exists$ semantic function definition (part 2) 108
4.6 The $\mathcal{E}^{==}$ semantic function definition (part 1) 111

4.7 The $\mathcal{E}^{==}$ semantic function definition (part 1) 112
4.8 The $\mathcal{E}^{W}$ semantic function definition 114
4.9 The $\mathcal{E}^{M}$ semantic function definition 116
4.10 The $\mathcal{E}^{\L}$ semantic function definition (part 1) 117
4.11 The $\mathcal{E}^{\L}$ semantic function definition (part 2) 118
4.12 The $\mathcal{E}^{B}$ semantic function definition (part 1) 121
4.13 The $\mathcal{E}^{B}$ semantic function definition (part 2) 122

5.1 The expressions used for the SOS rewrite rule comparison 168

6.1 An illustrative resolution refutation attempt 212
6.2 An illustrative resolution refutation subproof (part 1) 213
6.3 An illustrative resolution refutation proof establishing definedness (part 2) 213
6.4 An illustrative (optimised) resolution refutation proof establishing definedness 215
6.5 An illustrative resolution refutation proof establishing definedness 219
6.6 An illustrative resolution refutation proof where definedness obligations do not need introducing 219

List of Tables

4.1 An outline of the changes made to the LPF semantic function definition to define the other semantic function definitions . . . 123

4.2 A comparison of numerous approaches to coping with partial terms . . . 127

4.3 A comparison of three-valued logic approaches to coping with partial terms . . . 129

5.1 A number of rewrite comparisons between the big-step Structural Operational Semantic and the small-step Structural Operational Semantic definitions . . . 169

Chapter 1

Introduction

Contents

1.1 Setting the Scene 1
1.1.1 Mathematical Logic 1
1.1.2 Formal Methods 2
1.1.3 Proofs 3
1.1.4 The Issues that Arise When Reasoning About Partial Functions 4
1.2 Aims 11
1.3 Contributions 12
1.4 Structure of the Thesis 13

An aim of this thesis is to investigate the applicability of mechanised proof support for reasoning in the Logic of Partial Functions about logical formulae that can contain references to terms that may fail to denote proper values (partial terms), for instance, arising from the application of partial functions. In this chapter the scene is first set by briefly introducing some relevant background information, as well as highlighting the issues that arise when reasoning about partial functions, which provides the motivation for this work. The aims of this thesis are then discussed, which is followed by an overview of how they will be addressed over the course of this thesis.

1.1 Setting the Scene

In this section the scene is set by briefly introducing the topics of mathematical logic, formal methods, and proofs. Then the issues that arise when faced with reasoning about partial functions are introduced, which provides the motivation for the aims of this thesis.

1.1.1 Mathematical Logic

"The study of logic was begun by the ancient Greeks... where it was used to formalize deduction: the derivation of true statements from statements that are assumed to be true" [BA01]. At a later date mathematicians started using logic to study the foundations of mathematics [BA01]. Mathematical logic is used extensively today in computer science. A detailed history of mathematical logic is not presented here as the reader can refer to other texts, such as [BA01, Bun10, Wal97].

In propositional logic, formulae are built up from the constant truth values true and false, and from propositional variables. These can be combined using logical operators (connectives) which are given a precise formal meaning [BA01]. Formulae are traditionally two-valued, that is, they take one of two truth values either true or false.

First-order (predicate) logic extends from propositional logic. In first-order logic formulae can also be built up from non-propositional variables and constants using functions and predicates, and non-propositional variables can be quantified [Har09]. This provides a logic which is much more expressive than propositional logic.

1.1.2 Formal Methods

Software is becoming more complex, which means that there is a greater chance of errors being present. Furthermore, software is increasingly being used in situations where the failure of the software can put lives at risk, for example, in onboard aircraft systems and in medical systems. The failure of software can also lead to huge financial ramifications etc. Thus it is of no big surprise that software correctness is an important research topic in computer science. Formal methods are an approach to increasing confidence in computer systems.

Formal methods are mathematical techniques used in the development of computer systems [WLBF09], for specifying and verifying systems [CW96]. The use of formal methods is warranted by the expectation that mathematical analysis can contribute to the reliability and to the robustness of a design [Hol97]. The complexity of such mathematical proofs, and the time that it can take to discharge such proofs leads to limiting the extent to which formal methods are applied in practice [JJLM91].

The term formal methods can be used to describe: "writing a formal specification; proving properties about the specification; constructing a program by mathematically manipulating the specification; and verifying a program by mathematical argument" [Hal90]. Formal methods for instance, can refer to mathematically proving that properties of a system hold before it is implemented [JJLM91].

A formal specification is used to provide a precise definition of what a system should do and the system properties that are desired. Writing such specifications ensures that properties and requirements of a system are written down formally. Such specifications can lead to uncovering inconsistencies and design flaws etc. [CW96, Jon90].

Formal specifications employ a mathematical (logic) notation. Mathematical logic is used to verify programs. Two approaches to formal verification are model checking and theorem proving. Such techniques are used to check whether a system has the desired properties [CW96]. Given a model of a system, model checking is used to check whether a property holds in that model [CW96]. Theorem proving is the process of finding a proof of a property, where the different steps in a proof are justified by referring to known facts [CW96].

Numerous formal methods exist such as VDM [Jon90] and Z [Spi92]. They have been used successfully in industry, for instance in specifying safety critical software and in verifying hardware designs [CW96]. For an overview of the use of formal methods in industry the reader is referred to [CW96] and to [WLBF09].

1.1.3 Proofs

A mathematical proof is an argument that some statement/claim is true, whenever some assumed statements are true. Proofs are used frequently in the context of formal specifications, for instance in discharging proof obligations about formal specifications, and for showing that expected properties of a formal specification hold [BFL$^+$94]. A formal language allowing statements to be written, along with an interpretation giving a meaning to statements, and a set of inference rules and axioms which are rules stating how statements can be inferred from other statements, (where an axiom is an inference rule whose truth is taken without question), are needed as a basis for conducting proofs [BFL$^+$94].

Depending on the purpose of the proofs, they can be conducted (written) with different levels of formality [BFL$^+$94]. At one extreme is informal proofs. Informal proofs generally put forth a high-level argument that attempts to convince a reader that a claim holds. The lack of formality in such informal proofs means that they cannot be checked by tools, and thus there is a reliance on the reader to check that they are correct. Additionally, in informal proofs it could be the case that large steps are made without detailed justification. Thus such proofs are susceptible to errors.

At the other extreme are formal proofs. In formal proofs the level of detail in the proofs is greater than in informal proofs. Such proofs are generally conducted a step at a time, where each step is justified by referring to known rules, where generally no big jumps are made (certain tools may though allow for a number of steps to be made in a single step, for instance, steps that simplify formulae). Tool assistance can be used to aid in the development of such proofs, and for checking that such proofs are correct. Examples of such tools include PVS [ORS92], Isabelle/HOL [NWP02], and CVC [BT07]. Formal proofs can be checked for correctness by a tool, as the task can generally just be reduced to an exercise in symbol manipulation. Checking the correctness of steps in a formal proof can be done by *pattern matching* against rules [BFL$^+$94].

A proof must serve the purpose of eliminating any doubt about the claim being made not following. Proofs that lack a lot of formality cannot *generally* eliminate such doubt. The highest level of confidence in proofs can be gained by constructing formal proofs, since their detailed steps mean that tools can be used to check such proofs for correctness.

Undertaking mathematical proofs is generally a hard task, and as a result tool support is available to help in writing formal proofs (theorem provers). In theorem proving a user generally provides a set of rules taken to be true, as well as a formula to be proved. It is the purpose of the theorem prover to attempt to construct a proof, or at least to help the user to find a proof that the formula under consideration holds. Such tools can be aided by proof techniques such as resolution, paramodulation, and semantic tableauxs [BA01, Har09, Bun10].

In interactive theorem proving the onus is on the user to complete the proof but given the aid of a tool. The user could, for instance, have to provide a proof step by step, but significant proof tasks could still be performed automatically by the tool. Such a tool may only check each step of a user's proof for correctness. At another extreme of theorem proving are automated theorem provers, which will attempt to prove a formula, for instance by following pre-programmed strategies to attempt to find a proof of a formula under consideration automatically. However, such tools may still require user assistance/guidance [GMW79, BA01].

1.1.4 The Issues that Arise When Reasoning About Partial Functions

The interest in this thesis is reasoning about logical formulae that can contain reference to terms that may fail to denote, for example, arising from the application of partial functions. The terms total function and partial function will first be introduced, followed a discussion of the issues that arise when reasoning about partial functions with examples which provides motivation for this work. This section leads into the aims and the contributions of this thesis.

A total function is a function that will yield a result (value) for every member of its domain. The domain is the set of values to which a function may be applied. A total function is defined on all values that are within its domain. No matter what arguments are passed into a total function, a term that applies a total function will always denote a value, that is, the term will be defined.

A partial function may not yield a result for every member of its domain. Thus reasoning about partial functions is more problematic as a term that applies a partial function can fail to denote, as a partial function may not yield a proper defined value for some or possibly all of the arguments in its domain that it can be applied to. A defined domain is the set of values to which a function may be applied, where the function will yield a defined result. These two domains are the same for total functions, but for partial functions these two domains are different from each other. For instance, the domain of the partial integer division function is $\mathbb{Z} \times \mathbb{Z}$, but the defined domain of the partial integer division function is $\mathbb{Z} \times (\mathbb{Z} \setminus \{0\})$.

Reasoning about partial functions is necessary since they arise frequently in computing, see e.g. [CJ91, Jon06, Far96, Owe97]. Partial functions arise for example in the specification of computer programs (for instance in VDM and Z), where they can arise from recursive function definitions which are only defined when the recursion terminates and yields a defined value. Program specifications also employ a number of data types, such as sequences and maps which have associated operators that are partial, such examples include taking the head of a sequence (the sequence could be empty) and map lookup (the value may not exist in the map). Other examples where partiality can arise includes from array indexing (an invalid array index), division by zero, and taking the *log* of zero. Such partial operators are used frequently in program specifications, and there is a great need for recursive functions.

A term that applies a partial function with argument(s) from outside of the partial function's defined domain will not denote (the term denotes no value) and this is known as a partial term, or an undefined term, or a non-denoting term. The terms partial term, undefined term, and non-denoting term are used interchangeably throughout this thesis. Allowing partial functions (and partial operators) to occur leads one to having to reason about potential partial terms in proofs.

Formulae like $i/i = 1$, $log(i) = 0$, and $A[i] = 5$, can all be true, false or undefined, depending on the value assigned to the free variable i. Also consider a sentence such as i *is tall*. This is true for some i, false for some i, and neither true or false for some i; the sentence is *vague*. Undefinedness can propagate, since the term $1/0$ is undefined, the term $1 + (1/0)$ is undefined.

People have also argued (refer to [Sid10] for an introduction as well [Pri53] and [But55]) that propositions about the future also pose problems in classical logic, since propositions about the future are neither true nor false when stated. A sentence such as *it will rain next Wednesday* is neither true nor false at this moment in time, since it is not yet determined that it will rain next Wednesday, (cf. the arguments about Aristotle's sea battle argument [Pri53, But55]).

It is illustrated below that reasoning about logical formulae that can contain references to partial terms, for instance, arising from the application of partial functions, is problematic in two-valued classical logic. Numerous approaches to reasoning about logical formulae that can contain references to partial terms though have been proposed, for instance in [Kle52, McC67, CJ91, Owe97, FFL97, Far96, MS97, Meh08, GS95, Jon06, SB99, Art96, Häh05, WF08, Fit07, JL11, Sch11]. A review of these different approaches is presented in Chapter 2.

The issues that arise due to partial terms arose a long time ago due to the use of definite descriptors [SDG99]. Russell [Rus05] in the early 1900s discusses such issues using the infamous example of *the present King of France* among others, and introduces his own theory, where partial terms stand for no object (since France is a republic now the phrase *the present King of France* refers to no object), but assertions like *the King of France is bald* are false), as well as outlining the theories of Meinong and Frege, which are theories where *the King of France* would stand for some object. Refer to [SDG99] for such an historical perspective. Some of these ideas can be seen underlying the approaches that are considered in Chapter 2.

The focus in this thesis is on partial terms that arise through the application of partial functions in program specifications. Approaches to coping with partial terms are considered in Chapter 2.

In [Häh05] three different kinds of undefinedness/partiality that can be encountered in program specifications are discussed:

- Non-termination: "A subcomputation needed for the evaluation of an expression does not terminate" [Häh05];
- Error value: "A computation has an erroneous result, because it was called with an illegal value... an illegal value is not intended to occur, but if it does, one has to handle it" [Häh05]; and
- Non-determinism: "In contrast to error values, indeterminate values typically are intentional... An expression could be an error, but it could just as well be loosely specified: it has a defined value, but it is left to an implementation to fix that value" [Häh05].

To further illustrate the problems with partial terms consider the following formula where $\mathbf{hd}\, s$ extracts the first element from the sequence s:

$$s = [\,] \vee \mathbf{hd}\, s = 5$$

When s is an empty sequence the term $\mathbf{hd}\, s$ fails to denote a proper value; it is known as a partial term. The second disjunct will contain a partial term whenever the first disjunct is true.

Furthermore, consider the following formula where j is an integer and m is a map from $\mathbb{Z}$ to $\mathbb{Z}$:

$$j \in \mathbf{dom}\,(m) \wedge m(j) = 3$$

Notice that the map lookup in the second conjunct gives rise to a partial term when the first conjunct is false.

In these two examples the question is raised as to what meaning is to be given to the logical formulae given the existence of partial terms.

To further illustrate the issue of partial terms consider the *zero* function which is deliberately partial and was first presented in [CJ91]:

$$\begin{aligned}&\mathit{zero} : \mathbb{Z} \to \mathbb{Z}\\ &\mathit{zero}(i) \quad \triangleq \quad \mathbf{if}\ i = 0\ \mathbf{then}\ 0\ \mathbf{else}\ \mathit{zero}(i-1)\end{aligned}$$

This function is defined to return 0 when $i \geq 0$. However, when $i < 0$, the term $\mathit{zero}(i)$ will fail to denote an integer value, it will be a partial term.[1] For example, $\mathit{zero}(5)$ returns an integer value notably 0, but $\mathit{zero}(-1)$ is a partial term (it denotes no value).

The *zero* function has been chosen primarily because it allows for the issues surrounding undefinedness to be illustrated through such a simple definition. This *zero* function and a related *subp* function (see Chapter 2) have been promoted by Cliff Jones [Jon90, CJ91, Jon06] as a way of testing approaches to coping with partial terms.

In the following it is being considered that functions are evaluated according to a strict semantics, that is, if an argument passed into a function is undefined then the function itself is undefined. Of course, functions can also be undefined

[1]The domain could be restricted to $\mathbb{N}$ since the domain is just a single set and therefore the restricted set $\mathbb{N}$ can be used where all of the elements satisfy the precondition that is presented. That is, the *zero* function will be total over this restricted set $\mathbb{N}$. However, taking such an approach is not always this straightforward. Consider that the precondition is a relation between multiple domain elements. Taking this approach is considered in Section 2.2.

even if only defined arguments are passed into them.

The following property of the *zero* function attempts to capture the defined domain of the *zero* function, and further illustrates how the issues of partial terms can be manifested into logical formulae:

$$\forall i : \mathbb{Z} \cdot i \geq 0 \;\Rightarrow\; \mathit{zero}(i) = 0 \tag{1.1}$$

it should be clear that the truth of this property relies on the truth of implications such as:

$$1 \geq 0 \;\Rightarrow\; \mathit{zero}(1) = 0$$

which evaluates to:

$$\mathbf{true} \;\Rightarrow\; 0 = 0$$

and further to:

$$\mathbf{true} \;\Rightarrow\; \mathbf{true}$$

which is clearly true.

However, the truth of this property also relies on the truth of implications such as:

$$-1 \geq 0 \;\Rightarrow\; \mathit{zero}(-1) = 0$$

where the term $\mathit{zero}(-1)$ does not denote an integer value. There is a "gap", that is, an absence of a defined value. Blamey used the notion of "gaps" in the value space/in truth values, as opposed to an explicit undefined value [Bla80].

The term "gap" will be used irrespective of the type of undefinedness from the three types of undefinedness that can arise and that were listed above, so no distinction will be made between the different types of undefinedness in what follows. It is, however, convenient to illustrate the difficulties by writing $\perp_{\mathbb{Z}}$ and $\perp_{\mathbb{B}}$ to stand for missing integer values ("gaps") and missing Boolean values ("gaps") respectively.

Thus this example evaluates to:

$$\mathbf{false} \;\Rightarrow\; \perp_{\mathbb{Z}} = 0$$

when considering weak (strict) equality (which fails to denote if either operand fails to denote) means that this formula further evaluates to:

$$\textbf{false} \;\Rightarrow\; \perp_{\mathbb{B}}$$

where a non-denoting truth value (a "gap", that is, the absence of a truth value) has arisen. A partial term from the application of the *zero* function has propagated up.

This does not make any formal sense in two-valued classical logic since the truth tables only define the logical operators for proper Boolean values ($\mathbb{B}$, $\{\textbf{true}, \textbf{false}\}$), and no mention is made of formulae that fail to denote a Boolean value.

Referring again to Property 1.1 the reader may want to interpret the implication as a "guard", that is, whenever the antecedent is false then the implication is true. In other words interpreting the antecedent of the implication as "guarding" the implication from the possible partial term (a "gap") in the consequent, but there is no formal sense in two-valued classical logic in which the antecedent being false overcomes the problem of a "gap" in the consequent. Furthermore, it is not at all wise to rely on such a "guard" being present. A standard law in two-valued classical logic is that the contrapositive ($\neg\, q \;\Rightarrow\; \neg\, p$) of an implication is equivalent to the implication ($p \;\Rightarrow\; q$):

$$\forall i\colon \mathbb{Z} \cdot \neg\,(\mathit{zero}(i) = 0) \;\Rightarrow\; i < 0 \tag{1.2}$$

where the so called "guard" is less obvious.

A more problematic property of the *zero* function is:

$$\forall i\colon \mathbb{Z} \cdot \mathit{zero}(i) = 0 \vee \mathit{zero}(-i) = 0 \tag{1.3}$$

where it is clear that with the exception of the case when i denotes 0 one of the disjuncts will fail to denote a proper value. Depending on the value of i either of the operands can fail to denote a proper value. It should be clear that the truth of Property 1.3 relies on the truth of disjunctions such as:

$$\mathit{zero}(1) = 0 \vee \mathit{zero}(-1) = 0$$

which again since the term $\mathit{zero}(-1)$ does not denote an integer value evaluates to:

$$0 = 0 \vee \perp_{\mathbb{Z}} = 0$$

and due to the notion of weak equality this further evaluates to:

$$\mathbf{true} \vee \perp_{\mathbb{B}}$$

which again makes no sense in two-valued classical logic, since the truth tables of two-valued classical logic are only defined for proper Boolean values. In [Far90] Farmer states that reasoning about partial functions in classical logic is problematic as they can lead to a violation of the *existence assumption*, that is, that all terms have a denotation.

Examples can be constructed that serve the same illustration purpose as the *zero* function did but using division instead:

$$\forall i : \mathbb{Z} \cdot i \neq 0 \;\Rightarrow\; i/i = 1 \tag{1.4}$$

$$\forall i : \mathbb{Z} \cdot (i/i = 1) \vee ((i-1)/(i-1) = 1) \tag{1.5}$$

Specification languages in particular must handle partial terms, since partial terms arise frequently, and they pose problems. At ZUM97 (the Z User Meeting), it was reported by Mark Saaltink (the author of the Z Eves proof tool), that not one of 400 published Z specifications analysed was free of errors caused by undefined terms [SDG99].

The big question that is raised due to the presence of partial terms is how can reasoning about logical formulae that can contain references to partial terms be conducted formally. Approaches to coping with partial terms must provide an answer to this question. They must also address the issue of what terms like $zero(-1)$ and $0/0$ denote (they could just be left as a "gap" to be dealt with by other constructs), or eliminate such terms completely.

Chapter 2 outlines numerous approaches that have been proposed over the years to handle logical formulae that can contain references to partial terms. As yet there is no consensus on which is the best approach to cope with partial terms. Approaches include those that attempt to provide "workarounds" to remain within the realm of two-valued classical logic, and those approaches that make use of non-classical (three-valued) logics.

Non-classical logics have long been used to model undefinedness in formal specification languages [BCJ84]. The approach that the main body of this thesis is based on is known as the Logic of Partial Functions (LPF for short) [BCJ84, Che86, CJ91, JM94, Jon06], which is a non-classical (three-valued) logic, where a formula can be true, false, or undefined (a "gap"), and the interpretations of the logical operators are extended to cope with such "gaps". LPF is one of the approaches discussed in Chapter 2. Section 2.3 presents justifications for using LPF over other approaches that have been proposed over the years that allow for reasoning about logical formulae that can include references to partial terms. LPF is used in the Vienna Development Method (VDM).

A big obstacle to the use of LPF (and of non-classical logics in general) is that it is an unfamiliar logic. For instance, the available proof rules in LPF, differ from those of two-valued classical logic. In particular the law of the excluded middle does not hold in LPF. Thus more effort is required from a user who may be familiar with two-valued classical logic to learn how to reason in LPF.

The fact that LPF deviates from the world of two-valued classical logic leads to another big obstacle against the adoption of LPF, that being, that a large body of research and engineering has gone into two-valued classical logic, which has led to a wide range of proof procedures and to the development of (interactive/automated) proof based tool support for two-valued classical logic. All of this proof support cannot be reused without change for LPF due to its three-valued nature to cope with the occurrence of partial terms. Thus, it is the case that mechanised (automated) proof support for LPF requires additional effort. Proof support for LPF remains a subject of debate and research [Fit07]. Appropriate proof support to aid reasoning in LPF can go a long way to addressing this obstacle against the adoption of LPF, and investigating this topic is a major aim of this thesis.

1.2 Aims

LPF is a non-classical logic, which has for a long time been considered a viable candidate solution within which to conduct reasoning about logical formulae that can contain references to partial terms. A major obstacle affecting the adoption of LPF is that there is a distinct lack of direct proof support available for LPF.

An aim of this work is to:

Research into the applicability of mechanised (automated) proof support for LPF.

The thesis argues that the basic ideas of the two-valued classical logic proof procedure resolution and the associated technique of proof by contradiction [Bun10, BA01] can be reused for reasoning in LPF when supplemented with vital modifications to cover LPF. Furthermore, it argues that these procedures can be modified efficiently for LPF.

Being able to re-use the basis of two-valued classical logic proof procedures to be able to reason in LPF is essential. This ensures that existing ideas and work can be extended for LPF, rather than having to start from scratch. For instance, existing code bases and tool support can be adapted for LPF.

There is a question over whether the use of LPF will lead to a substantial increase in the work needed when applying proof procedures, compared to in two-valued classical logic. An efficient mechanisation of proof procedures is essential to support any future use of and future work on LPF.

An aim of this work as already mentioned is concerned with work on the mechanisation of LPF. In this thesis providing a formal comparison of approaches to coping with partial terms is also an aim. This is used to argue for the use of LPF for reasoning about logical formulae that can contain reference to partial terms.

1.3 Contributions

This work presents an investigation into the applicability of mechanised proof support for LPF. Over the years there has been a lack of direct proof support for LPF. This work is aimed at addressing this. Related work will be discussed in the appropriate places in the main body of this thesis.

This work focuses on investigating the basic but fundamental two-valued classical logic proof procedure: resolution and the associated technique of proof by contradiction [Rob65, BA01]. These basic fundamental proof techniques are the basis on which advanced proof techniques such as paramodulation [RW69] and superposition [BG94] are built. Thus investigating these basic proof techniques is the essential and obvious starting point for addressing the development of proof support for LPF. An investigation into the issues that arise in applying these basic techniques to LPF, and an investigation into the extent of the modifications needed to be made to these basic proof techniques for LPF is undertaken. This provides key insights into providing mechanised proof support for a non-classical logic like LPF, for instance into the amount of extra work that arises in a mechanisation of such techniques for LPF. This work provides the essential foundation on which to facilitate research into the modification of advanced proof techniques for LPF, and for providing tool support in the future.

Semantic definitions of LPF are defined. These semantic definitions provide the underlying basis of this work. The semantic definitions precisely and succinctly capture how LPF copes with logical formulae that can contain references to partial terms. A semantic definition allows for concepts required for the investigation of proof techniques to be presented unambiguously, and for issues that arise due to partial terms when applying these techniques to LPF to be illustrated precisely. Proofs of modifications required to carry the proof techniques over to LPF are also proved with respect to a semantic definition. It is also shown how interactive tool support for LPF in addition to the work on modifying resolution and the associated technique of proof by contradiction can be developed from semantic definitions for LPF.

Earlier publications (e.g. those cited at the end of Section 1.1.4) have discussed the use of LPF for reasoning about logical formulae that can contain references to partial terms. This thesis argues for the use of LPF for such a purpose, but instead of just presenting informal comparisons this thesis also provides formal comparisons based upon formal semantic definitions, providing a clear divide between this and earlier work. A wide range of approaches are compared in this thesis. A semantic definition of LPF is modified to formally capture the semantics of other approaches to coping with partial terms. This is used to facilitate comparisons and to undertake a vital task of identifying ways of being able to move theorems between the different approaches. The comparisons alongside the mechanisation work for LPF greatly aid in justifying the use of LPF for reasoning about logical formulae that can contain reference to partial terms.

1.4 Structure of the Thesis

Chapter 2 provides an overview of different approaches to coping with logical formulae that can contain references to partial terms. Justifications for the use of LPF, and prior work on mechanising LPF are then discussed in this chapter.

Chapter 3 formally captures the semantics of LPF with both Structural Operational Semantics (SOS) definitions and denotational semantic (DS) definitions being defined.

One of the purposes of the DS definitions is to provide a means to undertake formal comparisons of the different approaches to coping with logical formulae that can contain references to partial terms. An LPF DS definition is modified to formally capture the semantics of different approaches, and these definitions are presented in Chapter 4. This is followed by the comparisons that are made between the different approaches based upon these definitions, and the identification of relationships to allow for theorems to be moved between the different approaches. A DS definition for LPF is also the underlying basis with which to conduct proofs in Chapter 6.

The focus of Chapter 5 is on illustrating how SOS definitions that formally capture the semantics of LPF can give rise to mechanisations of LPF, in both a term-rewriting system and in a theorem prover.

Chapter 6 defines the concepts of satisfiability and validity, as well as related definitions in LPF. An investigation into the applicability of mechanised proof support for LPF is then presented, focusing on the resolution proof procedure. The issues that arise when applying it in LPF are highlighted with illustrative examples, followed by an investigation into determining how to modify it to cover LPF and into the extent of the modifications needed, which are presented alongside supplementary proofs.

Chapter 7 contains a summary of this thesis, and future work of interest is discussed.

Full semantic definitions from Chapter 3 are presented in Appendix A. This includes full abstract syntax definitions, full context conditions as well as the full SOS definitions, and the full DS definitions.

The notation used throughout this thesis in numerous examples is based on the mathematical VDM-SL notation. Appendix B presents the notation of selected VDM-SL data types and their associated operations etc. for reference. Any reader who is not familiar with the mathematical VDM-SL notation is advised to refer to this appendix first.

A glossary consisting of definitions of many terms used widely throughout this thesis is presented in Appendix C.

Chapter 2

Background

Contents

2.1 Categorising the Different Approaches to Coping with Partial Terms 16

2.2 Approaches to Coping with Partial Terms 18

2.2.1 Reformulating Expressions and Function Definitions 18

2.2.2 Classical Approaches 20

2.2.3 Semi-Classical Approaches 25

2.2.4 Non-Classical Logic Approaches 30

2.2.5 Sequent Interpretations 42

2.3 Summary of the Justifications for LPF 44

2.4 Previous Attempts at Mechanisation Support for LPF 49

2.4.1 The VDM Toolset and the Overture Toolset 49

2.4.2 mural 50

2.4.3 Utilising Existing Theorem Provers 51

2.5 Conclusions 52

Partial functions arise frequently in computer science, for instance in program specifications. The application of partial functions can give rise to partial terms, that is, terms that fail to denote a proper value. The presence of partial functions leads to complications when reasoning about logical formulae that can include references to partial terms. Reasoning about such logical formulae is needed for instance, when discharging proofs about properties of program specifications that are expected to hold. As illustrated in the previous chapter, two-valued classical logic cannot directly cope with undefined truth values (truth value "gaps", that is, the absence of a defined truth value true or false), as the two-valued classical logic truth tables are only defined for proper Boolean values.

There is a history of research that has gone into logics that can cope with partial terms. Numerous approaches to reasoning about logical formulae that can contain references to partial terms have been proposed over the years. The questions that must be addressed by approaches to coping with terms partial terms are: what meaning is to be given to the term $f(i)$ when the value i is outside of the defined domain of a partial function f, how undefinedness is to propagate through the various language constructs, and how can reasoning about logical formulae that can contain references to partial terms be conducted formally.

However, there is as yet no consensus on which is the best way of reasoning about logical formulae that can contain references to partial terms. This chapter surveys a number of different approaches to reasoning about such logical formulae. Chapter 4 then goes further by presenting a formal comparison between the approaches.

The different approaches can be classified into categories; these are outlined in Section 2.1. The different approaches are then considered in more detail in Section 2.2. Justifications for LPF then follow in Section 2.3. A discussion on prior mechanisation work that has been undertaken for LPF is presented in Section 2.4.

2.1 Categorising the Different Approaches to Coping with Partial Terms

The different approaches to coping with partial terms can be classified into two categories. The first category comprises of those approaches that attempt to continue using *two-valued classical logic*, and the second category comprises of those approaches that accept the need for a *non-classical logic*.

The approaches in the first category preserve the two-valued classical logic operators. However, the approaches in the second category essentially give up on two-valued classical logic in favour of the use of a non-classical logic. In a non-classical logic, proof rules that are sound in two-valued classical logic are no longer sound and thus they need modifying with definedness conditions, and additional (non-classical) proof rules may be needed for completeness. These complications can affect mechanised proof procedures, and these modifications and extensions can lead to a non-classical logic being too unfamiliar for a user, and thus difficult not only to learn but to use in practice.

The approaches to handling partial terms can be further categorised by describing where they attempt to cope with undefinedness, so what language constructs is undefinedness allowed to propagate through and where are the attempts made to catch undefinedness. It is this categorisation that will be used to structure Section 2.2:

1. Force the reformulation of formulae and/or function definitions to avoid

the introduction of partial terms;

2. Force all terms to denote;

3. Allow terms to fail to denote proper values, but still guard the logical operators from the resulting undefinedness by catching undefinedness at the predicate level; and

4. Adopt non-classical logics.

Category 1 includes those approaches that force the reformulation of any expressions containing references to partial functions, and those approaches that force the reformulation of the domain of partial functions to make partial functions total (i.e. restricting the types of arguments). See Section 2.2.1.

Category 2 attempts to deal with undefinedness by forcing all terms to denote proper values, for instance, through under/over-specifying partial functions on arguments from outside of their defined domain. Approaches where the consideration of partial terms are eliminated from validity proofs by forcing extra well-definedness (WD) conditions to be proved are also considered in this category. See Section 2.2.2.

Category 3 attempts to deal with undefinedness by allowing terms such as $\mathit{zero}(-1)$ to fail to denote proper values, but to force predicates such as $\mathit{zero}(-1) = 0$ to denote, even when their operands are terms that fail to denote proper values. This approach is referred to as the "semi-classical" approach. See Section 2.2.3.

Category 4 allows undefinedness from partial terms to propagate up to the logical operators. Approaches in this category are the non-classical (non-standard) logics, since the interpretations that are given to the logical operators are re-interpreted as undefinedness is incorporated into the logic itself, while the approaches in the three other categories attempt to catch undefinedness before it collides with the logical operators. The logics that are considered in this chapter are commonly referred to as three-valued logics; undefinedness is lifted to formulae by extending the truth values formulae can denote from $\{\mathbf{true}, \mathbf{false}\}$ to $\{\mathbf{true}, \mathbf{false}, \perp_{\mathbb{B}}\}$. Here the logics are not considered to have an explicit undefined value, it is just regarded as a "gap", that is, an absence of a (defined) value [Bla80, Fit07], (there is not an additional truth value, it is just regarded as the absence of a truth value). See Section 2.2.4.

Two-valued classical logic is bivalent, that is, there are two truth values, and every proposition has a truth value that is either true or false. The non-classical (three-valued) logics that are considered here are trivalent, that is, they can be either true, false, or undefined (but recall again there is no concrete undefined value, undefinedness is just treated here as a “gap”).

The major benefit of approaches in both Category 1 and Category 2 is that two-valued classical logic can be still used. Additionally, the approaches in Category 3 preserve the use of the two-valued classical logic operators. The approaches in Category 4 use non-classical logics in favour of two-valued classical logic.

For an overview of many valued logics refer to [Got05]. Four-valued logics also exist (see [Got05] for an overview), but for reasoning about logical formulae that can contain references to partial terms, a three-valued logical treatment suffices.

2.2 Approaches to Coping with Partial Terms

Each of the following four subsections correspond to one of the four categories for coping with partial terms outlined in Section 2.1. Numerous different approaches are discussed for each category. The final subsection in this section outlines different interpretations that can be given to a sequent in a non-classical (three-valued) logic approach.

2.2.1 Reformulating Expressions and Function Definitions

Relations

This approach forces reasoning about partial functions in terms of the corresponding graph of the functions. The graph of an n-ary function is an $(n+1)$-ary relation. So a partial function $f\colon \mathbb{Z} \to \mathbb{Z}$ is to be viewed as a relation $\mathbb{Z}\times\mathbb{Z}$. So, instead of writing a function application in the style of $f(x) = y$ it is to be written as $(x, y) \in f$, that is, is (x, y) a member of the graph of f. The key idea is that $(x, y) \in f$ is false when $x \notin \mathbf{dom}\, f$, for all y [Far90, CJ91, Jon06].

Reasoning about partial functions in this way forces formulae to be written in a non-standard way, and can lead to verbose definitions. As an example Property 1.3 becomes:

$$\forall i\colon \mathbb{Z} \cdot (i, 0) \in \mathit{zero} \vee (-i, 0) \in \mathit{zero}$$

when written in terms of the membership of the graph of the *zero* function. When there is no explicit result expression it is necessary to use existential quantifiers [Jon06].

Restricting the Bounds on Quantifiers

One solution is to restrict quantification to over sets that do not contain any values from outside of the defined domains of any partial functions used. The *zero* function presented earlier is always defined (denotes a value) when applied with a positive number as an argument, however, when applied with a negative number as an argument this function will not denote a proper value.

The key property of this function could thus be expressed as:

$$\forall i{:}\, \mathbb{N} \cdot i \geq 0 \;\Rightarrow\; \mathit{zero}(i) = 0 \tag{2.1}$$

where this property now avoids undefinedness and is therefore true in two-valued classical logic, due to the use of $\mathbb{N}$.

However, restricting the bounds on quantifiers is not always as straightforward. In order to illustrate this consider the following *subp* function presented in [Jon90, CJ91]:

$$\mathit{subp} : \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}$$

$$\mathit{subp}(i,j) \quad \triangleq \quad \textbf{if } i = j \textbf{ then } 0 \textbf{ else } \mathit{subp}(i, j+1) + 1$$

This function is designed to compute $i - j$, but when $i < j$ the function will fail to denote. For example, $\mathit{subp}(5,3)$ results in the value 2, but $\mathit{subp}(3,5)$ is undefined.

The key property of the *subp* function is:

$$\forall i, j{:}\, \mathbb{Z} \cdot i \geq j \;\Rightarrow\; \mathit{subp}(i,j) = i - j \tag{2.2}$$

When i has the value 5 and j has the value 3, there is no problem:

$$\begin{aligned} 5 \geq 3 \;\Rightarrow\; \mathit{subp}(5,3) &= 5 - 3 \\ \textbf{true} \;\Rightarrow\; 2 &= 2 \\ \textbf{true} \;\Rightarrow\; &\textbf{true} \\ &\textbf{true} \end{aligned}$$

However, in the case where i has the value 3 and j has the value 5:

$$\begin{aligned} 3 \geq 5 \;\Rightarrow\; \mathit{subp}(3,5) &= 3 - 5 \\ \textbf{false} \;\Rightarrow\; \bot_{\mathbb{Z}} &= -2 \\ \textbf{false} \;\Rightarrow\; &\bot_{\mathbb{B}} \\ &\bot_{\mathbb{B}} \end{aligned}$$

the formula makes no sense in two-valued classical logic.

Here, the defined domain is a partial relation based upon a relationship between the variables i and j (the function is only defined when $i \geq j$) and thus a set needs to be defined that takes into account this relationship between the two variables i and j (the defined domain of the *subp* function):

$$\{(i,j) \mid i\colon\mathbb{Z},\ j\colon\mathbb{Z} \wedge i \geq j\}$$

Thus Property 2.2 needs reformulating to:

$$\forall i,j\colon \{(i,j) \mid i\colon\mathbb{Z},\ j\colon\mathbb{Z} \wedge i \geq j\} \cdot \mathit{subp}(i,j) = i - j$$

to be able to guard against undefinedness arising from the application of the *subp* function. A similar related approach is considered in the next section.

Re-defining the Domain of Partial Functions

The aim of this approach is to re-define the domain of partial functions to turn them into total functions, by restricting the types of arguments. This approach is similar to the approach from the previous section in that it uses restricted sets which include only those values that lie within the defined domain of a partial function

For instance, the *zero* partial functions domain changes from $\mathbb{Z} \rightarrow \mathbb{Z}$ to $\mathbb{N} \rightarrow \mathbb{N}$, to make the function total. Additionally, the type of the second argument to the integer division operator could be restricted to arguments that belong to the subtype of non-zero integers, so integer division would be defined as $\mathbb{Z} \times \mathbb{Z}_1 \rightarrow \mathbb{Z}$, where $\mathbb{Z}_1$ is $\mathbb{Z} \setminus \{0\}$, that is, $\mathbb{Z}_1 = \{i \mid i\colon\mathbb{Z} \wedge i \neq 0\}$. One must ensure by the restricted sets that functions are never applied outside of their defined domains.

A difficulty can arise as operators such as subtraction etc. could be defined with the domain $\mathbb{Z} \times \mathbb{Z} \rightarrow \mathbb{Z}$, so $\mathit{zero}(10 - 20)$ and $0/(10 - 20)$ can both pose further issues in this approach. Also unfortunately, with this approach any set can be a type and in general the type system becomes undecidable [GS95, CJ91].

2.2.2 Classical Approaches

The first two approaches in this section force the application of functions and operators to denote an element of their range when applied with any arguments from their domain. Functions are forced to denote a value even when they are applied with arguments from outside of their defined domains. In other words a value is assigned to the application of $f(x)$, even in the cases when x is not in the defined domain of the partial function f.

Two contrasting approaches are to underspecify partial functions or to overspecify partial functions. These two approaches are both considered below. The two approaches each remain within two-valued classical logic. The Well-Definedness approach which also maintains the use of two-valued classical logic is then discussed.

Underspecification

This approach centers around forcing every partial function to yield a definite but unspecified (indeterminate/unknown) value from its range when applied with arguments from outside of their defined domain [Far90, GS95, FFL97, Jon06]. In this approach it is generally regarded that it should be impossible to determine (and to prove) which value is returned from a partial function when it is applied with arguments from outside of its defined domain.

For instance, the *zero* function is undefined when its integer argument is less than zero. So when applied with such an argument which is outside of its defined domain the *zero* function should return some definite but unspecified integer value. This forces the *zero* function to become total. In this approach $\mathit{zero}(-1)$ is a defined expression, but its value is left unspecified. Thus:

$$\mathit{zero}(-1) = 0 \lor \mathit{zero}(-1) \neq 0$$

is true if the partial *zero* function returns a definite but yet unknown integer value for arguments from outside of its defined domain ($i < 0$).

Since a partial function is underspecified (thus modelling it as a total function), the underspecification approach carries a major benefit which is that two-valued classical logic can still be used. In particular as illustrated the law of the excluded middle still holds.

Expressions such as the one just presented and an expression such as:

$$\mathit{zero}(-1) = \mathit{zero}(-1)$$

and:

$$x/x = x/x$$

are true. This allows assumptions to be made about partial terms. Note that in some common programming languages such expressions will typically lead to a runtime error.

However, in all cases the truth of a formula cannot be known, for example:

$$zero(-1) = 0$$

and:

$$5/0 = zero(-1)$$

are both defined, but it cannot be determined whether either of them is true or false. Again in common programming languages such expressions will typically lead to a runtime error.

Furthermore, there will be questions over whether:

$$zero(-1) = zero(-2)$$

is true.

A slight alternative to the underspecification approach that returns a definite but yet unknown/unspecified value is to return an arbitrary but still unknown value. So, $zero(-1)$ denotes some arbitrary unknown value. If an arbitrary value is returned instead of a definite value, but such a value is still an unknown value then:

$$zero(-1) = zero(-1)$$

can be true or false, and the law of the excluded middle cannot be assumed to hold (again it can be either true or false).

The paper [Jon95] puts forward a counter example that hints towards the underspecification approach being problematic in a specification language if recursive function definitions somehow overspecify. The counter example follows by defining single element types, for instance, define the type that consists of just the one integer value: $S = \{i \mid i : \mathbb{Z} \wedge i = 0\}$, then taking the head element of an empty sequence s of type S, it should be possible to conclude that: $\mathbf{hd}\, s = 0$, and similarly for the *zero* function where the range of the function is of type S, it should be possible to conclude that: $zero(-1) = 0$. Thus it has been possible to get an unintended overspecification.

This underspecification approach poses a problem when considering the fixed point rule. The authors in [SB99] show that if the fixed point rule is included in Gries's and Schneider's approach [GS95] (underspecified functions instead of partial functions) an inconsistency can result. The interpretation of *zero* denoting the fixed point is lost, since the term $zero(-1)$ denotes an unspecified value, an extension of the graph of *zero* is needed [Jon06].

When considering underspecified functions as opposed to partial functions, then a formula that refers to an underspecified function generally has to be put into the consequent of an implication, with an antecedent describing the defined domain [GS95, SB99]. Whenever, the antecedent is false, the implication is true; the consequent is some unknown (but defined) value.

Overspecification

The overspecification approach is similar to the underspecification approach, except that the result that should be returned by a partial function when applied with arguments from outside of its defined domain, is a known/specific value. For instance, the *zero* function when applied with an argument from outside of its defined domain yields a specific known integer, for example 0.

This approach also has the unfortunate consequence that it can give rise to theorems like:

$$0/0 = 0$$

and:

$$\mathit{zero}(-1) = 0$$

A very similar approach is to make any partial function a total function by adding an error value to the range of a function. Thus the *zero* function could be defined as: $\mathit{zero}\colon \mathbb{Z} \times \mathbb{Z}_\perp$, where $\perp$ is used here to denote the error value being used and $\mathbb{Z}_\perp = \mathbb{Z} \cup \{\perp\}$, and the application of the *zero* function for any argument from outside of its defined domain would yield this error value. For example, $\mathit{zero}(-1) = \perp$.

Because of non-termination all partial functions cannot be transformed into total functions [Che86]. Since in general, we cannot determine if a computation will terminate on some arbitrary input, assigning an arbitrary value to a partial application is uncomputable [Che86].

In HOL [GM93] and in Isabelle/HOL [NWP02] all functions are total. The idea of reasoning about partial functions in such environments follows using similar approaches. For instance, in Isabelle/HOL the expression 1 *div* 0 is 0.

The Well-Definedness Approach

One approach to coping with partial terms that can arise is known as the Well-Definedness (WD) [Meh08, DMR08] approach. The WD approach forces WD conditions to be proven, and as a result undefinedness need not be considered in validity proofs, (that is what the WD conditions are for). If $WD(e)$ is proved then e is guaranteed to contain no term or predicate that fails to denote. Any question over definedness is removed from the concern of validity. Well-definedness and validity are proved separately, and both need to be shown to hold. So, to prove $\Gamma \vdash e$, both validity:

$$\Gamma \vdash e$$

and well-definedness:

$$\vdash WD(\Gamma \vdash e)$$

must be shown to hold. When proving the former it can be assumed that the sequent is well-defined. The expression $WD(e)$ is true if e is not undefined. Showing that both hold is enough to ensure that undefined properties cannot be proved.

Thus for example the expression:

$$zero(-1) = 0$$

is rejected from this approach to coping with partial terms, as it cannot be shown to be well-defined.

The main benefit of this approach is that all reasoning, both of the WD conditions and of the validity proofs, can be done within two-valued classical logic. However, the WD conditions generated can be complicated and they can expand exponentially in size, and thus cause a significant time overhead, and well-definedness is undecidable and therefore needs proving [Meh08].

An alternative is to conduct all reasoning in a three-valued logic. This is considered in Section 2.2.4 and the WD approach is compared to this approach in that section.

PVS has an expressive specification language which is based on classical higher-order logic with a type system that includes predicate subtypes [ORS92, COR$^+$95]. PVS is based on a logic of total functions, but partial functions can be modelled as total functions where the domain is a predicate subtype. In other words total over a restricted domain. For instance, the second argument to the division function must belong to the type $\mathbb{R} \setminus \{0\}$. In PVS type checking is undecidable. Type Correctness Conditions (TCCs) are constructed and need to be proven. If the TCCs are proven then all of the corresponding terms, predicates and formulae are defined. However, as pointed out in [BBS$^+$05] it is possible for a formula $p \Rightarrow q$ to have a valid TCC whose contrapositive $\neg q$

$\Rightarrow\ \neg\, p$ has as invalid TCC.

The approach taken in CVC Lite [BBS$^+$05] is also to remove undefinedness from validity proofs. They start from a three-valued semantics but reduce the problem of checking for validity to checking two formulae in a two-valued logic. A TCC formula is constructed whose validity asserts that the original formula is defined. If the TCC is shown to be valid, then the formula itself is checked for validity. The defined domain of all partial functions needs to be stated.

The authors in [SB99] take a different approach. They have a type *boolean* that has the values true and false and a type *extboolean* that has the values true, false and $\perp$. In this approach a logical operator such as $\vee$ takes two extbooleans and returns a boolean. Two symmetric models of evaluation are defined. In one model $\perp$ is interpreted as being true and in the other model $\perp$ is interpreted as being false. Here a formula is to be regarded as valid if it evaluates to true in both symmetric models of evaluation. In this approach $\perp \vee (\neg \perp)$ holds as in both interpretations for $\perp$ it follows that $\perp \vee (\neg \perp)$ is true.

2.2.3 Semi-Classical Approaches

The semi-classical approaches here also maintain the use of the two-valued classical logic logical operators. Partial functions can still give rise to partial terms, but here the approaches force predicates to yield a defined value of true or false, even if the arguments applied to predicates fail to denote. The idea behind these approaches is presented in [Rus05].

A Partial First-Order Logic (PFOL) is presented in [Far96], which is a variant of first-order logic and, which supports the so called traditional approach to coping with partial functions. It is argued that this approach is "commonly used in mathematics" and "is taught to American students in high school" [Far96]. This approach is supported in the Interactive Mathematical Proof System (IMPS). This approach is also taken in [Far90] and stays close to two-valued classical logic.

In this approach [Far96]:

- Variables and constants always denote;
- Functions may be partial, so $zero(-1)$ is not assigned a value, and a function application is undefined if any of its arguments is undefined; and
- Formulae are always defined since predicates always denote. In this approach a predicate is considered to be false if a term within it (an

operand) does not denote, thus $zero(-1) = 0$ is false since an operand, that is $zero(-1)$, does not denote.

This approach, however, does not fit with the view taken by the author here that formulae such as $zero(-1) = 0$ should not be propositions at all, because the view taken is that no assumptions should be made about partial terms.

As previously mentioned the current notion of equality (which is referred to as weak equality) is undefined if either of its operands are undefined. The weak equality relational operator is illustrated in the truth table in Figure 2.1 when its arguments are integer values.

$=$	-1	0	1	...	$\perp_{\mathbb{Z}}$
-1	**true**	**false**	**false**	...	$\perp_{\mathbb{B}}$
0	**false**	**true**	**false**	...	$\perp_{\mathbb{B}}$
1	**false**	**false**	**true**	...	$\perp_{\mathbb{B}}$
...	...	...	...	...	...
$\perp_{\mathbb{Z}}$	$\perp_{\mathbb{B}}$	$\perp_{\mathbb{B}}$	$\perp_{\mathbb{B}}$	...	$\perp_{\mathbb{B}}$

Figure 2.1: The truth table for weak equality

In evaluating Property 2.2 a partial term can arise from the application of the partial function *subp* which is an operand to the weak equality relational operator. The other operand is defined. The result of using weak equality here is that the partial term propagates outwards making the equality relational operator (predicate) undefined, and thus giving rise to an undefined truth value, and leaving a formula that makes no sense in two-valued classical logic.

The equality relational operator (a predicate) must denote a value even if its operands fail to denote proper values. For instance the use of weak equality can be replaced with such a non-strict equality. Consider first existential equality which is illustrated in the truth table in Figure 2.2 when its arguments are integer values.

$=_{\exists}$	-1	0	1	...	$\perp_{\mathbb{Z}}$
-1	**true**	**false**	**false**	...	**false**
0	**false**	**true**	**false**	...	**false**
1	**false**	**false**	**true**	...	**false**
...	...	...	...	...	...
$\perp_{\mathbb{Z}}$	**false**	**false**	**false**	...	**false**

Figure 2.2: The truth table for existential equality

As can be seen from the truth table this equality is designed to denote false if either of its operands is undefined. Consider the modified version of the key *subp* property:

$$\forall i, j : \mathbb{Z} \cdot i \geq j \;\Rightarrow\; \mathit{subp}(i, j) =_{\exists} i - j \tag{2.3}$$

In the case when $i \geq j$ (the defined case) the evaluation using this equality is no different to what has already been presented (with weak equality), what changes is the evaluation in the undefined case. In the case when i has the value 3 and j has the value 5, the evaluation would now be as follows:

$$\begin{aligned} 3 \geq 5 \;\Rightarrow\; \mathit{subp}(3, 5) =_{\exists} 3 - 5 \\ \mathbf{false} \;\Rightarrow\; \perp_{\mathbb{Z}} =_{\exists} -2 \\ \mathbf{false} \;\Rightarrow\; \mathbf{false} \\ \mathbf{true} \end{aligned}$$

The implication logical operator has been guarded from the “gap”. The use of the non-strict equality (existential equality here) has prevented the “gap” in the term from propagating up past the existential equality to the logical operator. The existential equality predicate denotes a Boolean value, when any of its operands are undefined.

A problem with using a non-strict equality is that defined values are returned even in cases when at least the one operand is undefined. Thus in this approach the weak notion of equality still needs to be written in function definitions, with a non-strict notion of equality being needed to cope with partial terms in logical formulae. Thus, a user has to be aware of multiple notions of equality when reasoning about logical formulae that can contain reference to partial terms using such an approach.

In the examples, weak equality has been replaced with a non-strict equality, but the relational operator $\geq$ has been kept, because i and j are bound to integers and so $i \geq j$ is always defined. However, while this discussion has focused on equality all predicates that are subject to undefined operands will need modifying in a similar way. Thus:

$$\mathit{zero}(-1) >_{\exists} \mathit{zero}(-2)$$

and:

$$\mathit{zero}(-1) \leq_{\exists} \mathit{zero}(-2)$$

are both false.

However, the expression:

$$zero(-1) =_{\exists} zero(-1)$$

is false, and the expression:

$$zero(-1) \neq_{\exists} zero(-1)$$

is false, but the expression:

$$\neg\,(zero(-1) =_{\exists} zero(-1))$$

is true. It is the case that $\perp_{\mathbb{Z}} =_{\exists} \perp_{\mathbb{Z}}$ is false of whose negation is true, while $\perp_{\mathbb{Z}} \neq_{\exists} \perp_{\mathbb{Z}}$ is false. Thus $\perp_{\mathbb{Z}} \neq_{\exists} \perp_{\mathbb{Z}}$ is not logically equivalent to $\neg\,(\perp_{\mathbb{Z}} =_{\exists} \perp_{\mathbb{Z}})$.

Strong Equality

Another non-strict equality that could have been used instead of existential equality is strong equality. The truth table for strong equality is presented in Figure 2.3, when its arguments are integer values.

$==$	-1	0	1	...	$\perp_{\mathbb{Z}}$
-1	**true**	**false**	**false**	...	**false**
0	**false**	**true**	**false**	...	**false**
1	**false**	**false**	**true**	...	**false**
...	...	...	...	...	...
$\perp_{\mathbb{Z}}$	**false**	**false**	**false**	...	**true**

Figure 2.3: The truth table for strong equality

Note that this equality differs from its existential counterpart only by the fact that $\perp_{\mathbb{Z}} == \perp_{\mathbb{Z}}$ is true instead of false. The modified key *subp* property is presented in Property 2.4, but the evaluation is the same as already presented for existential equality and so will not be presented again.

$$\forall i, j\colon \mathbb{Z} \cdot i \geq j \;\Rightarrow\; subp(i, j) == i - j \tag{2.4}$$

So, the expression:

$$zero(-1) == 0$$

evaluates to false, and the expression

$$zero(-1) == zero(-1)$$

evaluates to true.

However, it is less clear why the expression:

$$zero(-1) == zero(-2)$$

should evaluate to true.

Furthermore, while the expression:

$$zero(-1) \neq\neq zero(-1)$$

evaluates to true, the expression:

$$\neg\,(zero(-1) == zero(-1))$$

evaluates to false. Thus $a \neq\neq b$ is not logically equivalent to $\neg\,(a == b)$. Again all predicates that are subject to undefined operands will need modifying in a similar way to what has been presented for strong equality.

The Logic of Computable Functions

Edinburgh Logic of Computable Functions (LCF) [GMW79] is a system for doing proofs interactively. This is based on domain theory [GMW79, Age94].

Terms can fail to denote. Terms include computable functions, and an undefined value is needed since computations may never terminate. Since the computation may not terminate an undefined value UU is assumed for such terms. Formulae are designed to be two-valued. The equality operator $==$ can be used in LCF to build up formulae from terms. Recall that this notion of equality is non-strict.

Quantifiers in LCF can range over undefined values. In LCF a variable may be undefined. For instance, the domain of integers would include the element UU. This has the undesired consequence that frequent reasoning about undefinedness can be needed, (consider using the natural number induction rule, where $P(UU)$ will need to be shown to hold).

Predicate Underspecification

This approach is similar to the underspecification approach discussed earlier for terms, but in this approach terms can be undefined whereas predicates are forced to be defined. However, this approach differs from the approaches discussed above where for example the truth value false is taken by a predicate when any of its operands are undefined.

In this approach when a predicate contains an undefined operand the predicate is defined, but in all cases it is not known whether the predicate is true or false. In [Spi92] they state that predicates are *undetermined*, it is not known whether they are true or false. The predicates do not have some intermediate status in which they are 'neither true nor false'; it is just not said whether they are true or not [Spi92]. It is this approach that has been used in the Z notation [Spi92].

Both:

$$1/0 = 1/0$$

and:

$$1/0 = 0 \vee \neg\,(1/0 = 0)$$

can both be proven to be true. However:

$$1/0 = 0$$

and:

$$1/0 = 2/0$$

are both known to be defined. It is, however, unknown whether the last two examples are true or false.

2.2.4 Non-Classical Logic Approaches

The approaches considered from here onwards make the case for the use of a non-classical (three-valued) logic to reason about logical formulae that can contain references to partial terms. The approaches considered above have the aim of eliminating the problem of undefinedness before the undefinedness propagates upwards and collides with the logical operators. In the non-classical logic approaches the logical operators are extended to cope with undefinedness.

Łukasiewicz in the 1920s presented the first three-valued logic [Che86]. Differences between that logic and other three-valued logics comes down to how undefinedness is interpreted.

$\vee_W$	**true**	$\bot_{\mathbb{B}}$	**false**
true	**true**	$\bot_{\mathbb{B}}$	**true**
$\bot_{\mathbb{B}}$	$\bot_{\mathbb{B}}$	$\bot_{\mathbb{B}}$	$\bot_{\mathbb{B}}$
false	**true**	$\bot_{\mathbb{B}}$	**false**

$\wedge_W$	**true**	$\bot_{\mathbb{B}}$	**false**
true	**true**	$\bot_{\mathbb{B}}$	**false**
$\bot_{\mathbb{B}}$	$\bot_{\mathbb{B}}$	$\bot_{\mathbb{B}}$	$\bot_{\mathbb{B}}$
false	**false**	$\bot_{\mathbb{B}}$	**false**

$\Rightarrow_W$	**true**	$\bot_{\mathbb{B}}$	**false**
true	**true**	$\bot_{\mathbb{B}}$	**false**
$\bot_{\mathbb{B}}$	$\bot_{\mathbb{B}}$	$\bot_{\mathbb{B}}$	$\bot_{\mathbb{B}}$
false	**true**	$\bot_{\mathbb{B}}$	**true**

$\neg_W$	
true	**false**
$\bot_{\mathbb{B}}$	$\bot_{\mathbb{B}}$
false	**true**

Figure 2.4: The weak Kleene truth tables for disjunction, conjunction, implication and negation

Weak Kleene (Bochvar Internal)

Weak Kleene [Kle52, Sid10] is a three-valued logic, where formulae can be undefined. The weak Kleene approach takes the viewpoint that if any formula has a part of it that is undefined then the entire formula is to be regarded as undefined. The truth values of all operands to a logical operator must be available (that is, defined) for a defined result to be returned, otherwise the result is undefined.

Here all of the logical operators are given a strict interpretation. When all operands of a logical operator are defined, that is, denote a truth value true or false, then the meaning coincides with the two-valued classical logic interpretations of the logical operators. The truth tables for this approach are presented in Figure 2.4. The weak Kleene truth tables are the same truth tables as in Bochvar's internal three-valued logic [Boc81].

Quantifiers in this approach are given a strict interpretation, and this is defined as a case analysis:

$$\forall i\colon D \cdot p(i) \triangleq \{ \begin{array}{l} \textbf{true} \text{ - } p(i) \text{ is true for all } i \in D \\ \textbf{false} \text{ - } * \\ \bot_{\mathbb{B}} \text{ - } p(i) \text{ is } \bot_{\mathbb{B}} \text{ for some } i \in D \end{array}$$

$*$ - $p(i)$ is false for at least the one $i \in D$, and $p(i)$ is never undefined for any $i \in D$

The existential quantifier follows in a similar way. Quantifiers will only range over a set of defined values. All variables range only over defined values.

Consider some of the earlier illustrative properties again; first an evaluation of Property 2.2:

$$3 \geq 5 \ \Rightarrow_W \ subp(3,5) = 3 - 5$$

$$\begin{aligned}\mathbf{false}\ \Rightarrow_W \bot_{\mathbb{Z}} = -2 \\ \mathbf{false}\ \Rightarrow_W \bot_{\mathbb{B}} \\ \bot_{\mathbb{B}}\end{aligned}$$

and an evaluation of Property 2.8:

$$\begin{aligned}\mathit{subp}(3,5) = 3 - 5 \vee_W \mathit{subp}(5,3) = 5 - 3 \\ \bot_{\mathbb{Z}} = -2 \vee_W 2 = 2 \\ \bot_{\mathbb{B}} \vee_W \mathbf{true} \\ \bot_{\mathbb{B}}\end{aligned}$$

As illustrated such properties do not hold in this approach. Additionally, the law of the excluded middle and the absorption properties do not hold in this approach.

The logical operators in this approach are strict, that is, if an operand is undefined then the formula is undefined. Any undefined operand ensures that no defined result can be returned ensuring that **true** $\vee \bot_{\mathbb{B}}$ and $\bot_{\mathbb{B}} \vee$ **true** are both undefined. The following non-classical logic approaches utilise non-strict logical operators, that is, that they attempt (in certain circumstances) to return a defined value even in the presence of undefined operand(s).

McCarthy's Conditional Operators

A non-classical interpretation of the logical operators was proposed by McCarthy [McC67]. In this approach the logical operators are defined through the use of conditional expressions, so for instance:

$$p \vee_M q$$

is defined as:

$$\textbf{if } p \textbf{ then true else } q$$

The other logical operators must also be given a conditional definition.

The truth tables for McCarthy's conditional operators are presented in Figure 2.5. The operators are monotone in the ordering presented in Figure 2.6. If a formula denotes a value true or false, then the formula will still denote that value it will not be contradicted, if any undefined term later evaluates to a defined value (e.g. through further evaluation).

When restricted to just the truth values true and false then McCarthy's

$\vee_M$	**true**	$\bot_{\mathbb{B}}$	**false**
true	**true**	**true**	**true**
$\bot_{\mathbb{B}}$	$\bot_{\mathbb{B}}$	$\bot_{\mathbb{B}}$	$\bot_{\mathbb{B}}$
false	**true**	$\bot_{\mathbb{B}}$	**false**

$\wedge_M$	**true**	$\bot_{\mathbb{B}}$	**false**
true	**true**	$\bot_{\mathbb{B}}$	**false**
$\bot_{\mathbb{B}}$	$\bot_{\mathbb{B}}$	$\bot_{\mathbb{B}}$	$\bot_{\mathbb{B}}$
false	**false**	**false**	**false**

$\Rightarrow_M$	**true**	$\bot_{\mathbb{B}}$	**false**
true	**true**	$\bot_{\mathbb{B}}$	**false**
$\bot_{\mathbb{B}}$	$\bot_{\mathbb{B}}$	$\bot_{\mathbb{B}}$	$\bot_{\mathbb{B}}$
false	**true**	**true**	**true**

$\neg_M$	
true	**false**
$\bot_{\mathbb{B}}$	$\bot_{\mathbb{B}}$
false	**true**

Figure 2.5: The McCarthy truth tables for disjunction, conjunction, implication and negation

$$\begin{tikzcd}
\textbf{true} \arrow[dr, dash, "\prec"'] & & \textbf{false} \arrow[dl, dash, "\prec"] \\
 & \bot_{\mathbb{B}} &
\end{tikzcd}$$

Figure 2.6: The ordering on truth values in McCarthy's approach

logical operators coincide with the two-valued classical logic operators. The logical operators have a lazy evaluation in that if a result can be determined from the first operand then that result is returned irrespective of the second operand. The logical operators are given a strict sequential interpretation in McCarthy's approach, and thus this approach can cover up "gaps" in the second operand in certain circumstances.

The key property of the *subp* function but this time with the conditional implication operator is:

$$\forall i, j : \mathbb{Z} \cdot i \geq j \;\Rightarrow_M\; subp(i, j) == i - j \tag{2.5}$$

where an example evaluation follows as:

$$\begin{aligned}
3 \geq 5 \;\Rightarrow_M\; subp(3,5) = 3 - 5 \\
\textbf{false} \;\Rightarrow_M\; \bot_{\mathbb{Z}} = -2 \\
\textbf{false} \;\Rightarrow_M\; \bot_{\mathbb{B}} \\
\textbf{true}
\end{aligned}$$

As shown a formula that made no sense in two-valued classical logic can now be evaluated to true using a conditional interpretation of the implication logical operator.

However, this approach does have some drawbacks. Firstly, in this logic the disjunction logical operator and conjunction logical operator no longer have the commutative property, that is, $p \vee_M q$ being equivalent to $q \vee_M p$. Second the implication contrapositive property, that is, $p \;\Rightarrow_M q$ being equivalent to $\neg q \;\Rightarrow_M \neg p$ no longer holds. Also, there is a problem that if the first variable is undefined then the expression is undefined because the conditional expressions are strict in their first argument; the first variable was referred to as the inevitable variable by McCarthy. However, if the second operand is undefined a defined result could be returned if enough information to decide the result is given through the first operand.

To emphasise the problem of the inevitable variable consider the following property of the *subp* function:

$$\forall i, j{:}\, \mathbb{Z} \cdot \mathit{subp}(i, j) = i - j \vee_M \mathit{subp}(j, i) = j - i \tag{2.6}$$

If i is set to the value 3 and j to the value 5 the evaluation is:

$$\begin{aligned} \mathit{subp}(3,5) = 3 - 5 \vee_M \mathit{subp}(5,3) = 5 - 3 \\ \bot_{\mathbb{Z}} = -2 \vee_M 2 = 2 \\ \bot_{\mathbb{B}} \vee_M \textbf{true} \\ \bot_{\mathbb{B}} \end{aligned}$$

Thus Property 2.6 does not hold in McCarthy's approach. The contrapositive of the key *subp* property also causes a problem with McCarthy's conditional interpretation of the logical operators:

$$\forall i, j{:}\, \mathbb{Z} \cdot \mathit{subp}(i, j) \neq i - j \;\Rightarrow_M i < j \tag{2.7}$$

A problematic evaluation of Property 2.7 follows as:

$$\begin{aligned} \mathit{subp}(3,5) \neq 3 - 5 \;\Rightarrow_M 3 < 5 \\ \bot_{\mathbb{Z}} \neq -2 \;\Rightarrow_M \textbf{true} \\ \bot_{\mathbb{B}} \;\Rightarrow_M \textbf{true} \\ \bot_{\mathbb{B}} \end{aligned}$$

Due to the treatment of "gaps" in this approach, the law of the excluded middle is also lost. Abandoning the law of the excluded middle gives rise to a logic that is weaker than two-valued classical logic.

Defining quantifiers is problematic in this logic. This issue is considered further in Section 4.1. Note that as in the weak Kleene approach (and in LPF) quantifiers will only range over a set of defined values. All variables range only over defined values.

Numerous programming languages contain logical operators which are interpreted in this way. Such a semantics is used in the specification language of the Rigorous Approach to Industrial Software Engineering (RAISE) [GHH$^+$92, GHH$^+$95].

Due to the problems of the lack of the commutative rules etc. there has been a proposal to use both the standard logical operators (for commutativity etc.) and the conditional forms of these logical operators (to cope with "gaps") [CJ91]. However, an extensive amount of non-standard rules are required for this approach to work.

LPF

The LPF approach, like McCarthy's approach considered, handles undefinedness by extending the interpretations of the logical operators. LPF was the logic designed to underlie the Vienna Development Method (VDM) [Jon90]. LPF itself is based upon Strong Kleene logic [Kle38, Kle52].

The truth tables in Figure 2.7 illustrate the way in which the propositional operators in LPF have been extended to handle truth values that may fail to denote proper values. These truth tables provide the strongest possible *monotonic* extension of the familiar two-valued classical logic propositional operators with respect to the ordering on truth values depicted in Figure 2.6 (the ordering is the same as in McCarthy's conditional operator approach). The truth tables can be viewed as describing a parallel lazy evaluation of the operands, whereby a result is delivered as soon as enough information is available, and such a result will not be contradicted if a $\perp_{\mathbb{B}}$ later evaluates to a proper Boolean value.

Thus as in McCarthy's conditional operator approach the logical operators are non-strict. These truth tables are presented by Kleene in [Kle38] and in [Kle52, §64], who in turn attributes them back to Łukasiewicz (discussed in the next section). These connectives were introduced to model non-terminating recursive functions [Kle38]. Notice that as in the other non-classical logics considered in this chapter, a partial term in a formula can be replaced by another partial term, and the meaning of the formula will not change.

It should be clear from the truth tables that familiar properties such as the commutativity of conjunction and disjunction hold in LPF, unlike in McCarthy's approach.

The quantifiers only range over a set D of proper (i.e. defined) values. No

$\vee$	**true**	$\bot_{\mathbb{B}}$	**false**
true	**true**	**true**	**true**
$\bot_{\mathbb{B}}$	**true**	$\bot_{\mathbb{B}}$	$\bot_{\mathbb{B}}$
false	**true**	$\bot_{\mathbb{B}}$	**false**

$\wedge$	**true**	$\bot_{\mathbb{B}}$	**false**
true	**true**	$\bot_{\mathbb{B}}$	**false**
$\bot_{\mathbb{B}}$	$\bot_{\mathbb{B}}$	$\bot_{\mathbb{B}}$	**false**
false	**false**	**false**	**false**

$\Rightarrow$	**true**	$\bot_{\mathbb{B}}$	**false**
true	**true**	$\bot_{\mathbb{B}}$	**false**
$\bot_{\mathbb{B}}$	**true**	$\bot_{\mathbb{B}}$	$\bot_{\mathbb{B}}$
false	**true**	**true**	**true**

$\neg$	
true	**false**
$\bot_{\mathbb{B}}$	$\bot_{\mathbb{B}}$
false	**true**

Figure 2.7: The LPF truth tables for disjunction, conjunction, implication and negation

$\bot$ element/value can be included in D. All variables in LPF range only over defined values, like in the weak Kleene and in McCarthy's conditional operator approach already introduced. For instance, one can quantify over the set of proper integer values $\mathbb{Z}$, but not over the set $\mathbb{Z}_\bot$, that is, $\mathbb{Z} \cup \bot_{\mathbb{Z}}$. The universal quantifier is defined through a case analysis:

$$\forall i \colon D \cdot p(i) \triangleq \{ \begin{array}{l} \textbf{true} \text{ - } p(i) \text{ is } \textbf{true} \text{ for all } i \in D \\ \textbf{false} \text{ - } p(i) \text{ is } \textbf{false} \text{ for one } i \in D \\ \bot_{\mathbb{B}} \text{ - otherwise} \end{array}$$

thus $\forall i \colon D \cdot p(i)$ can be false even if $p(i)$ is undefined for some $i \in D$. The universal quantifier is undefined if $p(i)$ is undefined for all i, or when $p(i)$ is undefined for at least the one i and $p(i)$ is never false. The existential quantifier follows in a similar way.

Since quantifiers range only over proper defined values the quantifiers are a natural extension of the propositional logical operators viewing universal quantification as a conjunction and existential quantification as a disjunction. The two quantifiers (universal and existential) are monotonic.

As with McCarthy's approach, LPF can handle the key *subp* property (Property 2.2), but unlike McCarthy's approach LPF can cope with the contrapositive of this property, where the problematic example from the discussion on McCarthy's conditional operators is now evaluated as:

$$\begin{aligned} \mathit{subp}(3,5) = 3 - 5 &\;\Rightarrow\; 3 < 5 \\ \bot_{\mathbb{Z}} = -2 &\;\Rightarrow\; \textbf{true} \\ \bot_{\mathbb{B}} &\;\Rightarrow\; \textbf{true} \\ &\quad\;\; \textbf{true} \end{aligned}$$

Additionally, LPF can also cope with the following property:

δ	
true	**true**
$\perp_{\mathbb{B}}$	$\perp_{\mathbb{B}}$
false	**true**

Δ	
true	**true**
$\perp_{\mathbb{B}}$	**false**
false	**true**

Figure 2.8: The LPF truth table for the definedness operators δ and Δ

$$\forall i, j \colon \mathbb{Z} \cdot \mathit{subp}(i, j) = i - j \vee \mathit{subp}(j, i) = j - i \tag{2.8}$$

where the problematic example from the discussion on McCarthy's conditional operators is now evaluated as:

$$\begin{aligned} \mathit{subp}(3,5) = 3 - 5 \vee \mathit{subp}(5,3) = 5 - 3 \\ \perp_{\mathbb{Z}} = -2 \vee 2 = 2 \\ \perp_{\mathbb{B}} \vee \textbf{true} \\ \textbf{true} \end{aligned}$$

One issue with this logic is that the, so called, *law of the excluded middle*:

$$\frac{}{p \vee \neg\, p}$$

does not hold because the disjunction of two undefined Boolean values is still undefined: thus $(\mathit{subp}(1,5) = 4) \vee \neg\,(\mathit{subp}(1,5) = 4)$ is not a tautology. The law of the excluded middle only holds for defined p. Additionally, $p \wedge \neg\, p$ being false does not hold, and $p \;\Rightarrow\; p$ also does not hold, due to the presence of the third undefined value.

For expressive completeness, definedness operators δ and Δ are introduced whose truth tables are presented in Figure 2.8. The two definedness operators δ and Δ are used to determine whether a formula is defined.

The δ logical operator is monotone, an undefined operand gives rise to an undefined result. Unlike all of the other logical operators presented, the Δ operator is not monotone, it is total (always returns a defined result) and obviously non-strict. While the Δ logical operator is not monotone it does add expressiveness to the logic.

A *sequent* is a statement about logical expressions that is used to represent the situation when a conclusion can be derived/deduced from a (possibly empty) set of assumptions. In LPF a sequent is interpreted with what is known as the *SS*-interpretation, see Section 2.2.5. Let $\Gamma = \{e_1, \ldots, e_n\}$ be a set of formulae, where the commas are to be interpreted as conjunctions, and let e be a single formula, then the sequent statement in LPF $\Gamma \vdash e$ is valid whenever for all interpretations:

- where Γ is true (that is, each formula in Γ is true) in an interpretation then e is also true in that interpretation;
- where Γ is false (that is, there exists a formula in Γ that is false) in an interpretation; and
- where Γ is undefined in an interpretation.

If Γ is false or undefined in an interpretation then no mention needs to be made of e; that is, in such situations e can be true, false or undefined in that interpretation. A sequent statement is invalid whenever there exists an interpretation where Γ is true, and e is false or undefined in that interpretation. Section 2.2.5 compares strong Kleene logic (the preferred approach, see Section 2.3) with other sequent interpretations.

Due to the sequent interpretation in LPF and due to the loss of the law of the excluded middle, certain rules need modifying in LPF. For instance, the unrestricted deduction theorem:

$$\frac{p \vdash q}{p \;\Rightarrow\; q}$$

does not hold in LPF because p could potentially be undefined. It is the case that $\bot_{\mathbb{B}} \vdash \bot_{\mathbb{B}}$ is valid in LPF, but it is the case that $\bot_{\mathbb{B}} \;\Rightarrow\; \bot_{\mathbb{B}}$ is not valid in LPF (it is undefined), since $p \;\Rightarrow\; p$ is not a tautology in LPF. A modified rule needs to be used that has the added hypothesis that p is defined (p must be shown to be defined in LPF):

$$\boxed{\Rightarrow\text{-}I}\;\frac{\delta p; p \vdash q}{p \;\Rightarrow\; q}$$

Compared to two-valued classical logic, extra axioms are required in LPF due to the loss of the law of the excluded middle to complete the propositional LPF definition; refer to [BFL+94] and [Jon06] for more detail.

Without the Δ operator there would be no tautologous formulae in this approach, due to the presence of "gaps" ($\bot_{\mathbb{B}}$). The Δ operator gives rise to an alternative property which is known as the *law of the excluded fourth*:

$$\frac{}{p \vee \neg\, p \vee \neg\, \Delta p}$$

that is, p is true, false, or undefined. Since the Δ operator is not monotone it rarely appears in normal assertions, and is an operator on the meta-level. To claim definedness in a proof the δ operator is normally used, as was illustrated in the $\Rightarrow$ *-I* rule.

The equality used in this approach is weak. No strong notion of equality is used here.

In the WD approach one has to prove well-definedness and validity separately, to avoid being able to prove undefined formulae valid. This approach does allow the assumption that when proving that a sequent is valid that it is well-defined, as another proof takes on the burden of well-definedness. In LPF validity and well-definedness are proved at the same time.

An untyped version of LPF is introduced in [Che86]. A typed version of LPF is introduced in [JM94].

Similar approaches: In [MJ12] the authors take a similar approach. Their aim is to prevent arithmetic overflows in Alloy. Formulae involving out of bound arithmetic applications are considered to be undefined. The LPF propositional operator semantics are utilised, but a different interpretation is given to the quantifiers. The quantified variables are to range only over values whereby the quantified expression is defined, that is, the quantified expression (the predicate) is true or false. Thus values where the predicate would be undefined are removed, which is determined through the LPF semantics, and as a result quantifiers can never be undefined. Thus quantifiers do not use the three-valued LPF interpretation. Every top-level formulae in an Alloy model is quantified.

Koletsos's three-valued logic [Kol76] uses the same truth tables as LPF for propositional logical operators, with only monotone operators being present. In this approach partial predicates can be present, however, partial functions are not available. Two notions of validity are considered. The first *strong validity*, where given a set of assumptions Γ and a set of formulae ϕ, $(\Gamma \vdash \phi)$ for every interpretation, if Γ is true in an interpretation then at least one formula $\phi_i \in \phi$ is true in that interpretation, and if all formulae in ϕ are false (that is, that each $\phi_i \in \phi$ is false) in an interpretation, then at least the one formula $\gamma_i \in \Gamma$ is false. The second notion of validity is similar to the sequent interpretation of LPF.

Blamey's partial logic [Bla80] again only deals with monotone operators. This approach considers partial functions, and also uses a notion of validity similar to Koletsos's strong validity notion.

The truth tables of the LPF approach were first introduced in [Kle38] and are also presented in [Kle52]. Additionally, it was Koletsos's logic [Kol76]

$\vee_{\text{Ł}}$	**true**	$\perp_{\mathbb{B}}$	**false**
true	**true**	**true**	**true**
$\perp_{\mathbb{B}}$	**true**	$\perp_{\mathbb{B}}$	$\perp_{\mathbb{B}}$
false	**true**	$\perp_{\mathbb{B}}$	**false**

$\wedge_{\text{Ł}}$	**true**	$\perp_{\mathbb{B}}$	**false**
true	**true**	$\perp_{\mathbb{B}}$	**false**
$\perp_{\mathbb{B}}$	$\perp_{\mathbb{B}}$	$\perp_{\mathbb{B}}$	**false**
false	**false**	**false**	**false**

$\Rightarrow_{\text{Ł}}$	**true**	$\perp_{\mathbb{B}}$	**false**
true	**true**	$\perp_{\mathbb{B}}$	**false**
$\perp_{\mathbb{B}}$	**true**	**true**	$\perp_{\mathbb{B}}$
false	**true**	**true**	**true**

$\neg_{\text{Ł}}$	
true	**false**
$\perp_{\mathbb{B}}$	$\perp_{\mathbb{B}}$
false	**true**

Figure 2.9: The Łukasiewicz truth tables for disjunction, conjunction, implication and negation

that inspired LPF [Che86]. Blamey's logic [Bla80] was also considered, but its notion of strong validity was deemed too strong, as the interested is just in being able to deduce truth from truth [Che86]. A combination of Koletsos's logic and Blamey's logic was used in the development of LPF [Che86].

Łukasiewicz

Łukasiewicz presented a three-valued logic in [Łuk20] (also refer to [Pri53, But55, Sid10]). The truth tables for this approach for disjunction, conjunction and negation are the same as in LPF (strong Kleene logic) as presented in Figure 2.7. In Łukasiewicz's approach, however, the implication logical operator differs from that for LPF. The truth tables for Łukasiewicz's logical operators are presented in Figure 2.9. Łukasiewicz's approach was put forward originally as a solution to the problem of propositions of future events [Pri53]. Undefinedness is often regarded as indefinite or possible in this approach, while in LPF/strong Kleene it is regarded as undefined, and it is frequently written as $\frac{1}{2}$ in Łukasiewicz's approach but here it will still be written as $\perp_{\mathbb{B}}$.

A difference to the LPF/strong Kleene approach comes down to the fact that $p \Rightarrow_{\text{Ł}} p$ is valid in Łukasiewicz's logic, since $\perp_{\mathbb{B}} \Rightarrow_{\text{Ł}} \perp_{\mathbb{B}}$ is taken to be true. So, while in the LPF/strong Kleene approach undefinedness cannot be used to infer anything, in Łukasiewicz's approach undefinedness can be used to infer truth, that is, you can infer that you do not know if you really do not know. Additionally, in Łukasiewicz's approach $\perp_{\mathbb{B}} \Leftrightarrow_{\text{Ł}} \perp_{\mathbb{B}}$ is true.

This means that unlike in the LPF/strong Kleene approach the logical operators in Łukasiewicz's are not monotone. Consider $\perp_{\mathbb{B}} \Rightarrow_{\text{Ł}} \perp_{\mathbb{B}}$ as being true, then if the antecedent "completes" (the evaluation of a function "completes") to true and if the consequent "completes" to false (the evaluation of a function "completes"), then the truth value of $\perp_{\mathbb{B}} \Rightarrow_{\text{Ł}} \perp_{\mathbb{B}}$ changes from true to false. This is one of the primary reasons why the LPF approach must be favoured for this work, (the same can be applied as an argument against Bochvar's external

$\vee_B$	**true**	$\bot_{\mathbb{B}}$	**false**
true	**true**	**true**	**true**
$\bot_{\mathbb{B}}$	**true**	**false**	**false**
false	**true**	**false**	**false**

$\wedge_B$	**true**	$\bot_{\mathbb{B}}$	**false**
true	**true**	**false**	**false**
$\bot_{\mathbb{B}}$	**false**	**false**	**false**
false	**false**	**false**	**false**

$\Rightarrow_B$	**true**	$\bot_{\mathbb{B}}$	**false**
true	**true**	**false**	**false**
$\bot_{\mathbb{B}}$	**true**	**true**	**true**
false	**true**	**true**	**true**

$\neg_B$	
true	**false**
$\bot_{\mathbb{B}}$	**true**
false	**true**

Figure 2.10: The Bochvar external truth tables for disjunction, conjunction, implication and negation

approach considered in the next subsection).

In Łukasiewicz's logic one has to give up the well known property that an implication $p \Rightarrow_{Ł} q$ is logically equivalent to $\neg\, p \vee_{Ł} q$, due to the case when both p and q are undefined. In fact $p \vee_{Ł} q$ is logically equivalent to $(p \Rightarrow_{Ł} q) \Rightarrow_{Ł} q$, and thus $\neg_{Ł} p \vee_{Ł} q$ is logically equivalent to $(\neg_{Ł} p \Rightarrow_{Ł} q) \Rightarrow_{Ł} q$. Additionally, $\neg_{Ł}\, p \Rightarrow_{Ł} p$ is not logically equivalent to p. Also like in the other non-classical logic approaches considered above, the law of the excluded middle also does not hold in Łukasiewicz's approach.

Bochvar External

Bochvar's External three-valued logic approach [Boc81] differs from the other non-classical logic approaches in that each logical operator returns true or false, that is, formulae are forced into a two-valued framework; the output of applying a logical operator is always two-valued. The terms nonsense N or meaningless are generally used in Bochvar's approaches as opposed to the term undefined, but in the descriptions that follow $\bot$ will continue to be written.

Even if both operands of a disjunction are undefined then a defined value (false) is returned. The reasoning is that if an operand to the disjunction logical operator is not true, then the disjunction logical operator cannot be true. In this approach undefined is treated as if it is false. The truth tables for this approach are presented in Figure 2.10.

Notice that in this approach the expression $\neg_B(1/0 = 0)$ is taken as being true since the undefined predicate $1/0 = 0$ is regarded as false. Also it is the case that the expression $\bot_{\mathbb{B}} \Rightarrow_B \bot_{\mathbb{B}}$ is true. Again this does not fit with the preferred viewpoint here that no assumptions should be made about partial terms. Furthermore, note that p is not logically equivalent to $\neg_B \neg_B p$.

An advantage of this three-valued logic is that the law of the excluded middle holds. The truth tables for the logical operators return only defined truth values, that is, true and false as results, but at the expense of monotonicity.

$\lceil$	
true	**true**
$\bot_{\mathbb{B}}$	**false**
false	**false**

$\rceil$	
true	**false**
$\bot_{\mathbb{B}}$	**false**
false	**true**

Figure 2.11: The Bochvar external truth tables for the assertion operator $\lceil$ and the denial operator $\rceil$

In Bochvar's internal approach (and the weak Kleene approach) if the Boolean expression p is undefined then any logical operator with p as an operand is undefined. Bochvar, when introducing his external approach [Boc81] introduces an assertion logical operator $\lceil p$ and a denial logical operator $\rceil p$, which are to be read as "p is true" and as "p is false" respectively. The truth tables for the logical operator $\lceil$ and for the logical operator $\rceil$ are presented in Figure 2.11.

Bochvar's external logical operators are never undefined when given undefinedness as input. The external logical operators can be defined through the use of the internal logical operators. For example, $p \vee_B q$ is defined as $\lceil p \vee_W \lceil q$. The internal truth tables and the external truth tables correspond when all operands to any of the logical operators denote.

Property 2.2 and Property 2.8 both hold in this approach. Consider some of the earlier illustrative evaluations again, first of Property 2.2:

$$\begin{aligned} 3 \geq 5 \ \Rightarrow_B \mathit{subp}(3,5) &= 3 - 5 \\ \mathbf{false} \ \Rightarrow_B \bot_{\mathbb{Z}} &= -2 \\ \mathbf{false} \ \Rightarrow_B \bot_{\mathbb{B}}& \\ \mathbf{true}& \end{aligned}$$

and an evaluation of Property 2.8:

$$\begin{aligned} \mathit{subp}(3,5) = 3 - 5 \vee_B \mathit{subp}(5,3) &= 5 - 3 \\ \bot_{\mathbb{Z}} = -2 \vee_B 2 &= 2 \\ \bot_{\mathbb{B}} \vee_B \mathbf{true}& \\ \mathbf{true}& \end{aligned}$$

2.2.5 Sequent Interpretations

In this section different interpretations of the sequent (consequence relation — $\vdash$) are discussed in the context of the strong Kleene approach. Thus the other approaches considered in this section use the same propositional operations and quantifiers as mentioned earlier for LPF (which is based on the strong Kleene logic) in Section 2.2.4, but different interpretations are given to the sequent statement.

One possible interpretation of the sequent has already been discussed and that is the *SS* interpretation that is used in the LPF approach. Other interpretations that can be given to sequents in the strong Kleene approach are: *SW*, *WW* and *WS*, where *W* stands for a *weak* ($\Delta p \Rightarrow p$) interpretation and *S* stands for a *strong* ($\Delta p \wedge p$) interpretation. Let $\Gamma = \{e_1, \ldots, e_n\}$ be a set of formulae, where the commas are to be interpreted as conjunctions, and let e be a single formula. A sequent is to be interpreted as from the set of formulae Γ, the formula e can be inferred, ($\Gamma \vdash e$). The first *W* or *S* is for the assumptions interpretation and the second *W* or *S* is for the conclusion interpretation. Note that Γ is a set of formulae that can be empty. The choice of which interpretation is given to a sequent determines the inference rules that are sound, and thus has an impact on the applicability of certain proof procedures.

A comparison of this work on the different sequent interpretations (that is discussed below in this section) is presented in [Owe97]. A brief overview follows of the different sequent interpretations to enable some justifications in favour of LPF to be introduced.

A sequent is valid in the same manner as in two-valued classical logic whenever Γ and e are defined. That is, a sequent is valid iff for all interpretations, if Γ is true in an interpretation (that is, where all formulae in Γ are true), e is also true in that interpretation, or if an interpretation makes Γ false (that is, there is a formula in Γ that is false). However, the sequent is invalid whenever there exists an interpretation that makes Γ true but e false.

The *SS* (LPF) sequent interpretation is invalid whenever Γ is true, and e is false or undefined in an interpretation. The sequent is valid in all other cases.

The *SW* sequent interpretation is invalid whenever Γ is true, and e is false in an interpretation. The sequent is valid in all other cases.

The *WW* sequent interpretation is invalid whenever Γ is true, and e is false in an interpretation. The sequent is also invalid whenever Γ is undefined and e is false in an interpretation. The sequent is valid in all other cases.

The *WS* sequent interpretation is invalid whenever Γ is true, and e is false or undefined in an interpretation. The sequent is also invalid whenever Γ is undefined, and e is false or undefined in an interpretation. The sequent is valid in all other cases.

All four sequent interpretations have inference rules that hold which do not hold in other sequent interpretations, and all four sequent interpretations have inference rules that do not hold.

For instance, in the *SS* interpretation, as already mentioned the deduction theorem:

$$\frac{p \vdash q}{p \;\Rightarrow\; q}$$

does not hold. Also proof by contradiction does not hold in this interpretation.

In the *SW* interpretation, modus ponens:

$$\frac{p; p \;\Rightarrow\; q}{q}$$

does not hold. Additionally, the cut rule does not hold.

In the *WW* interpretation, again modus ponens does not hold.

In the *WS* interpretation, the trivial sequent:

$$\frac{}{p \vdash p}$$

does not hold.

The sequent interpretations are formalised in Section 4.2.

A sound and a relatively complete set of proof rules for *WS* is presented in [Owe97].

2.3 Summary of the Justifications for LPF

There are two arguments that are needed to justify the choice of LPF. First that the semantics of the logical operators etc. in LPF is the preferred way to reason about logical formulae that can include references to partial terms. Second to justify that the LPF *SS* sequent interpretation is the preferred sequent interpretation.

It is pleasing that it is relatively trivial to be able to convert theorems between two-valued classical logic and LPF. The big difference between LPF and two-valued classical logic is the loss of the law of the excluded middle. All theorems of LPF are theorems of two-valued classical logic, for example, if all partial functions are overspecified. But some theorems of two-valued classical logic cannot be proved in LPF, due to the loss of the law of the excluded middle in LPF. Adding a definedness hypotheses for all propositions in a valid two-valued classical logic formula is sufficient to make the validity of a formula in two-valued classical logic and in LPF coincide.

Another big consideration in the choice of a logic is the ease with which proofs can be completed. In LPF one can reason about recursive functions using inference rules that can be generated from recursive function definitions. For instance, for the *zero* function the following two inference rules can be generated:

$$\boxed{zero\_b}\ \frac{}{zero(0) = 0}$$

$$\boxed{zero\_i}\ \frac{i: \mathbb{Z};\quad i \neq 0;\quad zero(i-1) = k}{zero(i) = k}$$

where the equality used in the inference rules is weak equality. One can imagine the automatic generation of such inference rules from the function definitions themselves.

These inference rules can be used to aid completing proofs of properties relating to the *zero* function. The proof of Property 1.1 is presented in Figure 2.12, and is referred to as *Lem* in the subsequent proof. This proof is complicated by the fact that a δ definedness obligation needs to be discharged in the proof step referenced by the line number 5. In this case showing definedness is trivial, since in this proof the definedness of $i \geq 0$ follows from the type $i: \mathbb{Z}$.

```
      from $i: \mathbb{Z}$
1         $zero(0) = 0$                                   $zero\_b$
2         from $n: \mathbb{N}; zero(n) = 0$
2.1           $n + 1: \mathbb{Z}$                          2.h2, $\mathbb{Z}$
2.2           $n + 1 \neq 0$                               2.h1, 2.1, $\mathbb{Z}$
          infer $zero(n + 1) = 0$                         $zero\_i(2.1, 2.2, 2.h2)$
3         $\forall n: \mathbb{N} \cdot zero(n) = 0$       $\forall$-$I$($\mathbb{N}$-ind(1, 2))
4         from $i \geq 0$
4.1           $i: \mathbb{N}$                              4.h1, $\mathbb{N}$
          infer $zero(i) = 0$                             $\forall$-$E$(3, 4.1)
5         $\delta(i \geq 0)$                               h1, $\mathbb{Z}$
      infer $\forall i: \mathbb{Z} \cdot i \geq 0 \Rightarrow zero(i) = 0$    $\forall$-$I$($\Rightarrow$-$I$(5, 4))
```

Figure 2.12: Proof of *zero* Property 1.1

A proof of Property 1.3 is presented in Figure 2.13. This proof is identical

```
     from $i\colon\mathbb{Z}$
1           $i \geq 0 \vee i < 0$                                        h1, $\mathbb{Z}$
2           from $i \geq 0$
2.1               $zero(i) = 0$                              $\Rightarrow$-$E$-$L$(Lem(h1), 2.h1)
            infer $zero(i) = 0 \vee zero(-i) = 0$                       $\vee$-$I$-$R$(2.1)
3           from $i < 0$
3.1               $-i \geq 0$                                          h1, 3.h1, $\mathbb{Z}$
3.2               $-i\colon\mathbb{Z}$                                  h1, 3.1, $\mathbb{Z}$
3.3               $zero(-i) = 0$                            $\Rightarrow$-$E$-$L$(Lem(3.2), 3.1)
            infer $zero(i) = 0 \vee zero(-i) = 0$                       $\vee$-$I$-$L$(3.3)
     infer $\forall i\colon\mathbb{Z} \cdot zero(i) = 0 \vee zero(-i) = 0$    $\forall$-$I$($\vee$-$E$(1, 2, 3))
```

Figure 2.13: Proof of *zero* Property 1.3

to how a proof of the same property would look in two-valued classical logic. It is not necessary to make any assumption about the meaning of terms such as $zero(-1)$, as in LPF one reasons from truth to truth.

The proofs of the corresponding *subp* properties, can all be proved in a similar way.

An issue that can be raised about the non-classical logic approaches is that the familiar logic rules can become complicated by the addition of definedness obligations to ensure the soundness of such rules. Since such definedness obligations need discharging more time and effort is needed by a user during the proof process. But this is also the case in the well-definedness approach where despite the fact that all reasoning can be done in two-valued classical logic, definedness needs to be established in a separate proof to the validity proof.

The approaches that avoid the need for a non-classical logic raise subtle questions and issues, for instance, forcing the introduction of multiple notions of equality, forcing arbitrary values to be returned by a function if the function is applied with argument(s) from outside of the partial functions defined domain, or separating definedness out from validity proofs. Such issues have been outlined in Section 2.2. Certain non-classical logics can provide a *natural* way of reasoning about logical formulae that can contain references to partial terms, but at the expense of having to use a non-standard (unfamiliar) three-valued logic semantics.

All of the non-classical three-valued logics considered do have the pleasing property that any formula that only includes defined operands corresponds to the two-valued classical logic interpretation. There are differences between these non-classical logics that have been considered, that lead the author to the choice of LPF.

In weak Kleene's (Bochvar's Internal approach) where if undefinedness arises then undefinedness results, it is preferred that the strongest possible interpretation is given to the propositional logical operators, and to the quantifiers, that is, that they are as defined as possible. If a result can be determined from a single operand and such a result cannot be contradicted later by the completion of further evaluations, then it is preferred that such a result should be returned.

The sequential interpretation that is given to the logical operators in McCarthy's conditional operator approach is used in certain tool support environments; for example the McCarthy conditional disjunction operator is closely related to the standard disjunction that is provided in programming languages such as *C* and *Java*, as a sequential interpretation can be implemented instead of a parallel interpretation. However, when conducting proofs the parallel interpretation of LPF (strong Kleene logic) is preferred over McCarthy's sequential interpretation as the former, for instance allows for both $\mathbf{true} \vee \perp_{\mathbb{B}}$ and $\perp_{\mathbb{B}} \vee \mathbf{true}$ to be true, while in the latter approach only the first of these two examples is true.

Due to the sequential interpretation that is given to the logical operators in McCarthy's conditional operator approach, basic algebraic properties such as the commutativity of disjunctions and conjunctions do not hold. Having to conduct proofs where operands to the disjunction and the conjunction logical operators cannot be *easily* commuted can be tedious; this can for example complicate mechanised proof procedures, for instance complicating the clausal form notation that is used in the resolution proof procedure and so on. Issues such as this is why the parallel interpretation of LPF is preferred here for conducting proofs in. The truth tables in LPF provide the strongest possible monotonic extension of the corresponding familiar two-valued truth tables. Thus properties such as the commutativity and the distributivity of conjunctions and disjunctions hold, as well as familiar properties such as the idempotent and the de Morgan properties. There is an argument that the acceptance of logics (to cope with partial terms) can be decided by their algebraic properties [CJ91].

In Łukasiewicz's approach $p \quad \Rightarrow_{\text{Ł}} p$ is true, and in Bochvar's External approach $\neg_{B} \perp_{\mathbb{B}}$ is true. Such logics are not monotone, and if a function later completes its evaluation (it becomes defined), results can changes from true to false. Thus Łukasiewicz's approach and Bochvar's external approach are not deemed as appropriate in comparison to other approaches considered here for this work.

Additionally, the *SS* sequent interpretation is preferred over the other sequent interpretations. In the *SS* sequent interpretation one can only infer truth from truth. While in the *SW* sequent interpretation, an interpretation where $\mathbf{true} \vdash \bot_{\mathbb{B}}$ occurs is actually valid. The same holds for the *WW* sequent interpretation. In the *WS* sequent interpretation, an interpretation that gives rise to either $\bot_{\mathbb{B}} \vdash \mathbf{false}$ or to $\bot_{\mathbb{B}} \vdash \bot_{\mathbb{B}}$ causes the sequent to be invalid. In the *SS* sequent interpretation, when the left hand side is false or is undefined in an interpretation, there is no constraint on the goal in such an interpretation. Of course no matter which sequent interpretation is chosen in the strong Kleene approach, some two-valued classical proof rules do not hold and need modifying with definedness conditions for soundness. The notion of strong validity is too strong, and can distract from the overall goal that is to deduce truth from truth.

So, LPF stays *close* to two-valued classical logic in that a lot of the proof rules of two-valued classical logic are still valid in LPF. However, some proof rules (such as the deduction theorem, see LPF Section 2.2.4) need modifying to maintain soundness, and some extra proof rules are needed for completeness. Thus such a logic is less familiar for a user accustomed to two-valued classical logic. One big drawback of LPF is that the proof by contradiction proof technique does not follow as it does in two-valued classical logic due to the presence of "gaps". However, in Chapter 6 it is shown that such a technique can be modified to cover LPF.

There have been arguments presented for the adoption of LPF for a while, for instance in [CJ91, Jon06]. As mentioned an obstacle against the adoption of LPF concerns the lack of proof support that is available for LPF. Attempting to address this obstacle can be seen as an overall aim of this work. A focus of this work is on investigating the applicability of basic but fundamental proof techniques in LPF. Extra work will result in LPF due to definedness obligations that will need discharging due to the presence of partial terms, but identifying the extent of this extra work, and how to address and reduce this extra work is the goal. In Chapter 6 an investigation into the applicability of fundamental proof techniques in LPF is presented which goes a long way to addressing the issue of a lack of mechanised proof support for LPF. Addressing the issue of proof support for LPF can be used to counter the obstacle against LPF, and in doing so to further justify the choice of LPF for reasoning about logical formulae that can contain references to partial terms.

Chapter 4 provides a semantically based comparison between numerous different approaches that have been discussed in this chapter. Relationships, in particular how theorems can be moved between the different approaches are also identified from this comparison.

2.4 Previous Attempts at Mechanisation Support for LPF

LPF is the logic that underlies VDM [Jon90, FLM$^+$05, FL09], but its potential for application goes beyond that particular formalism [Fit07]. VDM was established back in the 1970s from work done at IBM's Vienna Laboratory. VDM is a range of techniques centered around the modelling, the specification and the design of computer systems [Fit07] and is a widely used model-based formal method [LBF$^+$10]. A model-oriented formal method is used to define the behaviour of a system by constructing a model of the system using mathematical structures such as sets and functions [Win90].

The issues posed behind proving properties of computer systems which include references to partial functions was a motivating factor behind the development of LPF [Fit07]. It has already been illustrated that partial terms arise frequently in program specifications and this is indeed the case in VDM models, as can be seen in a wide range of literature, for example [Jon90, BFL$^+$94, Bic98, FLM$^+$05, Jon06, Fit07].

To date there have been numerous attempts at providing a mechanisation of LPF which have all been centered around work done for VDM, notably tools which have been designed to encourage the modelling and exploration of models [Fit07] (the VDM Toolset and the Overture Toolset); a formal development support system (mural); and attempts at providing proof support, for instance using the PVS theorem prover and the HOL theorem prover.

The aim of this section is twofold. Firstly, to introduce what was achieved in the approaches alluded to above, and secondly to illustrate how the work to be presented in the following chapters of this thesis differs from the previous attempts at mechanising VDM/LPF.

2.4.1 The VDM Toolset and the Overture Toolset

The commercial VDM Toolset [ELL94, Fit07, FLS08][1] was designed to encourage modelling in VDM and the exploration of VDM models. Facilities provided include but are not limited to: syntax checking, static type checking, the execution of models in an interpreter (testing by execution), and the generation of proof obligations.

The Overture Project[2] is an open source initiative still currently under development [LBF$^+$10]. Its aim is to provide tools to support the modelling of computer-based systems using VDM. Currently the Overture tool supplies tool support for creating VDM models such as syntax checking and static type

[1]www.vdmtools.jp
[2]www.overturetool.org

checking, as well as providing functionality that allows for VDM models to be executed and to be tested.

One of the key ideas behind LPF is to provide a parallel (lazy) evaluation of the operands of the propositional operators, such that:

$$0 = 0 \lor (1/0) = 0$$

and:

$$(1/0) = 0 \lor 0 = 0$$

are both true. However, due to issues in mechanising such a parallel evaluation the VDM Toolset and the Overture Toolset do not provide the propositional operators present in VDM with the LPF interpretation, but instead give the propositional operators McCarthy's conditional operator interpretation [Fit07]. Thus only the first of the two examples above can be evaluated to true, and interpreting the second example just results in an error being returned.

Before continuing it is also worth pointing out that proof obligations can be generated through tool sets such as the VDM Toolset and the Overture Toolset. A proof obligation is an unproven Boolean expression that highlights some constraint/property from a given model that must be discharged in order for the model to be able to be regarded as consistent.

2.4.2 mural

The mural tool is a Formal Development Support System [JJLM91]. The aim of mural was to provide a support tool and a proof assistant for VDM. The mural tool was specified in VDM.

The mural tool supports the full development cycle. A number of tools support different stages of this cycle such as the toolsets outlined in the previous section for the specification phase and the proof obligation generation phase. The mural tool supports the construction of VDM specifications, has facilities to generate the associated proof obligations, as well as providing support to construct proofs for instance, of the associated proof obligations.

The generic proof tool of mural can be used with a wide range of logics. The logical frame of the mural tool has been instantiated with different logics such as LPF and First-Order Predicate Calculus (FOPC).

The mural proof construction support tool is an interactive proof assistant. A key requirement envisaged for the mural tool is that the proof assistant tool should help users in the proof construction process, that is, that the human rather than the program should be in control throughout the proof construction process (user guided proof). The proofs are to be conducted interactively with some automated aids.

For an account of using the mural tool to specify a system, the reader is referred to [VMH01].

2.4.3 Utilising Existing Theorem Provers

The following approaches attempted to use existing theorem provers to try and discharge proof obligations.

In [Bic98, §6] the authors discuss utilising the PVS theorem prover [ORS92, COR$^+$95] as a tool to support VDM-SL. The aim was to translate VDM-SL specifications into the PVS specification language, and to be able to use PVS for type checking and for verifying properties of VDM-SL specifications. A large subset of VDM-SL can be translated to PVS but it must be done manually. The PVS proof checker can be used for proving proof obligations that arise from specifications. Logical formulae from VDM-SL need representing as logical formulae in PVS, so that the proof capabilities of PVS can be used for VDM.

However, the proof rules of VDM-SL are not accurately captured because of the differences between the logics of VDM-SL and PVS. PVS as mentioned earlier does not support actual partial functions (division by zero for example is subtyped), so PVS will generate obligation(s) to ensure that the functions are total (partial VDM-SL functions need translating into total PVS functions).

A tool that automatically translates a large subset of VDM and its associated proof obligations to the theorem prover HOL is discussed in [Ver07, VHL10]. The goal of this work was to be able to discharge as many proof obligations as possible that are generated by the VDM Toolset automatically using the theorem prover HOL. This work involved the development of a VDM++ to HOL translator and using HOL to then prove the proof obligations of the model.

It is mentioned in [Ver07] that one of the main challenges in the translation is: "Partial to Total: All partial functions in VDM need to get some kind of total representation in HOL, as HOL does not allow partial functions". This is because LPF is the logic that underlies VDM, while HOL uses a two-valued logic. Thus this approach is not faithful to the semantics of LPF, as they stay within a two-valued subset of VDM models.

This work on HOL built upon work that was undertaken in the PROSPER project [DCN$^+$00]. A case study that was undertaken in this project was to investigate the automatic translation of VDM-SL models into the HOL 98 theorem prover. But again this work is not faithful to LPF.

In [AF97] and also documented in [Bic98, §7] the authors attempt to instantiate Isabelle with a VDM-SL variant of LPF. However, this attempt again suffers due a lack of faithfulness to LPF, when it comes to undefinedness.

In a similar fashion to the attempts made in HOL, the approach documented in [WF08] uses domain checks to attempt to avoid having to conduct proofs in a three-valued logic based setting.

In the paper on linking VDM and Z [WF08] the authors show that a theorem prover for Z (Z/Eves) can be used a verify theorems in VDM. Despite the fact that VDM uses a three-valued logic (LPF) to cope with partial terms while Z uses a semi-classical logic, and the theorem prover Z/Eves uses two-valued classical logic.

Classical logic is used to reason about facts in VDM, where the soundness of the proofs relies on finding guards to guarantee the definedness of expressions. In order to prove a VDM fact e the authors proceed along the lines of taking a guard G for the formula e. By proving in classical logic that both G and e can be inferred the authors illustrate how e follows in LPF. Such guards can, however, expand exponentially. This approach is like the Well-Definedness (WD) approach, as a guard needs proving, and a separate validity proof needs proving.

The *subp* function is used as an example by the authors in [WF08], and when Z/Eves checks the *subp* equation a domain check using guards is generated which needs proving to ensure that every application of *subp* is within the defined domain of the *subp* function. The defined domain of the *subp* function needs specifying to be able to discharge these guards.

However, as alluded to earlier, both:

$$\mathit{subp}(3,5) = \mathit{subp}(3,5)$$

and:

$$\mathit{subp}(3,5) = 0 \vee \neg\,(\mathit{subp}(3,5) = 0)$$

can be proven valid in Z/Eves. This is despite the fact that the term $\mathit{subp}(3,5)$ fails to denote a proper value, but due to the semi-classical nature of Z, it must be the case that predicates denote.

2.5 Conclusions

This chapter has provided a comprehensive survey of different approaches to coping with partial terms, e.g. arising from the application of partial functions.

The different approaches all attempt to address the issues that partial terms bring about as discussed in Section 1.1.4. The survey has addressed how the different approaches cope with the presence of partial terms; the advantages of the different approaches; issues that arise with the different approaches; and the logical semantics of the different approaches, e.g. whether they maintain the use of two-valued classical logic, or whether they are a non-classical logic approach.

Justifications for taking the LPF approach have been outlined. LPF is the logic that underlies VDM. Previous work on mechanising LPF (usually in the context of VDM) has been discussed. The existing work on mechanising LPF either has different goals as to what will be presented in the following chapters such as the VDM Toolset, or was not conducting reasoning in a way completely faithful to LPF.

A formal semantic comparison of a number of the approaches to coping with partial terms presented in this chapter will be presented in Chapter 4. This allows for further justifications for LPF to be presented.

Chapter 3

Semantic Definitions for LPF

Contents

3.1 Expression Constructs 56
3.2 Context Conditions 58
3.3 Operational Semantics 63
3.3.1 Big-Step Structural Operational Semantics Definition 65
3.3.2 Small-Step Structural Operational Semantics Definition 67
3.4 Denotational Semantics 80
3.5 Relationships between the Semantic Definitions . 84
3.6 Conclusions 89

The semantics of LPF is formally defined in this chapter using different semantic definitions. The aim of dynamic semantics is to give a meaning to a construct by defining its evaluation. In particular two types of semantic definitions are presented in this chapter. The first is known as Operational Semantics (OS) and the second is known as Denotational Semantics (DS) [NN92]. Both semantic formalisations are used to provide the meaning of well-formed (syntactically correct) expression constructs by defining their evaluation according to the semantics of LPF.

For the first semantic description both a big-step semantics and a small-step semantics is provided. The former is generally referred to as a natural semantics [Kah87] and the latter is generally referred to as a Structural Operational Semantics (SOS) [Plo81, Plo04]. Here the former is referred to as a big-step SOS and the latter as a small-step SOS.

An OS definition is concerned with how programs are executed not just with what the results are [NN92]. How the final value is computed must be stated. Big-step SOS definitions serve the purpose of describing how overall results of execution are obtained, while small-step SOS definitions serve the purpose of describing how individual steps take place [NN92].

In the DS approach, the interest is in the effect (the association between initial and final states) of executing a program, not how it is obtained [NN92].

"The meaning of a program is modelled by mathematical objects that represent the effect of executing the constructs" [NN92].

The SOS definitions provide an intuitive introduction to the semantics of LPF, and how LPF addresses the issues of handling logical formulae that can contain references to partial terms, by illustrating the process of how expressions are evaluated in LPF. The DS definitions provide set theoretic definitions of the values that are denoted by expressions according to the semantics of LPF.

It is beneficial to provide such definitions since doing so allows one to be clear about the semantics of LPF before beginning with a mechanisation of it. Additionally, such formal semantic definitions can form the basis of mechanisations, and they provide criteria with which to check whether any mechanisation is "correct", (e.g. they provide a key underlying basis on which proofs of modifications made to proof procedures for LPF can be conducted on, see Chapter 6). Mechanisations in both the Maude term-rewriting system, and in the Isabelle proof assistant are considered in Chapter 5.

A DS semantic definition is made use of in subsequent chapters to present a formal comparison between different approaches to coping with partial terms (in Chapter 4), as well as being used, as a basis from which to prove modifications of two-valued classical logic proof techniques for LPF, to precisely define concepts, and to illustrate issues with applying selected proof techniques to LPF (in Chapter 6).

Before the semantics are presented the expression constructs for which the semantics are to be provided are presented. This is followed by the context conditions which allow for the ill-formed expressions that can be constructed to be removed from further consideration, that is, to be removed from further consideration in the semantic definitions. The semantic objects that are needed, followed by the semantic definitions themselves (the transition relations for the SOS definitions and the DS set theoretic definitions) are then presented which are followed by a number of illustrative proofs, for example, to show that some of the different semantic definitions presented coincide.

(Some of the content from Sections 3.1, 3.2, and 3.3.2 are presented in the paper [Lov10], but such content has been extended for this chapter. Some of the content from Sections 3.3.2 and 3.4 are presented in [JL11] but with concrete syntax, whereas abstract syntax and context conditions are used here. The content has again been thoroughly extended for this chapter. The SOS definitions are mine. Cliff Jones also suggested producing DS definitions (a small basis of which was provided, which was then modified and extended by myself), which were derived from the SOS definitions. The paper [JL11] also contains proofs on a partial predicate done by Cliff Jones which forms no part of this thesis.)

3.1 Expression Constructs

The expression constructs present in the language are all introduced below using abstract syntax [McC62, §12]. The purpose of using abstract syntax as opposed to using concrete syntax is to allow for the necessary information to be conveyed without having to concern oneself about the actual (concrete) syntactical representation of such expressions. For instance, which of the following syntactical representations for a disjunction expression should be used: $p \vee q$, or $p \; or \; q$, $p \mid q$, $p \mid\mid q$, $\vee \, (p, q)$, or $or(p, q)$? Here such considerations do not matter so abstract syntax is used so that just the essential information is presented as opposed to having to also convey the syntactical representation of the expression constructs. Abstract syntax is independent of the notation used, and is similar to the function that a Back-Naur Form specification provides, but without any concern for the concrete syntactical representation.

The "basic" language includes numerous expression constructs, where all of the expressions must be of the type Boolean or of the type integer. The restriction on the domain is for simplicity in the following semantic definitions only. But, at the same time, even with just these two types, the issues encountered with partial terms can still be adequately illustrated. Extending the following semantic definitions to incorporate more datatypes is straightforward. This can be necessary because most programming languages contain many different datatypes.

A constant value, that is, a Boolean value ($\mathbb{B}$, **true** or **false**) or an integer value ($\mathbb{Z}$, $\ldots, -1, 0, 1, \ldots$) is itself treated as an expression. Other valid expressions in the language include: referring to an identifier; arithmetic expressions; a relational (equality) expression; a conditional expression (useful for defining recursive function definitions); propositional logic expressions, in particular negation, disjunction, and a definedness operator δ; an existential quantifier; and function and predicate call expressions.

Of course certain logical operators that are not presented here can be defined in terms of the subset presented just as in two-valued classical logic, for example, $p \wedge q$ is equivalent to $\neg\,(\neg\, p \vee \neg\, q)$, $p \;\Rightarrow\; q$ is equivalent to $\neg\, p \vee q$, $p \;\Leftrightarrow\; q$ is equivalent to $(p \;\Rightarrow\; q) \wedge (q \;\Rightarrow\; p)$, and $\forall x \cdot p$ is equivalent to $\neg\, \exists x \cdot \neg\, p$ in LPF. The definitions of such operators could of course be defined in the different semantic definitions, but since such operators can be defined in terms of the other operators in LPF, the inclusion of these other operators would expand the definitions at the expense of clarity.

In addition only quantification over the set of integer values is considered for simplicity. Recall that in LPF quantification is only over a set of proper (i.e. defined) values. It is straightforward to remove this restriction.

Four sorts of identifiers can occur in expressions, those for propositions (*Prop*), for integer variables (*Var*), for functions (*Fn*) and for predicates (*Pr*). *Prop*, *Var*, *Fn* and *Pr* are assumed to be disjoint sets. Notice that while a $e \in \mathit{Prop}$ and a $e \in \mathit{Var}$ are expressions, any reference to a *Fn* or to a *Pr* identifier must be made through the *FuncCall* and the *PredCall* expression constructs.

The abstract syntax is:

$$\begin{aligned}\mathit{Expr} = {} & \mathit{Value} \mid \mathit{Id} \mid \mathit{Arith} \mid \mathit{Equality} \mid \mathit{Cond} \mid \mathit{Not} \mid \mathit{delta} \mid \\ & \mathit{Or} \mid \mathit{Exists} \mid \mathit{FuncCall} \mid \mathit{PredCall}\end{aligned}$$

$$\mathit{Value} = \mathbb{B} \mid \mathbb{Z}$$

$$\mathit{Id} = \mathit{Prop} \mid \mathit{Var} \mid \mathit{Fn} \mid \mathit{Pr}$$

$$\begin{aligned}\mathit{Arith} :: \; & a : \mathit{Expr} \\ & op : + \mid - \mid \times \mid \div \\ & b : \mathit{Expr}\end{aligned}$$

$$\begin{aligned}\mathit{Equality} :: \; & a : \mathit{Expr} \\ & b : \mathit{Expr}\end{aligned}$$

$$\begin{aligned}\mathit{Cond} :: \; & p : \mathit{Expr} \\ & a : \mathit{Expr} \\ & b : \mathit{Expr}\end{aligned}$$

$$\mathit{Not} :: p : \mathit{Expr}$$

$$\mathit{delta} :: p : \mathit{Expr}$$

$$\begin{aligned}\mathit{Or} :: \; & p : \mathit{Expr} \\ & q : \mathit{Expr}\end{aligned}$$

$$\begin{aligned}\mathit{Exists} :: \; & x : \mathit{Var} \\ & p : \mathit{Expr}\end{aligned}$$

Function/predicate call expressions require both the name of the function/predicate to be called as well as the arguments to be passed into the function/predicate:

$$\begin{aligned}\mathit{FuncCall} :: \; & \mathit{function} : \mathit{Fn} \\ & \mathit{args} : \mathit{Expr}^*\end{aligned}$$

$$\begin{aligned} \mathit{PredCall} :: \mathit{predicate} &: \mathit{Pr} \\ \mathit{args} &: \mathit{Expr}^* \end{aligned}$$

A function in the language always takes integer arguments and returns an integer result. A function definition thus contains a sequence of parameter names and a resulting expression (that might include recursive calls to the function), where a record is used to represent a single function definition:

$$\begin{aligned} \mathit{Func} :: \mathit{params} &: \mathit{Var}^* \\ \mathit{result} &: \mathit{Expr} \end{aligned}$$

A predicate is similar; it always takes integer arguments but will instead return a Boolean result:

$$\begin{aligned} \mathit{Pred} :: \mathit{params} &: \mathit{Var}^* \\ \mathit{result} &: \mathit{Expr} \end{aligned}$$

It is assumed that all variables and function/predicate definitions are defined before use alongside the expression to be evaluated, as follows:

$$\begin{aligned} \mathit{Start} :: \quad \mathit{vars} &: \mathit{Prop} \mid \mathit{Var} \xrightarrow{m} \mathit{Value} \\ \mathit{funcs} &: \mathit{Fn} \xrightarrow{m} \mathit{Func} \\ \mathit{preds} &: \mathit{Pr} \xrightarrow{m} \mathit{Pred} \\ \mathit{body} &: \mathit{Expr} \end{aligned}$$

3.2 Context Conditions

Now that the abstract syntax has been presented the context conditions can be introduced. The abstract syntax provides a structure for all expressions. The set of expressions from the abstract syntax is a proper subset of all well-formed expressions. The purpose of the context conditions is to be able to remove ill-formed expression constructs from consideration in the semantic definitions that follow, thus leaving only those well-formed expressions. For instance, semantics should be provided for expressions such as $\mathit{mk\text{-}Arith}(1, +, 2)$ and $\mathit{mk\text{-}Or}(\mathbf{true}, \mathbf{false})$, but not for expressions such as $\mathit{mk\text{-}Arith}(1, +, \mathbf{true})$ and $\mathit{mk\text{-}Or}(\mathbf{true}, 1)$ which can be constructed, according to the abstract syntax presented.

There are two types in the language specifically the Boolean type and the integer type:

$$\mathit{Type} = \textsc{Bool} \mid \textsc{Int}$$

where Bool and Int are primitive tokens naming the types $\mathbb{B}$ and $\mathbb{Z}$ respectively.

To be able to perform type checks in the language a map entitled *Types* is introduced that maps variable identifiers to the type of data that the variables store, for example, a *Prop* will map to a Bool, and a *Var* will map to a Int:

$$\begin{aligned} & \mathit{Types} = \mathit{Prop} \xrightarrow{m} \textsc{Bool} \mid \\ & \qquad \mathit{Var} \xrightarrow{m} \textsc{Int} \end{aligned}$$

Furthermore, a map entitled *Defs* is introduced that maps function and predicate identifiers to the corresponding function and predicate definitions, to enable more type checks to be made:

$$\begin{aligned} & \mathit{Defs} = \mathit{Fn} \xrightarrow{m} \mathit{Func} \mid \\ & \qquad \mathit{Pr} \xrightarrow{m} \mathit{Pred} \end{aligned}$$

It is intended that the only expressions for which, the semantics will be provided are those expressions which satisfy the following criteria:

- A constant expression (of the type Bool or Int);
- A variable identifier (*Prop* or *Var*) which is defined within the domain of a given *Types* map, and thus maps to an appropriate type (Bool or Int);
- An arithmetic expression if both of its operands are well-formed and are both of the type Int, and the operator is $+$, $-$, $\times$, or $\div$;
- A relational (equality) expression if both of its operands are well-formed and of the type Int;
- A negation expression if its operand is well-formed and of the type Bool;
- A definedness operator δ expression if its operand is well-formed and of the type Bool;
- A disjunction expression if both of its operands are well-formed and of the type Bool;
- A conditional expression if the expression condition is well-formed and of the type Bool, and the true and the false sub-expressions are both well-formed and of the type Int;
- An existentially quantified expression if the quantified expression is well-formed and of the type Bool when the quantified variable is included within the given *Types* map and is constrained to be of the type Int;
- A function call expression if the arguments are well-formed and of the type Int, and the function to be called exists in the given *Defs* map; and
- A predicate call expression if the arguments are well-formed and of the type Int, and the predicate to be called exists in the given *Defs* map.

These criteria are presented formally in Figure 3.1 and Figure 3.2.

As well as checking whether expressions are well-formed, function and predicate definitions need to be checked to ensure that they are well-formed. Function and predicate definitions are considered well-formed if the result expression is of the same type as the intended return type of the function/predicate definitions in this language notably Int and Bool respectively. This is formalised in Figure 3.3 and Figure 3.4. It is intended that the only variables that should be used within a functions/predicates result expression are the parameter variables, so a functions/predicates result expression is checked to see if it is well-formed with only the functions/predicates parameter names included as a variable within a given *Types* map.

All of the context conditions (*wf* functions) that have been presented are total as they are defined on all possible members of *Expr*.

Having the context conditions simplifies the syntax and the following semantic definitions since it can be assumed in Section 3.3 and in Section 3.4 that for any expression, and function definitions and predicate definitions constructed and used in the following semantic definitions, that they satisfy all of the necessary context conditions.

An Alternative Approach

A solution to reducing the size of the context conditions that are needed is to define *Expr* along the following lines:

$$Value = \mathbb{B} \mid \mathbb{Z}$$

$$Id = Prop \mid Var \mid Fn \mid Pr$$

$$Expr = BoolExpr \mid IntExpr$$

$$\begin{aligned} BoolExpr = {} & \mathbb{B} \mid Prop \mid Equality \mid Not \mid delta \mid \\ & Or \mid Exists \mid PredCall \end{aligned}$$

$$IntExpr = \mathbb{Z} \mid Var \mid Arith \mid Cond \mid FuncCall$$

$$\begin{array}{lrcl} Arith :: & a & : & IntExpr \\ & op & : & + \mid - \mid \times \mid \div \\ & b & : & IntExpr \end{array}$$

$$\begin{array}{lrcl} Equality :: & a & : & IntExpr \\ & b & : & IntExpr \end{array}$$

$$\begin{array}{lrcl} Cond :: & p & : & BoolExpr \\ & a & : & IntExpr \\ & b & : & IntExpr \end{array}$$

```
$wf\text{-}Expr : Expr \times Types \times Defs \to (Type \mid \textsc{Error})$

$wf\text{-}Expr(e, vars, defs) \quad \triangleq$
    **cases** $e$ **of**
                    $e \in \mathbb{B} \to \textsc{Bool}$
                    $e \in \mathbb{Z} \to \textsc{Int}$
                 $e \in Prop \to$ **if** $e \in$ **dom** $vars$
                              **then** $\textsc{Bool}$
                              **else** $\textsc{Error}$
                  $e \in Var \to$ **if** $e \in$ **dom** $vars$
                              **then** $\textsc{Int}$
                              **else** $\textsc{Error}$
        $mk\_Arith(a, op, b) \to$ **let** $l = wf\text{-}Expr(a, vars, defs)$ **in**
                           **let** $r = wf\text{-}Expr(b, vars, defs)$ **in**
                           **if** $l = \textsc{Int} \land l = r \land op \in \{+, -, \times, \div\}$
                           **then** $\textsc{Int}$
                           **else** $\textsc{Error}$
        $mk\_Equality(a, b) \to$ **let** $l = wf\text{-}Expr(a, vars, defs)$ **in**
                           **let** $r = wf\text{-}Expr(b, vars, defs)$ **in**
                           **if** $l = \textsc{Int} \land l = r$
                           **then** $\textsc{Bool}$
                           **else** $\textsc{Error}$
         $mk\_Cond(p, a, b) \to$ **let** $l = wf\text{-}Expr(p, vars, defs)$ **in**
                           **let** $r = wf\text{-}Expr(a, vars, defs)$ **in**
                           **let** $s = wf\text{-}Expr(b, vars, defs)$ **in**
                           **if** $l = \textsc{Bool} \land r = \textsc{Int} \land r = s$
                           **then** $\textsc{Int}$
                           **else** $\textsc{Error}$
               $mk\_Not(p) \to$ **if** $wf\text{-}Expr(p, vars, defs) = \textsc{Bool}$
                              **then** $\textsc{Bool}$
                              **else** $\textsc{Error}$
             $mk\_delta(p) \to$ **if** $wf\text{-}Expr(p, vars, defs) = \textsc{Bool}$
                              **then** $\textsc{Bool}$
                              **else** $\textsc{Error}$
             $mk\_Or(p, q) \to$ **let** $l = wf\text{-}Expr(p, vars, defs)$ **in**
                           **let** $r = wf\text{-}Expr(q, vars, defs)$ **in**
                           **if** $l = \textsc{Bool} \land l = r$
                           **then** $\textsc{Bool}$
                           **else** $\textsc{Error}$
         $mk\_Exists(x, p) \to$ **if** $wf\text{-}Expr(p, vars \dagger \{x \mapsto \textsc{Int}\}, defs) = \textsc{Bool}$
                              **then** $\textsc{Bool}$
                              **else** $\textsc{Error}$
                       ...
    **end**
```

Figure 3.1: The expression context conditions (part 1)

```
$wf\text{-}Expr : Expr \times Types \times Defs \to (Type \mid \textsc{Error})$

$wf\text{-}Expr(e, vars, defs) \quad \triangleq$
    **cases** $e$ **of**
                          $\ldots$
    $mk\_FuncCall(id, args) \to$ **if** $(\forall i\colon \textbf{inds}\ args \cdot$
                                        $wf\text{-}Expr(args(i), vars, defs) = \textsc{Int}) \wedge$
                                          $id \in \textbf{dom}\ defs \wedge$
                                          $\textbf{len}\ args = \textbf{len}\ defs(id).params$
                                 **then** $\textsc{Int}$
                                 **else** $\textsc{Error}$
    $mk\_PredCall(id, args) \to$ **if** $(\forall i\colon \textbf{inds}\ args \cdot$
                                        $wf\text{-}Expr(args(i), vars, defs) = \textsc{Int}) \wedge$
                                          $id \in \textbf{dom}\ defs \wedge$
                                          $\textbf{len}\ args = \textbf{len}\ defs(id).params$
                                 **then** $\textsc{Bool}$
                                 **else** $\textsc{Error}$
    **others** $\textsc{Error}$
    **end**
```

Figure 3.2: The expression context conditions (part 2)

```
$wf\text{-}Func : Func \times Types \times Defs \to \mathbb{B}$

$wf\text{-}Func(mk\_Func(p, r), vars, defs) \quad \triangleq$
    $wf\text{-}Expr(r, \{p(i) \mapsto \textsc{Int} \mid i\colon \textbf{inds}\ p\}, defs) = \textsc{Int}$
```

Figure 3.3: The function definitions context conditions

```
$wf\text{-}Pred : Pred \times Types \times Defs \to \mathbb{B}$

$wf\text{-}Pred(mk\_Pred(p, r), vars, defs) \quad \triangleq$
    $wf\text{-}Expr(r, \{p(i) \mapsto \textsc{Int} \mid i\colon \textbf{inds}\ p\}, defs) = \textsc{Bool}$
```

Figure 3.4: The predicate definitions context conditions

$$\begin{array}{l}\mathit{Not} :: p : \mathit{BoolExpr}\end{array}$$

$$\begin{array}{l}\mathit{delta} :: p : \mathit{BoolExpr}\end{array}$$

$$\begin{array}{llll}\mathit{Or} :: & p & : & \mathit{BoolExpr}\\ & q & : & \mathit{BoolExpr}\end{array}$$

$$\begin{array}{llll}\mathit{Exists} :: & x & : & \mathit{Var}\\ & p & : & \mathit{BoolExpr}\end{array}$$

$$\begin{array}{rrll}\mathit{FuncCall} :: & \mathit{function} & : & \mathit{Fn}\\ & \mathit{args} & : & \mathit{IntExpr}^*\end{array}$$

$$\begin{array}{rrll}\mathit{PredCall} :: & \mathit{predicate} & : & \mathit{Pr}\\ & \mathit{args} & : & \mathit{IntExpr}^*\end{array}$$

A function definition could be defined as:

$$\begin{array}{rrll}\mathit{Func} :: & \mathit{params} & : & \mathit{Var}^*\\ & \mathit{result} & : & \mathit{IntExpr}\end{array}$$

A predicate definition could be defined as:

$$\begin{array}{rrll}\mathit{Pred} :: & \mathit{params} & : & \mathit{Var}^*\\ & \mathit{result} & : & \mathit{BoolExpr}\end{array}$$

Some of the conditions in the context conditions are still needed, particularly those that make reference to the *Types* map and to the *Defs* maps. For instance, some of the conditions in the *mk_FuncCall* case of the *wf-Expr* context condition are still needed, $id \in \textbf{dom}\ defs$, and $\textbf{len}\ args = \textbf{len}\ defs(id).params$.

3.3 Operational Semantics

The LPF expression evaluation process is defined in terms of a set of transition relations, presented as inference rules which define the valid expression evaluations (transitions) that can occur for the expression constructs introduced earlier. The inference rules provide an abstraction of how an expression is evaluated in LPF.

Only those expressions that are constructed that pass the context conditions, that is, where given any $e \in Expr$ where $wf\text{-}Expr(e, vars) \neq$ ERROR, and reference well-formed functions and predicates, that is, for every function and predicate $wf\text{-}Func(\ldots) = \textbf{true}$ and $wf\text{-}Pred(\ldots) = \textbf{true}$, are considered from here on in the semantic definitions.

The SOS specifications provide an intuitive introduction to the semantics of LPF but are problematic when it comes to the quantified expressions.

All expressions in the language, that reduce to a constant value are to be considered defined, since such values cannot be reduced any further. The constant values present in this language are the Boolean values ($\{\textbf{true}, \textbf{false}\}$), and the integer values ($\{\ldots, -1, 0, 1, \ldots\}$). If an expression is evaluated to a member of one of these two sets then the expression is fully evaluated (no more evaluation can occur) and the evaluated expression denotes a value. For instance, the expression 0 denotes and the expression $\mathit{mk\_FuncCall}(\mathit{zero}, [0])$ denotes a value (given the definition of the *zero* function presented earlier), but the expression $\mathit{mk\_FuncCall}(\mathit{zero}, [-1])$ cannot be evaluated to a member of one of the two aforementioned sets, and thus, while the argument expression denotes, the whole expression fails to denote; it is a partial term, regarded as a "gap".

A map entitled Σ (referred to as a memory store) needs to be introduced to map identifiers to the values that they store, and the corresponding definitions at "run-time":

$$\begin{aligned}\Sigma = {} & \mathit{Prop} \xrightarrow{m} \mathbb{B} \mid \\ & \mathit{Var} \xrightarrow{m} \mathbb{Z} \mid \\ & \mathit{Fn} \xrightarrow{m} \mathit{Func} \mid \\ & \mathit{Pr} \xrightarrow{m} \mathit{Pred}\end{aligned}$$

where Σ is to be regarded as the set of all possible memory stores and σ ($\sigma \in \Sigma$) is used to represent a specific memory store. A memory store (σ) is a global static object in the sense that associations defined between the *Prop* and *Var* identifiers and the values that they store cannot be changed as a result of applying any of the semantic rules that follow. Additionally, no change can be made to a function definition or to a predicate definition by any of the semantic rules that follow. Notice that the last two maps in Σ are just the two maps from the *Defs* map that were presented in Section 3.2.

All variables must be present (and initialised) within a given σ before expression evaluation is undertaken using the following semantic rules. In this language there is no way to create new variables or to assign new values to variables.

The map involving *Prop* in a σ can be partial, that is, not include a mapping for a used propositional identifier, so a propositional identifier can be absent from the domain of a σ to allow for the possibility of undefined propositional identifiers. However, the maps involving *Var*, *Fn*, and *Pr* are assumed to be total, that is, each used *Var* maps to an integer (since all integer variables are defined), each used *Fn* maps to a function definition, and each used *Pr* maps to a predicate definition). The function definitions (*Func*) and predicate definitions (*Pred*) themselves can be partial (not define a result for certain arguments), to allow for "gaps". It is assumed that *Predicate*s have at least an arity of 1.

All functions and predicates are considered to be strict, that is, if there is a "gap" in an argument then there is a "gap" in the result of applying a function/predicate with that argument. For example, given a function $f\colon \mathbb{Z} \to \mathbb{Z}$, if f is applied with a "gap" ($f(\bot_{\mathbb{Z}})$) then the result is a "gap". This is illustrated in the following case analysis:

$$f(a) = \{ \begin{array}{l} f(a) \text{ if } f(a) \text{ is defined and } a \text{ is defined} \\ \bot_{\mathbb{Z}} \text{ if } f(a) \text{ is undefined and } a \text{ is defined} \\ \bot_{\mathbb{Z}} \text{ if } a \text{ is undefined.} \end{array}$$

Additionally, functions and predicates have a fixed arity in any given σ, and will always return the same result when given the same argument(s) in a given σ.

3.3.1 Big-Step Structural Operational Semantics Definition

The semantic (transition) relation used to model the process of expression evaluation is:

$\xrightarrow{e}\colon \mathcal{P}((\mathit{Expr} \times \Sigma) \times \mathit{Value})$

Notice that there is no undefined value, instead the treatment of undefinedness is as "gaps".

Consider the semantic rules for the evaluation of the disjunction logical operator:

$$\boxed{Or\_E1}\ \frac{(p, \sigma) \xrightarrow{e} \mathbf{true}}{(mk\_Or(p, q), \sigma) \xrightarrow{e} \mathbf{true}}$$

$$\boxed{Or\_E2}\ \frac{(q, \sigma) \xrightarrow{e} \mathbf{true}}{(mk\_Or(p, q), \sigma) \xrightarrow{e} \mathbf{true}}$$

$$\boxed{Or\_E3}\ \frac{\begin{array}{c}(p, \sigma) \xrightarrow{e} \mathbf{false}; \\ (q, \sigma) \xrightarrow{e} \mathbf{false}\end{array}}{(mk\_Or(p, q), \sigma) \xrightarrow{e} \mathbf{false}}$$

Remember that the truth tables in LPF can be regarded as describing a parallel lazy evaluation of the logical operators. The semantic rules above do not allow for this. Notice that the evaluation can get *stuck* in evaluating an operand (c.f. the Or_E1 semantic rule and the Or_E2 semantic rule). As a result if the evaluation starts with evaluating a non-denoting operand then the evaluation process can get stuck. It could be that the other operand is defined and this operand alone could determine the result but this other operand may not be given the chance to run.

To further highlight the problems that defining a big-step semantics could cause when specifying the semantics of LPF, consider the semantic rule which defines the evaluation of the function call expression and thus illustrates one of the places where "gaps" can arise in the first place:

$$\boxed{\mathit{FuncCall\_E}}\ \frac{\begin{array}{c}\mathbf{let}\ a = [\mathit{args}'(i) \mid i\colon \mathbf{inds}\ \mathit{args} \wedge \\ (\mathit{args}(i), \sigma) \xrightarrow{e} \mathit{args}'(i) \wedge \mathit{args}'(i) \in \mathbb{Z}]\ \mathbf{in} \\ \mathbf{len}\ \mathit{args} = \mathbf{len}\ a; \\ (\sigma(\mathit{id}).\mathit{result}, \sigma \dagger \{\sigma(\mathit{id}).\mathit{params}(i) \mapsto a(i) \mid \\ i\colon \mathbf{inds}\ \sigma(\mathit{id}).\mathit{params}\}) \xrightarrow{e} \mathit{res}; \\ \mathit{res} \in \mathbb{Z}\end{array}}{(\mathit{mk\_FuncCall}(\mathit{id}, \mathit{args}), \sigma) \xrightarrow{e} \mathit{res}}$$

where $a \in \mathbb{Z}$ is to check that a is a constant integer value, e.g. $0 \in \mathbb{Z}$ is true, but $\mathit{mk\_Arith}(1, +, 1) \in \mathbb{Z}$ is false, as $\mathit{mk\_Arith}(1, +, 1)$ has not yet been evaluated to a constant value. Additionally, $\sigma(\mathit{id})$ is used to retrieve a function definition (which is represented as a *Func* record) from the given σ map corresponding to the function name id. The trailing *.result* and *.params* are used to retrieve the selected data from the function definition in question. The abstract syntax ensures that $\mathit{id} \in \mathit{Fn}$.

All functions (and predicates) are strict, so the evaluation strategy that is used is *call by value*, that is, the argument expressions are evaluated first and then their resulting results/values (if there is no "gap" in any argument) are then bound to the corresponding parameter variables in the function, by updating the given memory store σ (temporarily updated, just during the evaluation of that statement in a transition rule). No permanent change to σ is made as σ is not present on the right hand side of the $\xrightarrow{e}$ semantic transition. Thus a "gap" in an argument passed into a function causes a "gap" in the function call term to occur, even if the function makes no use of the argument in the functions defined result expression.

An operand to the disjunction operator could essentially be a "gap" (it will not denote). For instance, the operand could contain a function call expression, which for the given arguments could result in no result (a constant value *res*) being returned; the function may not yield a result for such arguments, or at least one of the arguments could be undefined (a "gap").

Since the purpose of this SOS specification is to model the process of expression evaluation according to the semantics of LPF, a small-step semantics is the preferred way of defining an SOS specification to precisely define, and to precisely illustrate the semantics of LPF. In fact a small-step SOS definition is needed to be able to define the expression evaluation process in a way that is faithful to the semantics of LPF. Recall that the truth tables in LPF can be viewed as describing a parallel lazy evaluation of the logical operators. The small-step SOS allows for more execution details to be presented, since the big-step SOS definition is more abstract and denotational in nature than a small-step SOS definition, resulting in the need for fewer semantic rules/transition relations to define the expression evaluation process for LPF.

A small-step semantics definition allows for the interleaving of steps in different expression branches as can be seen for the semantic rules for the arithmetic expressions and for the semantic rules for the disjunction logical operator among others that are presented in Section 3.3.2. It is important to use a small-step semantics definition as interleaving is required for logical operators such as disjunction since they have to cope with the "gaps" that can occur. If a "gap" operand starts to be evaluated the other operand, which could be defined and thus could in fact determine the overall result of the evaluation, needs to be given a chance to be evaluated. This point will be further discussed when the semantic rules for the disjunction logical operator are introduced in Section 3.3.2.

The full set of semantic rules (both big-step and small-step) which model the process of expression evaluation in LPF are presented in Appendix A.

3.3.2 Small-Step Structural Operational Semantics Definition

In this semantic definition the emphasis is on the individual steps that take place during evaluating an expression. In the big-step semantics the evaluation of an expression e with respect to a σ either returns a *Value* (e is fully evaluated), or the evaluation is stuck (that is, there is no v such that $(e, \sigma) \xrightarrow{e} v$) and thus no result can be returned. In the small-step semantics after executing a transition rule there are three possible outcomes, the evaluation of e is not complete (there is an intermediate expression to evaluate), or the expression evaluation process is stuck as before, or the evaluation has completed (that is, has returned a *Value*).

The semantic rules that follow are all based upon a small-step semantics definition unless otherwise stated. The semantic (transition) relation used to model the process of expression evaluation is:

$$\xrightarrow{e} : \mathcal{P}((Expr \times \Sigma) \times Expr)$$

Nowhere in the semantic definitions presented will a given σ ($\sigma \in \Sigma$) be changed as a result of applying a transition rule. This is the reason for there being no Σ present on the right-hand side of the semantic relation $\xrightarrow{e}$. The presence of Σ only on the left of the $\xrightarrow{e}$ semantic relation illustrates that there is no notion of side-effects that can change a given memory store. By not including Σ on the right-hand side of the semantic relation, information (a σ) that is not changed by a semantic rule is not repeated.

However, this does pose a problem that needs resolving before any small-step SOS rules can be presented, that is, that a conventional transitive closure cannot be used. Performing a transition provides a resulting expression after applying the one single expression evaluation step, which may not be the final value that should be obtained from evaluating an expression. The left-hand side of the semantic relation $\xrightarrow{e}$ is: $(\mathit{Expr} \times \Sigma)$, while the right-hand side of the semantic relation $\xrightarrow{e}$ is: Expr. Thus after a rule has been applied the term from the right-hand side will no longer match the term on the left-hand side. The two sides of the semantic relation $\xrightarrow{e}$ do not match, and therefore the application of several $\xrightarrow{e}$ transitions cannot be concatenated by a conventional transitive closure.

One solution to this problem is to use a semantic relation $\xrightarrow{tc}$:

$$\xrightarrow{tc} : \mathcal{P}((\mathit{Expr} \times \Sigma) \times (\mathit{Expr} \times \Sigma))$$

but this approach is not favoured for the reason already given above since information that is not changed by a semantic transition rule is repeated.

The preferred approach and the approach taken from here on is to use the semantic relation $\xrightarrow{e}$, but to define in addition a semantic relation $\xrightarrow{E}$ that is the reflexive, transitive closure of $\xrightarrow{e}$. There are two cases to consider for such a semantic relation $\xrightarrow{E}$. A base case for the one evaluation step, and a step case to allow for intermediate steps to be made during the expression evaluation process, where $e \in \mathit{Expr}$, and $v \in \mathit{Value}$:

$$(e, \sigma) \xrightarrow{E} v \;\Leftrightarrow\; e = v \vee \exists e' : \mathit{Expr} \cdot (e, \sigma) \xrightarrow{e} e' \wedge (e', \sigma) \xrightarrow{E} v$$

Note that both $\xrightarrow{e}$ and $\xrightarrow{E}$ are needed as without defining this reflexive, transitive closure with $\xrightarrow{E}$ in certain places an infinite rewrite can occur.

The first semantic rule is for constant expressions:

$$\boxed{\mathit{Value\_E}}\;\frac{v \in \mathit{Value}}{(v, \sigma) \xrightarrow{e} v}$$

where since a constant expression cannot be evaluated anymore, no change is made to the constant expression.

The next set of semantic rules simply returns the value to which a variable identifier is mapped in a given memory store:

$$\boxed{Prop\_E}\frac{id \in Prop;\quad id \in \mathbf{dom}\,\sigma}{(id, \sigma) \xrightarrow{e} \sigma(id)}$$

$$\boxed{Var\_E}\frac{id \in Var}{(id, \sigma) \xrightarrow{e} \sigma(id)}$$

informally the *Prop_E* rule states that given the expression *id* then the expression *id* (with respect to a memory store σ) can be evaluated to (replaced with) its associated value within the given memory store σ, if *id* is a propositional variable, and it is contained within the memory store ($id \in \mathbf{dom}\,\sigma$) as propositional identifiers can be missing from particular memory stores, to allow for undefined propositional identifiers to be present.

The *Var_E* semantic rule does not need the $id \in \mathbf{dom}\,\sigma$ restriction as a *Var* map for every $\sigma \in \Sigma$ is assumed to be total, that is, that all integer variables denote.

The next set of semantic rules to be presented are those which define the evaluation of arithmetic expressions. Notice that the operands a and b must be evaluated as much as possible (both need evaluating to constant values) before a result can be computed, i.e. eliminating the arithmetic operator from the given expression. The choice of which rule is evaluated is non-deterministic; there is no notion of fairness in the SOS rules:

$$\boxed{Arith\_L}\frac{(a, \sigma) \xrightarrow{e} a'}{(mk\_Arith(a, op, b), \sigma) \xrightarrow{e} mk\_Arith(a', op, b)}$$

$$\boxed{Arith\_R}\frac{(b, \sigma) \xrightarrow{e} b'}{(mk\_Arith(a, op, b), \sigma) \xrightarrow{e} mk\_Arith(a, op, b')}$$

$$\boxed{Arith\_E1}\frac{a \in \mathbb{Z};\quad b \in \mathbb{Z}}{(mk\_Arith(a, +, b), \sigma) \xrightarrow{e} [\![+]\!](a, b)}$$

$\sigma = \{x \mapsto 3\}$
$(mk\_Arith(x, -, x), \sigma) \xrightarrow{e} mk\_Arith(3, -, x)$ — *Arith_L*, *Var_E*
$(mk\_Arith(3, -, x), \sigma) \xrightarrow{e} mk\_Arith(3, -, 3)$ — *Arith_R*, *Var_E*
$(mk\_Arith(3, -, 3), \sigma) \xrightarrow{e} 0$ — *Arith_E2*
thus:
$(mk\_Arith(x, -, x), \sigma) \xrightarrow{E} 0$

Figure 3.5: A sample small-step SOS expression evaluation

$$\boxed{Arith\_E2}\ \frac{a \in \mathbb{Z};\quad b \in \mathbb{Z}}{(mk\_Arith(a, -, b), \sigma) \xrightarrow{e} [\![-]\!](a, b)}$$

$$\boxed{Arith\_E3}\ \frac{a \in \mathbb{Z};\quad b \in \mathbb{Z}}{(mk\_Arith(a, \times, b), \sigma) \xrightarrow{e} [\![\times]\!](a, b)}$$

$$\boxed{Arith\_E4}\ \frac{a \in \mathbb{Z};\quad b \in \mathbb{Z};\quad b \neq 0}{(mk\_Arith(a, \div, b), \sigma) \xrightarrow{e} [\![\div]\!](a, b)}$$

where $[\![op]\!](a, b)$ is to be regarded as the standard mathematical result of the specified operator op applied to two given operands a and b.

Partial terms arise from arithmetic expressions that reduce to something of the form $mk\_Arith(i, \div, 0)$, so the *Arith_E4* semantic rule is one of the places that gives rise to "gaps" in this SOS definition.

As an illustration of how expressions are evaluated in this language a simple illustrative example is presented in Figure 3.5. In this example the expression being evaluated is $mk\_Arith(x, -, x)$, with a (global) memory store (σ) containing the *Var* x mapped to the value 3. Another illustration is presented in Figure 3.6 which illustrates how "gaps" can arise and how they are represented in this language; the evaluation becomes stuck as shown in Figure 3.6.

The following set of semantic rules are used to define weak/strict equality. Such a notion of equality is defined to return a result only if both operands

$\sigma = \{x \mapsto 3\}$
$(mk\text{\_}Arith(x, \div, 0), \sigma) \xrightarrow{e} mk\text{\_}Arith(3, \div, 0)$ *Arith_L*, *Var_E*
$(mk\text{\_}Arith(3, \div, 0), \sigma) \xrightarrow{e}$ *No more rules can be applied leaving a "gap".*

Figure 3.6: A sample small-step SOS "gap" expression evaluation

denote values, that is, in this language both operands denote integer values. If both operands do not denote values then the given equality expression will also not denote a (defined) value:

$$\boxed{Equality\text{\_}L}\ \frac{(a, \sigma) \xrightarrow{e} a'}{(mk\text{\_}Equality(a, b), \sigma) \xrightarrow{e} mk\text{\_}Equality(a', b)}$$

$$\boxed{Equality\text{\_}R}\ \frac{(b, \sigma) \xrightarrow{e} b'}{(mk\text{\_}Equality(a, b), \sigma) \xrightarrow{e} mk\text{\_}Equality(a, b')}$$

$$\boxed{Equality\text{\_}E}\ \frac{a \in \mathbb{Z};\quad b \in \mathbb{Z}}{(mk\text{\_}Equality(a, b), \sigma) \xrightarrow{e} [\![=]\!](a, b)}$$

the reader should notice how partial terms that are operands to such weak relational operators can lead to a non-denoting truth value.

The set of semantic rules for the conditional expression follows:

$$\boxed{Cond\text{\_}A}\ \frac{(p, \sigma) \xrightarrow{e} p'}{(mk\text{\_}Cond(p, a, b), \sigma) \xrightarrow{e} mk\text{\_}Cond(p', a, b)}$$

$$\boxed{Cond\text{\_}E1}\ \frac{}{(mk\text{\_}Cond(\mathbf{true}, a, b), \sigma) \xrightarrow{e} a}$$

$$\boxed{Cond\text{\_}E2}\ \frac{}{(mk\text{\_}Cond(\mathbf{false}, a, b), \sigma) \xrightarrow{e} b}$$

where the *Cond_A* semantic rule describes the small-step semantics for evaluating the condition expression in the conditional expression construct. If this condition expression can be evaluated to a Boolean value (the expression is defined), then one of two elimination semantic rules ($Cond\_E1$ or $Cond\_E2$) can be applied. Either simply replaces the conditional expression construct with the appropriate sub-expression (a or b). The interpretation of the conditional expression construct is undefined if the condition expression is undefined, even if both of the sub-expressions a and b evaluate to the same value.

Attention is now turned to defining the evaluation of the logical operators, starting with the negation logical operator. If the operand expression p can be evaluated to a constant Boolean value then it is inverted:

$$\boxed{Not\_A}\ \frac{(p, \sigma) \xrightarrow{e} p'}{(mk\_Not(p), \sigma) \xrightarrow{e} mk\_Not(p')}$$

$$\boxed{Not\_E1}\ \frac{}{(mk\_Not(\mathbf{true}), \sigma) \xrightarrow{e} \mathbf{false}}$$

$$\boxed{Not\_E2}\ \frac{}{(mk\_Not(\mathbf{false}), \sigma) \xrightarrow{e} \mathbf{true}}$$

The definedness operator (δ), as mentioned earlier, must return **true** only if its argument is defined. For instance, given $\delta(p)$, if p can be evaluated to **true** or to **false**, then return **true** as p is defined, otherwise p is non-denoting. This is illustrated in the following set of semantic rules:

$$\boxed{delta\_A}\ \frac{(p, \sigma) \xrightarrow{e} p'}{(mk\_delta(p), \sigma) \xrightarrow{e} mk\_delta(p')}$$

$$\boxed{delta\_E1}\ \frac{}{(mk\_delta(\mathbf{true}), \sigma) \xrightarrow{e} \mathbf{true}}$$

$$\boxed{delta\_E2}\ \frac{}{(mk\_delta(\mathbf{false}), \sigma) \xrightarrow{e} \mathbf{true}}$$

Because of the way that expressions are being evaluated in this semantic definition this rule for δ is exactly the same as the rule that would be provided for Δ; the denotational semantics introduces Δ into *Expr* and illustrates how the truth value false can be returned.

The idea behind providing a small-step semantics is to allow for interleaving of steps in different expression branches since in LPF a result can be returned even in the presence of "gaps" in operands, as long as there is enough information available from evaluating the other operand. For example, $mk\_Or(p, \mathbf{true})$ can be evaluated to true even though the first operand has not been fully evaluated; it could be that this operand could be fully evaluated, or that this operand will fail to denote a value with respect to a given σ.

Considering the first operand of the previous example as containing a term that will never denote a proper (i.e. constant) value (for example arising from a function call, e.g. $mk\_Equality(mk\_FuncCall(zero, [-1]), 0)$), without such interleaving in expression branches being able to occur then the evaluation of this operand could start, and with a big-step semantics the evaluation will not stop without this operand being evaluated to a constant Boolean value (which it will never denote). Thus in the big-step SOS definition evaluating an expression of the form of $mk\_Or(\bot_{\mathbb{B}}, \mathbf{true})$ can get stuck, and not return the constant Boolean value true as would be expected according to the semantics of LPF.

The following set of semantic rules illustrates the evaluation of the disjunction logical operator according to the truth table presented in Figure 2.7.

$$\boxed{Or\_L}\ \frac{(p, \sigma) \xrightarrow{e} p'}{(mk\_Or(p, q), \sigma) \xrightarrow{e} mk\_Or(p', q)}$$

$$\boxed{Or\_R}\ \frac{(q, \sigma) \xrightarrow{e} q'}{(mk\_Or(p, q), \sigma) \xrightarrow{e} mk\_Or(p, q')}$$

$$\boxed{Or\_E1}\ \frac{}{(mk\_Or(\mathbf{true}, q), \sigma) \xrightarrow{e} \mathbf{true}}$$

$$\boxed{Or\_E2}\ \frac{}{(mk\_Or(p, \mathbf{true}), \sigma) \xrightarrow{e} \mathbf{true}}$$

$$\boxed{Or\_E3}\ \frac{}{(mk\_Or(\mathbf{false}, \mathbf{false}), \sigma) \xrightarrow{e} \mathbf{false}}$$

The two rules Or_E1 and Or_E2 can be seen as "coping with gaps" since they are able to return a value even if one of their operands fails to denote.

The choice of which rule is used is non-deterministic; there is no control over which rule is used. Ideally when evaluating a disjunction expression each operand would be evaluated in parallel, and then an elimination rule would be used to return a result once enough information is available from at least one evaluated operand. Alternatively, this parallel evaluation is simulated by performing the one evaluation step on the left hand operand and then the one evaluation step on the right hand operand, iterating this process until enough information is available for an elimination rule to be applied (to complete the evaluation of a disjunction expression — if an elimination rule can ever be applied).

The fact that there is no control over when and what semantic rule is evaluated could be problematic. The left hand operand may always be "chosen" for evaluation and never the right hand operand. Alternatively the left hand operand could be evaluated to **true** and then the right hand operand could be "chosen" to be evaluated continuously (with multiple applications of the semantic rule), and this right hand operand may not denote (see the semantic rules for the function call expression later), and thus the disjunction expression may never denote a Boolean value. Additionally there are other similar evaluations that are possible with these rules that could cause no result to be returned even if a result could be expected to be returned according to the semantics of LPF. An internal rewriting strategy could be used to control the rewriting process.

The next set of semantic rules sees the move from only coping with "gaps" in the propositional calculus to the inclusion of quantified expressions. Here the quantification semantic rules are first defined using the $\xrightarrow{E}$ semantic relation. For the following semantic rule, it is necessary that for one integer i which when applied to the expression e causes e to evaluate to true. In particular true can even be returned if the quantified expression e fails to denote with certain values of i; clearly the choice of the value for i is important:

$$\boxed{\mathit{Exists\_E1}}\ \frac{\exists i\colon \mathbb{Z} \cdot (p, \sigma \dagger \{x \mapsto i\}) \xrightarrow{E} \textbf{true}}{(\mathit{mk\_Exists}(x, p), \sigma) \xrightarrow{e} \textbf{true}}$$

The false case is expressed in the following rule, where the expression e for every integer i must evaluate to false:

$$\boxed{\mathit{Exists\_E2}}\ \frac{\forall i\colon \mathbb{Z} \cdot (p, \sigma \dagger \{x \mapsto i\}) \xrightarrow{E} \textbf{false}}{(\mathit{mk\_Exists}(x, p), \sigma) \xrightarrow{e} \textbf{false}}$$

At a first glance at these semantic rules it becomes clear that quantifiers are being used to define the existential quantifier in this language. The existential quantifier semantic rules contain infinitely many premises since the quantification is performed over the set of integers. In the case of proof systems this is referred to as *semi-formal*. While this is fine for one's intuition, this is not acceptable because if the meta-language interpretation of the quantifiers changes then so does the implied semantics. This core issue will be resolved soon enough. For now, think of the use of the existential quantifier above the line in the $Exists\_E1$ semantic rule as shorthand for an infinite disjunction (using the LPF disjunction logical operator already introduced), and the use of the universal quantifier above the line in the $Exists\_E2$ semantic rule as shorthand for an infinite conjunction, both over the set of integers.

Alternatively, the semantic rules to define the existential quantifier can be expressed differently. *Expr* is first extended to:

$$Expr = \ldots \mid ExistsInter$$

$$\begin{array}{lrcl} ExistsInter :: & x & : & Id \\ & pairs & : & ExistsPair^* \end{array}$$

where:

$$\begin{array}{lrcl} ExistsPair :: & i & : & \mathbb{Z} \\ & p & : & Expr \end{array}$$

A context condition is not included since this additional expression construct (and the *FuncInter* expression construct introduced later) is only to be formed through the application of a semantic rule.

The first existential quantifier rule creates an *ExistsInter* expression:

$$\boxed{Exists\_E}\frac{}{\begin{array}{c}(mk\_Exists(x,p),\sigma) \xrightarrow{e} \\ mk\_ExistsInter(x,[mk\_ExistsPair(i,p) \mid i\colon\mathbb{Z}])\end{array}}$$

An expression evaluation step can now be made for an arbitrary integer value, where the let expression makes an arbitrary choice here of a valid sequence index:

$$\boxed{ExistsInter\_A}\frac{\begin{array}{c}\textbf{let } j \in \textbf{inds}\, pairs \textbf{ in} \\ (pairs(j).p, \sigma \dagger \{x \mapsto pairs(j).i\}) \xrightarrow{e} pairs'(j).p\end{array}}{(mk\_ExistsInter(x, pairs), \sigma) \xrightarrow{e} mk\_ExistsInter(x, pairs')}$$

where *pairs'* is *pairs* but incorporating the change made to the jth element: $pairs'(j).p$.

The final two semantic rules for the existential quantifier return a result if enough information is available:

$$\boxed{\mathit{ExistsInter\_E1}}\;\frac{\mathbf{true} \in \{\mathit{pairs}(i).p \mid i\colon \mathbf{inds}\, \mathit{pairs}\}}{(\mathit{mk\_ExistsInter}(x, \mathit{pairs}), \sigma) \xrightarrow{e} \mathbf{true}}$$

$$\boxed{\mathit{ExistsInter\_E2}}\;\frac{\{\mathit{pairs}(i).p \mid i\colon \mathbf{inds}\, \mathit{pairs}\} = \{\mathbf{false}\}}{(\mathit{mk\_ExistsInter}(x, \mathit{pairs}), \sigma) \xrightarrow{e} \mathbf{false}}$$

Notice that the existential quantifier can give rise to a “gap”.

These semantic rules express the process of quantified expression evaluation according to the semantics of LPF, despite the fact that infinitely many premises exist.

Until this point “gaps” have only been introduced into the language through a propositional variable identifier being absent from the domain of a given σ, or through applying the division operator in an obvious way. The next set of semantic rules allows for another way of “gaps” being introduced through the function call expression construct.

The following semantic rule represents the small-step semantics for evaluating the argument expressions to be passed into the function being invoked. This rule is to be utilised until the argument expressions have all been reduced to a constant value. Any argument used in a function call must denote otherwise the function is not evaluated, and thus a function call expression is a “gap”, that is, if an argument to a function is undefined, then the function's result is undefined. Here an arbitrary argument is selected for an evaluation step:

$$\boxed{\mathit{FuncCall\_A}}\;\frac{\mathbf{let}\ i \in \mathbf{inds}\, \mathit{args}\ \mathbf{in}\ (\mathit{args}(i), \sigma) \xrightarrow{e} \mathit{args}'(i)}{(\mathit{mk\_FuncCall}(\mathit{id}, \mathit{args}), \sigma) \xrightarrow{e} \mathit{mk\_FuncCall}(\mathit{id}, \mathit{args}')}$$

Another expression construct is included in the language here in order to define a small-step semantics for evaluating the result of a function:

$\mathit{Expr} = \ldots \mid \mathit{FuncInter}$

$$\begin{array}{lrcl} \mathit{FuncInter} :: & \mathit{result} & : & \mathit{Expr} \\ & \mathit{paramid} & : & \mathit{Var}^* \\ & \mathit{args} & : & \mathit{Expr}^* \end{array}$$

A *FuncInter* expression construct is used to represent a function call expression that is currently under evaluation. The data stored in a *FuncInter* expression comprises of information belonging to a given function call expression (the argument expressions) and information belonging to the function being called, the function's result expression and the function's parameter identifiers.

Once (if) all of the argument expressions have been evaluated to a constant value (they are defined) attempts can then be made to evaluate the function's result expression. Since the semantics are to allow for the possibility of interleaving of steps in different expression branches the *FuncCall_E* semantic rule that follows first creates a *FuncInter* expression to allow for this possibility:

$$\boxed{\mathit{FuncCall\_E}}\ \frac{[\mathit{args}(i) \mid i\colon \mathbf{inds}\,\mathit{args} \wedge \mathit{args}(i) \in \mathbb{Z}] = \mathit{args}}{(\mathit{mk\_FuncCall}(\mathit{id}, \mathit{args}), \sigma) \xrightarrow{e} \mathit{mk\_FuncInter}(\sigma(\mathit{id}).\mathit{result}, \sigma(\mathit{id}).\mathit{params}, \mathit{args})}$$

The next semantic rule is used to make a (further) step in evaluating a functions result, which is now represented through a *FuncInter* expression construct, each time it is applied:

$$\boxed{\mathit{FuncInter\_A}}\ \frac{(\mathit{res}, \sigma \dagger \{\mathit{paramids}(i) \mapsto \mathit{args}(i) \mid i\colon \mathbf{inds}\,\mathit{paramids}\}) \xrightarrow{e} \mathit{res}'}{(\mathit{mk\_FuncInter}(\mathit{res}, \mathit{paramids}, \mathit{args}), \sigma) \xrightarrow{e} \mathit{mk\_FuncInter}(\mathit{res}', \mathit{paramids}, \mathit{args})}$$

notice that the parameter is included in the memory store (σ) during the evaluation of the function's result expression, but that the updated memory store is not returned by the semantic rule. After the one evaluation step has been made through an application of the *FuncInter_A* semantic rule the update made to the given memory store σ is effectively undone. Only the updated result expression along with the parameter information to (possibly) be used to update the memory store σ in the same way later is returned by this semantic rule (in the form of a *FuncInter* expression construct). This is to achieve the necessary variable scoping since interleaving of steps in different expression branches is allowed and is necessary to define the semantics of LPF precisely.

The final function application semantic rule (*FuncInter_E*) returns the result of a function call expression once (if) it has been evaluated to an integer value:

$$\boxed{\mathit{FuncInter\_E}}\ \frac{\mathit{res} \in \mathbb{Z}}{(\mathit{mk\_FuncInter}(\mathit{res}, \mathit{paramids}, \mathit{args}), \sigma) \xrightarrow{e} \mathit{res}}$$

The purpose of using the *FuncInter* expression construct is to allow for the current state of the result to be stored (alongside the parameter data) so that the evaluation of a functions result can resume from where it left off previously if any interleaving of the steps in expression branches occurs. This is to be able to provide a small-step semantics, so for instance given an expression such as:

$$\begin{aligned}&\mathit{mk\_Or}(\mathit{mk\_Equality}(\mathit{mk\_FuncCall}(\mathit{zero}, [1]), 0),\\ &\quad \mathit{mk\_Equality}(\mathit{mk\_FuncCall}(\mathit{zero}, [-1]), 0))\end{aligned}$$

where *zero* is defined in σ as the partial function considered earlier. If evaluation starts on the second non-denoting operand of the disjunction operator, one step can be made in evaluating this operand using the small-step function semantic rules presented above. After that one evaluation step has been performed it is then possible for the other (denoting in this case) operand to be evaluated, and thus a result (true in this case) could eventually be returned. This may not be possible with the original *FuncCall_E* semantic rule which made a big step in evaluating the result of a function, if the non-denoting operand is chosen to be evaluated first.

The rules for evaluating a predicate call expression are similar to the rules provided for evaluating the function call expression. These extra semantic rules are documented in Appendix A, where a full list of the big-step SOS rules, and the small-step SOS rules are presented.

To illustrate how an expression containing a function call term is evaluated in this language consider a sample evaluation of the expression:

$$\mathit{zero}(1) = 0 \vee \mathit{zero}(-1) = 0$$

where the evaluation is presented using the small-step semantic rules introduced earlier, but using a concrete syntax in places to fit on a page. The given σ contains only the one maplet, and that is the maplet for the *zero* function:

$$\sigma = \{\mathit{zero} \mapsto \mathit{mk\_Func}([i], i = 0\ ?\ 0 :\ \mathit{zero}(i-1))\}$$

$\sigma = \{\mathit{zero} \mapsto \mathit{mk\_Func}([i], i = 0 \mathbin{?} 0 : \mathit{zero}(i-1))\}$

$(\mathit{zero}(1) = 0 \vee \mathit{zero}(-1) = 0, \sigma) \xrightarrow{e} \mathit{mk\_FuncInter}(i = 0 \mathbin{?} 0 : \mathit{zero}(i-1), [i], [1]) = 0 \vee \mathit{zero}(-1) = 0$ *Or_L, Equality_L, FuncCall_E*

$(\mathit{mk\_FuncInter}(i = 0 \mathbin{?} 0 : \mathit{zero}(i-1), [i], [1]) = 0 \vee \mathit{zero}(-1) = 0, \sigma) \xrightarrow{e}$

$\quad \mathit{mk\_FuncInter}(1 = 0 \mathbin{?} 0 : \mathit{zero}(i-1), [i], [1]) = 0 \vee \mathit{zero}(-1) = 0$ *Or_L, Equality_L, FuncInter_A, Cond_A, Equality_L, Var_E*

$(\mathit{mk\_FuncInter}(1 = 0 \mathbin{?} 0 : \mathit{zero}(i-1), [i], [1]) = 0 \vee \mathit{zero}(-1) = 0, \sigma) \xrightarrow{e}$

$\quad \mathit{mk\_FuncInter}(\mathbf{false} \mathbin{?} 0 : \mathit{zero}(i-1), [i], [1]) = 0 \vee \mathit{zero}(-1) = 0$ *Or_L, Equality_L, FuncInter_A, Cond_A, Equality_E*

$(\mathit{mk\_FuncInter}(\mathbf{false} \mathbin{?} 0 : \mathit{zero}(i-1), [i], [1]) = 0 \vee \mathit{zero}(-1) = 0, \sigma) \xrightarrow{e} \mathit{mk\_FuncInter}(\mathit{zero}(i-1), [i], [1]) = 0 \vee \mathit{zero}(-1) = 0$ *Or_L, Equality_L, FuncInter_A, Cond_E2*

$(\mathit{mk\_FuncInter}(\mathit{zero}(i-1), [i], [1]) = 0 \vee \mathit{zero}(-1) = 0, \sigma) \xrightarrow{e}$

$\quad \mathit{mk\_FuncInter}(\mathit{zero}(1-1), [i], [1]) = 0 \vee \mathit{zero}(-1) = 0$ *Or_L, Equality_L, FuncInter_A, FuncCall_A, Arith_L, Var_E*

$(\mathit{mk\_FuncInter}(\mathit{zero}(1-1), [i], [1]) = 0 \vee \mathit{zero}(-1) = 0, \sigma) \xrightarrow{e} \mathit{mk\_FuncInter}(\mathit{zero}(0), [i], [1]) = 0 \vee \mathit{zero}(-1) = 0$ *Or_L, Equality_L, FuncInter_A, FuncCall_A, Arith_E2*

$(\mathit{mk\_FuncInter}(\mathit{zero}(0), [i], [1]) = 0 \vee \mathit{zero}(-1) = 0, \sigma) \xrightarrow{e}$

$\quad \mathit{mk\_FuncInter}(\mathit{mk\_FuncInter}(i = 0 \mathbin{?} 0 : \mathit{zero}(i-1), [i], [0]), [i], [1]) = 0 \vee \mathit{zero}(-1) = 0$ *Or_L, Equality_L, FuncInter_A, FuncCall_E*

$(\mathit{mk\_FuncInter}(\mathit{mk\_FuncInter}(i = 0 \mathbin{?} 0 : \mathit{zero}(i-1), [i], [0]), [i], [1]) = 0 \vee \mathit{zero}(-1) = 0, \sigma) \xrightarrow{e}$

$\quad \mathit{mk\_FuncInter}(\mathit{mk\_FuncInter}(0 = 0 \mathbin{?} 0 : \mathit{zero}(i-1), [i], [0]), [i], [1]) = 0 \vee \mathit{zero}(-1) = 0$ *Or_L, Equality_L, FuncInter_A, FuncInter_A, Cond_A, Equality_L, Var_E*

$(\mathit{mk\_FuncInter}(\mathit{mk\_FuncInter}(0 = 0 \mathbin{?} 0 : \mathit{zero}(i-1), [i], [0]), [i], [1]) = 0 \vee \mathit{zero}(-1) = 0, \sigma) \xrightarrow{e}$

$\quad \mathit{mk\_FuncInter}(\mathit{mk\_FuncInter}(\mathbf{true} \mathbin{?} 0 : \mathit{zero}(i-1), [i], [0]), [i], [1]) = 0 \vee \mathit{zero}(-1) = 0$ *Or_L, Equality_L, FuncInter_A, FuncInter_A, Cond_A, Equality_E*

$(\mathit{mk\_FuncInter}(\mathit{mk\_FuncInter}(\mathbf{true} \mathbin{?} 0 : \mathit{zero}(i-1), [i], [0]), [i], [1]) = 0 \vee \mathit{zero}(-1) = 0, \sigma) \xrightarrow{e}$

$\quad \mathit{mk\_FuncInter}(\mathit{mk\_FuncInter}(0, [i], [0]), [i], [1]) = 0 \vee \mathit{zero}(-1) = 0$ *Or_L, Equality_L, FuncInter_A, FuncInter_A, Cond_E1*

$(\mathit{mk\_FuncInter}(\mathit{mk\_FuncInter}(0, [i], [0]), [i], [1]) = 0 \vee \mathit{zero}(-1) = 0, \sigma) \xrightarrow{e} \mathit{mk\_FuncInter}(0, [i], [1]) = 0 \vee \mathit{zero}(-1) = 0$ *Or_L, Equality_L, FuncInter_A, FuncInter_E*

$(\mathit{mk\_FuncInter}(0, [i], [1]) = 0 \vee \mathit{zero}(-1) = 0, \sigma) \xrightarrow{e} 0 = 0 \vee \mathit{zero}(-1) = 0$ *Or_L, Equality_L, FuncInter_E*

$(0 = 0 \vee \mathit{zero}(-1) = 0, \sigma) \xrightarrow{e} \mathbf{true} \vee \mathit{zero}(-1) = 0$ *Or_L, Equality_E*

$(\mathbf{true} \vee \mathit{zero}(-1) = 0, \sigma) \xrightarrow{e} \mathbf{true}$ *Or_E1*

thus:

$(\mathit{zero}(1) = 0 \vee \mathit{zero}(-1) = 0, \sigma) \xrightarrow{E} \mathbf{true}$

Figure 3.7: A further sample small-step SOS expression evaluation

3.4 Denotational Semantics

This section carries the intuition of the SOS definitions over to DS definitions by providing a set theoretic definition of the values that are denoted by expressions.

Both DS definitions are defined compositionally, that is, a relation is provided for each base element, and for each composite element the relation is defined in terms of applying the relation to the sub-parts of the composite elements [NN92]. Each DS definition describes the effect of executing each of the available expression constructs according to the semantics of LPF. The DS definitions are more abstract than the small-step SOS definitions because the internal evaluation process is not what is of interest here, but rather the values produced by the expression constructs. Section 3.5 shows that the two DS definitions are equivalent and shows their relationship with the SOS definitions. The DS definitions are useful for proving properties of programs, and are in fact used to perform such a task in Chapter 6.

The semantic relation $\mathcal{E}$ is defined as:

$$\mathcal{E} \colon \mathcal{P}((Expr^{\Delta} \times \Sigma) \times Value)$$

where $Expr^{\Delta}$ is:

$$Expr^{\Delta} = Expr \mid Delta$$

where:

$$Delta \;::\; p \;:\; Expr$$

and the context condition for *Delta* is the same as for *delta*.

The memory store Σ is now defined as:

$$\begin{aligned}\Sigma = {} & Prop \xrightarrow{m} \mathbb{B} \mid \\ & Var \xrightarrow{m} \mathbb{Z} \mid \\ & Fn \xrightarrow{m} Function \mid \\ & Pr \xrightarrow{m} Predicate\end{aligned}$$

notice that only the function (Fn) and the predicate (Pr) maps have been changed. The denotations of *Function* and *Predicate* are relations (set of pairs):

$$Function = \mathcal{P}(\mathbb{Z}^* \times \mathbb{Z})$$

$$Predicate = \mathcal{P}(\mathbb{Z}^* \times \mathbb{B})$$

where the function *Function* and predicate *Predicate* denotations themselves can be partial, that is, they may not yield a result for every member of their domain.

"Gaps" that arise are modelled by choosing relations as the space of denotations, instead of partial functions as is classical in denotational semantics. Notice that there is no undefined value. The treatment of undefinedness is as a "gap" in the denotation. How "gaps" arise and are coped with can be seen clearly in these denotational semantic definitions.

Recall that in a given σ a propositional identifier may be missing from the domain of σ and this represents that the propositional identifier is undefined. Each used *Var* will map to an integer value in a σ as all integer variables denote. Also in a given σ each used *Fn* will map to a function definition *Function*, but this *Function* may not contain a result for every member of the domain, allowing for partial functions (and similarly for predicates). Note that the values that a function will not yield a defined value for are not contained within the relevant *Function* that is mapped to by a function identifier, (e.g. considering the *zero* function as defined earlier, $([0], 0) \in \sigma(\mathit{zero})$, but $([-1], 0) \notin \sigma(\mathit{zero})$). The defined domain of *Function*s does not need to be specified. There is no undefined value assigned to values from outside of the defined domain of *Function*s. The treatment is as a "gap".

$\mathcal{E}$ is defined in parts:

$$\mathcal{E} = \mathcal{E}\mathit{value} \cup \mathcal{E}\mathit{id} \cup \mathcal{E}\mathit{arith} \cup \mathcal{E}\mathit{equality} \cup \mathcal{E}\mathit{cond} \cup \mathcal{E}\mathit{or} \cup \mathcal{E}\mathit{not} \cup \mathcal{E}\mathit{delta} \cup \\ \mathcal{E}\mathit{Delta} \cup \mathcal{E}\mathit{exists} \cup \mathcal{E}\mathit{funccall} \cup \mathcal{E}\mathit{predcall}$$

A constant value (*Value*) cannot be reduced any further and is defined in any σ:

$$\mathcal{E}\mathit{value} = \\ \{((e, \sigma), e) \mid e \in \mathit{Value}\}$$

Accessing propositional variables and integer variables is defined as (remember that the *Var* map is total, while the *Prop* map can be partial in the sense that a propositional identifier can be absent from the domain of a specific map to allow for the possibility of undefined propositional identifiers):

$$\mathcal{E}\mathit{id} = \\ \{((v, \sigma), \sigma(v)) \mid v \in \mathit{Prop} \wedge v \in \mathbf{dom}\, \sigma\} \cup \\ \{((v, \sigma), \sigma(v)) \mid v \in \mathit{Var}\}$$

Arithmetic expressions:

$$\mathcal{E}\mathit{arith} = \\ \{((\mathit{mk\_Arith}(a, op, b), \sigma), [\![op]\!](a', b')) \mid \\ ((a, \sigma), a') \in \mathcal{E} \wedge ((b, \sigma), b') \in \mathcal{E} \wedge op \in \{+, -, \times\}\} \cup \\ \{((\mathit{mk\_Arith}(a, \div, b), \sigma), [\![\div]\!](a', b')) \mid \\ ((a, \sigma), a') \in \mathcal{E} \;\wedge\; ((b, \sigma), b') \in \mathcal{E} \;\wedge\; b' \neq 0\}$$

Weak equality:

$$\begin{array}{l}
\mathcal{E}\,\mathit{equality} = \\
\quad \{((\mathit{mk\_Equality}(a,b),\sigma), [\![=]\!](a',b')) \mid \\
\qquad ((a,\sigma),a') \in \mathcal{E} \wedge ((b,\sigma),b') \in \mathcal{E}\}
\end{array}$$

The conditional expression:

$$\begin{array}{l}
\mathcal{E}\,\mathit{cond} = \\
\quad \{((\mathit{mk\_Cond}(p,a,b),\sigma), a') \mid \\
\qquad ((p,\sigma),\mathbf{true}) \in \mathcal{E} \wedge ((a,\sigma),a') \in \mathcal{E}\} \cup \\
\quad \{((\mathit{mk\_Cond}(p,a,b),\sigma), b') \mid \\
\qquad ((p,\sigma),\mathbf{false}) \in \mathcal{E} \wedge ((b,\sigma),b') \in \mathcal{E}\}
\end{array}$$

The negation operator:

$$\begin{array}{l}
\mathcal{E}\,\mathit{not} = \\
\quad \{((\mathit{mk\_Not}(p),\sigma),\mathbf{false}) \mid ((p,\sigma),\mathbf{true}) \in \mathcal{E}\} \cup \\
\quad \{((\mathit{mk\_Not}(p),\sigma),\mathbf{true}) \mid ((p,\sigma),\mathbf{false}) \in \mathcal{E}\}
\end{array}$$

The definedness operator δ:

$$\begin{array}{l}
\mathcal{E}\,\mathit{delta} = \\
\quad \{((\mathit{mk\_delta}(p),\sigma),\mathbf{true}) \mid (p,\sigma) \in \mathbf{dom}\,\mathcal{E}\}
\end{array}$$

The definedness operator Δ:

$$\begin{array}{l}
\mathcal{E}\,\mathit{Delta} = \\
\quad \{((\mathit{mk\_Delta}(p),\sigma),\mathbf{true}) \mid \\
\qquad (p,\sigma) \in \mathbf{dom}\,\mathcal{E}\} \cup \\
\quad \{((\mathit{mk\_Delta}(p),\sigma),\mathbf{false}) \mid \\
\qquad (p,\sigma) \in (\{(p,\sigma) \mid \sigma \in \Sigma\} \setminus \{(p,\sigma) \mid (p,\sigma) \in \mathbf{dom}\,\mathcal{E}\})\}
\end{array}$$

The disjunction operator which can cope with "gaps" that can arise is defined as:

$$\begin{array}{l}
\mathcal{E}\,\mathit{or} = \\
\quad \{((\mathit{mk\_Or}(p,q),\sigma),\mathbf{true}) \mid ((p,\sigma),\mathbf{true}) \in \mathcal{E}\} \cup \\
\quad \{((\mathit{mk\_Or}(p,q),\sigma),\mathbf{true}) \mid ((q,\sigma),\mathbf{true}) \in \mathcal{E}\} \cup \\
\quad \{((\mathit{mk\_Or}(p,q),\sigma),\mathbf{false}) \mid ((p,\sigma),\mathbf{false}) \in \mathcal{E} \wedge ((q,\sigma),\mathbf{false}) \in \mathcal{E}\}
\end{array}$$

Notice that "gaps" are handled by non-denoting propositional expressions being absent from the domain of $\mathcal{E}$: The existential quantifier:

$$\begin{array}{l}
\mathcal{E}\,\mathit{exists} = \\
\quad \{((\mathit{mk\_Exists}(x,p),\sigma),\mathbf{true}) \mid \\
\qquad \mathbf{true} \in \mathbf{rng}\,(\{(p,\sigma \dagger \{x \mapsto i\}) \mid i{:}\,\mathbb{Z}\} \lhd \mathcal{E})\} \cup \\
\quad \{((\mathit{mk\_Exists}(x,p),\sigma),\mathbf{false}) \mid \\
\qquad \mathbf{rng}\,(\{(p,\sigma \dagger \{x \mapsto i\}) \mid i{:}\,\mathbb{Z}\} \lhd \mathcal{E}) = \{\mathbf{false}\}\}
\end{array}$$

The definitions of $\mathcal{E}funccall$ and $\mathcal{E}predcall$ are now considered.

The small-step SOS definition illustrates how logical formulae are evaluated which includes rules to compute the result of functions. But here in the DS definitions functions (and predicates) are relations where the result can be obtained by a lookup if such a result is defined.

Function call expressions are now defined as:

$$\begin{aligned}&\mathcal{E}funccall = \{((mk\_FuncCall(f, al), \sigma), res) \mid \\ &\quad \forall i{:}\,\mathbf{inds}\, al \cdot ((al(i), \sigma), vl(i)) \in \mathcal{E} \wedge \\ &\quad (vl, res) \in \sigma(f)\}\end{aligned}$$

where vl is the set of results (each $vl(i)$) from evaluating each $al(i)$. Notice that if an argument to be passed to the function is undefined, then the result of the function call is undefined.

The definition needed for $\mathcal{E}predcall$ is virtually the same as the definition of $\mathcal{E}funccall$.

This has achieved what was expected which is that for example for every $\sigma \in \Sigma$:

$$((mk\_Arith(1, \div, 1), \sigma), 1) \in \mathcal{E}$$

but:

$$(mk\_Arith(1, \div, 0), \sigma) \notin \mathbf{dom}\, \mathcal{E}$$

It is, however, interesting to note that the fixed point construction of the function/predicate denotations can be thought of as a bottom up construction of what is done in the SOS definitions by abandoning infinite expansion of the function/predicate calls.

Another version of $\mathcal{E}$ is defined in Figure 3.8, which is of the form:

$$\begin{aligned}&\mathcal{E} : Expr^{\Delta} \to \mathcal{P}(\Sigma \times Value)\\ &\mathcal{E}(e) \;\;\triangleq\;\; \ldots\end{aligned}$$

The denotational function $\mathcal{E}$ is a function from expressions to a set of memory stores and constant values (results), which assigns a meaning to each expression construct that is being considered. The $\mathcal{E}$ semantic function definition maps expressions to relations over interpretations and results. The $\mathcal{E}$ semantic function for each expression construct yields a set of pairs of σ and the corresponding result value, for every $\sigma \in \Sigma$ where the expression is defined. If under a particular σ the expression is undefined then a pair involving that σ will not appear in the returned set of σ and $Value$ pairs.

It is this second DS definition that is used throughout the paper from this point forward. The modified version of $\mathcal{E}$ makes for clearer more concise proofs, especially when considering logical equivalence. Section 3.5 shows that the two versions of $\mathcal{E}$ are equivalent.

Also in this definition as expected, for any $\sigma \in \Sigma$:

$$(\sigma, 1) \in \mathcal{E}(mk\_Arith(1, \div, 1))$$

but:

$$\sigma \notin \mathbf{dom}\, \mathcal{E}(mk\_Arith(1, \div, 0))$$

It is useful to record that the definition of any relation $\mathcal{E}(e)$ is deterministic (or "functional").

Lemma 1. For any expression e it follows that $(\sigma, v_1) \in \mathcal{E}(e) \wedge (\sigma, v_2) \in \mathcal{E}(e)$ $\Rightarrow$ $v_1 = v_2$.
Proof. This follows from the fact that there is exactly one rule for each type of expression construct. Even though the case for the disjunction operator is defined by the use of two set unions, the domains of the relations only overlap in the case of $mk\_Or(\mathbf{true}, \mathbf{true})$ where the result is the same regardless. In all of the other cases that are defined by the use of a set union, the domains of the relations never overlap. □

3.5 Relationships between the Semantic Definitions

This section will be used to record important results between the semantic definitions that have been presented in the previous sections.

A relationship between the big-step SOS definition and the small-step SOS definition is illustrated in the following lemma.

Lemma 2. If a result is returned from applying the big-step SOS definition, i.e. $(e, \sigma) \xrightarrow{e} v$, then the same result can be returned from applying the reflexive, transitive relation $\xrightarrow{E}$ from the small-step SOS definition, i.e. $(e, \sigma) \xrightarrow{E} v$, where $e \in Expr$, $\sigma \in \Sigma$, and v is a constant value ($Value$).
Proof. The proof follows by cases:

- $e \in Value$: The rule is the same;
- $e \in Prop$: The rule is the same;
- $e \in Var$: The rule is the same;

```
$\mathcal{E} : Expr^{\Delta} \to \mathcal{P}(\Sigma \times Value)$

$\mathcal{E}(e) \;\;\triangleq$
    cases $e$ of
                    $e \in Value \to \{(\sigma, e) \mid \sigma \in \Sigma\}$
                     $e \in Prop \to \{(\sigma, \sigma(e)) \mid \sigma \in \Sigma \wedge e \in \mathbf{dom}\,\sigma\}$
                      $e \in Var \to \{(\sigma, \sigma(e)) \mid \sigma \in \Sigma\}$
         $mk\_Arith(a, op, b) \to \{(\sigma, [\![op]\!](a', b')) \mid$
                                   $(\sigma, a') \in \mathcal{E}(a) \wedge (\sigma, b') \in \mathcal{E}(b) \wedge$
                                     $op \in \{+, -, \times\}\} \cup$
                               $\{(\sigma, [\![\div]\!](a', b')) \mid$
                                   $(\sigma, a') \in \mathcal{E}(a) \wedge (\sigma, b') \in \mathcal{E}(b) \wedge$
                                     $\wedge\; op = \div \wedge b' \neq 0\}$
         $mk\_Equality(a, b) \to \{(\sigma, [\![=]\!](a', b')) \mid$
                                   $(\sigma, a') \in \mathcal{E}(a) \wedge (\sigma, b') \in \mathcal{E}(b)\}$
           $mk\_Cond(p, a, b) \to \{(\sigma, a') \mid$
                                   $(\sigma, \mathbf{true}) \in \mathcal{E}(p) \wedge (\sigma, a') \in \mathcal{E}(a)\} \cup$
                               $\{(\sigma, b') \mid$
                                   $(\sigma, \mathbf{false}) \in \mathcal{E}(p) \wedge (\sigma, b') \in \mathcal{E}(b)\}$
                  $mk\_Not(p) \to \{(\sigma, \mathbf{true}) \mid (\sigma, \mathbf{false}) \in \mathcal{E}(p)\} \cup$
                               $\{(\sigma, \mathbf{false}) \mid (\sigma, \mathbf{true}) \in \mathcal{E}(p)\}$
                $mk\_delta(p) \to \{(\sigma, \mathbf{true}) \mid \sigma \in \mathbf{dom}\,\mathcal{E}(p)\} \cup$
                $mk\_Delta(p) \to \{(\sigma, \mathbf{true}) \mid \sigma \in \mathbf{dom}\,\mathcal{E}(p)\} \cup$
                               $\{(\sigma, \mathbf{false}) \mid \sigma \in (\Sigma \setminus \mathbf{dom}\,\mathcal{E}(p))\}$
                 $mk\_Or(p, q) \to \{(\sigma, \mathbf{true}) \mid (\sigma, \mathbf{true}) \in \mathcal{E}(p)\} \cup$
                               $\{(\sigma, \mathbf{true}) \mid (\sigma, \mathbf{true}) \in \mathcal{E}(q)\} \cup$
                               $\{(\sigma, \mathbf{false}) \mid (\sigma, \mathbf{false}) \in \mathcal{E}(p) \wedge$
                                   $(\sigma, \mathbf{false}) \in \mathcal{E}(q)\}$
            $mk\_Exists(x, p) \to \{(\sigma, \mathbf{true}) \mid$
                                   $\sigma \in \Sigma \wedge$
                                   $\mathbf{true} \in$
                                     $\mathbf{rng}\,(\{\sigma \dagger \{x \mapsto i\} \mid i{:}\,\mathbb{Z}\} \lhd \mathcal{E}(p))\} \cup$
                               $\{(\sigma, \mathbf{false}) \mid$
                                   $\sigma \in \Sigma \wedge$
                                   $\mathbf{rng}\,(\{\sigma \dagger \{x \mapsto i\} \mid i{:}\,\mathbb{Z}\} \lhd \mathcal{E}(p)) =$
                                     $\{\mathbf{false}\}\}$
    $mk\_FuncCall(f, al)$,
      $mk\_PredCall(f, al) \to \{(\sigma, r) \mid$
                                   $f \in (Fn \cup Pred) \wedge$
                                   $\sigma \in \Sigma \wedge$
                                   $\forall i{:}\, \mathbf{inds}\; al \cdot (\sigma, vl(i)) \in \mathcal{E}(al(i)) \wedge$
                                   $(vl, r) \in \sigma(f)\}$
    end
```

Figure 3.8: The $\mathcal{E}$ semantic function definition which defines the semantics of LPF.

- $mk\_Not(p)$: The big-step rules return a Boolean value if p can be evaluated to a Boolean value. The Not_A rule serves the purpose of performing one evaluation step on p at a time, where repeated application followed by applying a negation elimination rule is in effect performing the same function as the big-step $(p, \sigma) \xrightarrow{e}$ **true**/**false** transitions. The small-step elimination rules can return the same Boolean value if p can be evaluated to a Boolean value;

- $mk\_Cond(p, a, b)$: Follows in a similar way to the negation case;

- $mk\_Or(p, q)$: If true is returned by the big-step SOS rule then either p or q has been evaluated to true. The Or_L and the Or_R small-step SOS rules allow for repeated evaluation of p and q, and if p or q is evaluated to true then the Or_E1 and the Or_E2 small-step SOS rules if applied ensure that the same result is returned (the rules coincide with the semantics of the big-step SOS rules). The situation follows in a similar way when false is returned in the big-step SOS definition;

- $mk\_Equality(a, b)$: Follows in a similar way to the disjunction case;

- $mk\_Arith(a, op, b)$: Follows in a similar way to the disjunction case, since the guard in the division case of b' not being zero is present in both SOS definitions;

- $mk\_Exists(x, p)$: The big-step SOS rules are defined using quantifiers, which is a shorthand for disjunctions/conjunctions. If a result is returned by one of these big-step rules then either p is true for some $i \in \mathbb{Z}$, or p is false for every $i \in \mathbb{Z}$ (the quantifiers carry the problem with undefinedness as mentioned for the big-step disjunction logical operator). If a result is returned by the big-step SOS rules, then it is guaranteed by the use of the sets in the small-step SOS rules, that the same result can be returned, since in the small-step SOS rules no order of evaluation is specified, the choice that is made by the let expression is arbitrary;

- $mk\_FuncCall(id, args)$: If an argument in either definition (big-step SOS and small-step SOS) cannot be evaluated to an integer value then the functions definition expression is not evaluated, both the big-step SOS rule and the small-step SOS rule have a guard to ensure that each argument is evaluated to an integer value. If all arguments can be evaluated to an integer value then the result follows as discussed in the negation case; and

- $mk\_PredCall(id, args)$: Follows in a similar way to the function call case.

$\square$

Of course note that the small-step SOS definition can return results that the big-step SOS definition cannot, for instance, the big-step SOS definition could send *control* down an undefined operand to the disjunction logical operator. If this was to happen in the small-step SOS definition, then only the one single evaluation step would be made, then *control* could be passed to a rule corresponding to the other operand to the disjunction logical operator, which could be defined and true. No *strategy* has been defined to determine the selection of a small-step SOS rule in the small-step SOS definition presented, when multiple rules are available to be selected. This issue is considered in the mechanisation of the big-step SOS definition and the small-step SOS definition in Maude, in Section 5.1.

A relationship between the small-step SOS definition and the second $\mathcal{E}$ definition is illustrated in the following lemma. Functions and predicates are dealt with differently between the SOS definition and the $\mathcal{E}$ semantic definition. In the SOS definition: $Fn \xrightarrow{m} Func$, while in the $\mathcal{E}$ definition: $Fn \xrightarrow{m} Function$, where a *Func* is a record with an *Expr* field that needs evaluating (the *FuncCall* semantic/transition rules have to compute a value) while $Function = \mathcal{P}(\mathbb{Z}^* \times \mathbb{Z})$, so the functions result is not evaluated, it is checked whether for the arguments a result exists.

Certainly, any function mapped to by a *Fn* can be made to coincide. The set of *Fn* is the same, and for each $f \in Fn$, there is a *Func* and a *Function* (depending on the definition), taking the same number of arguments, and the corresponding *Func* and *Function* return the same result for the same arguments, that is, given $f \in Fn$, then if $mk\_FuncCall(f, [args]), \sigma) \xrightarrow{E} r$ then $(\sigma, r) \in \mathcal{E}(mk\_FuncCall(f, [args]))$, (so, $([args], r) \in \sigma(f)$) must hold. Additionally, if evaluating a $\sigma(f).result$ in the SOS rules and it is a "gap", that is, that no defined result can be computed, then for the same arguments that the function is being applied with, a result should not be defined in the corresponding *Function* definition, for instance, given the standard *zero* function, $([0], 0) \in \sigma(zero)$, but $([-1], r) \notin \sigma(zero)$, for any $r \in \mathbb{Z}$. The same can apply to predicates.

Lemma 3. If a result is returned from applying the small-step SOS rules, i.e. $(e, \sigma) \xrightarrow{E} v$, then $(\sigma, v) \in \mathcal{E}(e)$, where $e \in Expr$, $\sigma \in \Sigma$, and v is a constant value (*Value*). It is assumed that all function definitions and predicate definitions coincide.

Proof. The aim is to show that if an expression is evaluated to a *Value* through a transition relation $\xrightarrow{E}$ then the same *Value* will be returned in the DS definition. The proof follows by cases, for any $\sigma \in \Sigma$:

- $e \in \mathit{Value}$: In the SOS definition no further evaluation takes place since e is evaluated fully, and in the DS definition $(e, \sigma) \in \mathcal{E}(e)$ and thus no change is made to e.
- $e \in \mathit{Prop}$: If $(e, \sigma) \xrightarrow{E} v$, then $e \in \mathbf{dom}\,\sigma$ and $v = \sigma(e)$, and by the definition of $\mathcal{E}$ it follows that $(\sigma, v) \in \mathcal{E}(e)$.
- $\mathit{mk\_Or}(p, q)$: If $(\mathit{mk\_Or}(p, q), \sigma) \xrightarrow{E} \mathbf{true}$ then either p evaluated to true, or q evaluated to true. By the definition of $\mathcal{E}$, $(\sigma, \mathbf{true}) \in \mathcal{E}(\mathit{mk\_Or}(p, q))$ results from either case. The false case follows in a similar way.
- $\mathit{mk\_Exists}(x, p)$: If $(\mathit{mk\_Exists}(x, p), \sigma) \xrightarrow{E} \mathbf{true}$ then for some $i \in \mathbb{Z}$, $(p, \sigma \dagger \{x \mapsto i\}) \xrightarrow{E} \mathbf{true}$. The *let* expression makes an arbitrary selection for which i to evaluate p for. This coincides with the exists case of the $\mathcal{E}$ semantic function definition, where $\mathbf{true} \in (\{p, \sigma \dagger \{x \mapsto i\} \mid i \colon \mathbb{Z}\} \lhd \mathcal{E}(p))$. The false case in $\mathcal{E}$ needs false to be the value denoted for p for each $i \in \mathbb{Z}$. This coincides with the SOS definition for the false case of the existential quantifier.
- If $(\mathit{mk\_FuncCall}(id, args), \sigma) \xrightarrow{E} v$, then since *Func* and *Function* are assumed to coincide, and all argument(s) must denote otherwise a "gap" would have occurred (v would not have been output), then by the definition of $\mathcal{E}$ it follows that, $(\sigma, v) \in \mathcal{E}(\mathit{mk\_FuncCall}(id, args))$, since the functions coincide.

The rest of the cases follow in a similar way and are not outlined here. □

The two $\mathcal{E}$ definitions are shown to be equivalent in the following lemma.

Lemma 4. Let $\mathcal{E}_1$ be the first definition of $\mathcal{E}$ presented, and let $\mathcal{E}_2$ be the second definition of $\mathcal{E}$ presented, then $\mathcal{E}_1$ and $\mathcal{E}_2$ are equivalent, that is, given $e \in \mathit{Expr}$, $\sigma \in \Sigma$, and where v is a constant value (*Value*), then $((e, \sigma), v) \in \mathcal{E}_1$ iff $(\sigma, v) \in \mathcal{E}_2(e)$.

Proof. Note that Σ is defined the same in both $\mathcal{E}_1$ and $\mathcal{E}_2$. The proof follows by the following cases for any $\sigma \in \Sigma$:

- $e \in \mathit{Value}$: Since a constant value is defined within any σ it follows that $((e, \sigma), e) \in \mathcal{E}_1$ and $(\sigma, e) \in \mathcal{E}_2(e)$;

- $e \in Prop$: If $e \in \mathbf{dom}\,\sigma$ then $((e,\sigma),\sigma(e)) \in \mathcal{E}_1$ and $(\sigma,\sigma(e)) \in \mathcal{E}_2(e)$, otherwise both give rise to "gaps" and thus $(e,\sigma) \notin \mathbf{dom}\,\mathcal{E}_1$ and $\sigma \notin \mathbf{dom}\,\mathcal{E}_2(e)$;

- $mk\_Or(p,q)$, where p and q are both Boolean expressions (ensured by the context conditions): If it follows that $((p,\sigma),\mathbf{true}) \in \mathcal{E}_1$ and thus $(\sigma,\mathbf{true}) \in \mathcal{E}_2(p)$, then it follows by the definition of both $\mathcal{E}_1$ and $\mathcal{E}_2$ that $((mk\_Or(p,q),\sigma),\mathbf{true}) \in \mathcal{E}_1$ and $(\sigma,\mathbf{true}) \in \mathcal{E}_2(mk\_Or(p,q))$. The other cases follow in a similar way.

- $mk\_FuncCall(f, al)$:

  1. If one of the arguments $al(i)$ is a "gap" (i.e. it is not the case that $((al(i),\sigma),vl(i)) \in \mathcal{E}_1$ and thus $(\sigma,vl(i)) \in \mathcal{E}_2(al(i)))$ then in both $\mathcal{E}_1$ and in $\mathcal{E}_2$ a "gap" arises; and
  2. If for every argument $al(i)$ it follows that $((al(i),\sigma),vl(i)) \in \mathcal{E}_1$ and thus $(\sigma,vl(i)) \in \mathcal{E}_2(al(i))$, then either $(vl,r) \notin \sigma(f)$ and thus in both $\mathcal{E}_1$ and in $\mathcal{E}_2$ the function call expression gives rise to a "gap", or $(vl,r) \in \sigma(f)$, and thus by the definition of $\mathcal{E}_1$, $((mk\_FuncCall(f,al),\sigma),r) \in \mathcal{E}_1$, and by the definition of $\mathcal{E}_2$, $(\sigma,r) \in \mathcal{E}_2(mk\_FuncCall(f,al))$.

The rest of the cases follow in a similar way and are not outlined here. □

3.6 Conclusions

This chapter provided semantic definitions: SOS definitions (both big-step and small-step) and DS definitions, which formally capture the semantics of LPF. The SOS definitions illustrate the process of evaluating expressions according to the semantics of LPF. The DS definitions provide set theoretic definitions of the values that are denoted by expressions, again according to the semantics of LPF.

The benefits of providing such definitions are to have a precise semantic definition of LPF. This allows one to be clear about the semantics of LPF before attempting to provide a mechanisation of LPF. Additionally, the semantic definitions can form the basis of mechanisations. The SOS definitions form the basis of mechanisations in Chapter 5.

A DS definition of LPF is the key underlying basis used to facilitate some of the key work that is presented in the remainder of this thesis. The second DS definition that captures LPF is used in future chapters for two purposes. In Chapter 4 it is used to formally compare the semantics of different approaches to coping with partial terms, which arise for example from the application of partial functions. As part of this the second LPF DS definition is modified to formally define the semantics of the different approaches to coping with partial terms. It is interesting to note that only small changes need making to the LPF $\mathcal{E}$ DS definition and to the Σ variable, and function and predicate definition map, to be able to move between the different approaches considered to coping with partial terms. These DS definitions aid in making comparisons and identifying relationships between the different approaches to coping with partial terms, that is, to show how theorems can be moved between the different approaches. Additionally, in Chapter 6 the second LPF $\mathcal{E}$ DS definition is used to illustrate the issues in applying selected two-valued classical logic based proof techniques to LPF, to precisely define concepts, and to prove the changes to modifications made to carry the proof techniques over to LPF.

Notice that it is straightforward to remove the restrictions introduced during this chapter in allowing only Boolean values and integer values. Such a restriction was introduced to simplify the semantic definitions presented, but at the same time it was ensured that the issues that surround partial functions could still be adequately illustrated.

Chapter 4

A Formal Comparison of Approaches to Coping with Partial Terms

Contents

4.1 Alternative Semantic Definitions 93

4.1.1 Relations 94

4.1.2 Forcing all Terms to Denote 97

4.1.3 Semi-Classical Approaches 105

4.1.4 Weak Kleene Logic 112

4.1.5 McCarthy's Conditional Operators 113

4.1.6 Łukasiewicz's Logic 115

4.1.7 Bochvar's External Logic 115

4.1.8 Concluding Remarks 120

4.2 Comparing the Sequent Interpretations 123

4.2.1 SS 124

4.2.2 SW 124

4.2.3 WW 124

4.2.4 WS 125

4.3 Comparisons between the Different Approaches to Coping with Partial Terms 125

4.3.1 Comparison 1: The Values Denoted in the Different Approaches 125

4.3.2 Comparison 2: Non-classical Logic Comparisons . . 128

4.4 Relationships between the Different Approaches to Coping with Partial Terms and LPF 130

4.4.1 Relationship with Two-Valued Classical Logic using Overspecification 131

4.4.2 Relationship with the Well-Definedness Approach . . 132

4.4.3 Relationship with Two-Valued Classical Logic using Strong Equality 132

4.4.4 Relationship with the Weak Kleene Approach 134
4.4.5 Relationship with the McCarthy Conditional Operator Approach 134
4.4.6 Relationship with Łukasiewicz's Approach 136
4.4.7 Relationship with Bochvar's External Approach . . . 136
4.4.8 Concluding Remarks 137
4.5 Conclusions 139

It is useful to formally compare numerous approaches introduced in Chapter 2 to coping with logical formulae that can include references to partial terms. The $\mathcal{E}$ semantic function definition presented in Figure 3.8, which uses the variable and definition map Σ to capture the semantics of LPF, is used in this chapter as a semantic basis with which to conduct a formal comparison between different approaches to coping with partial terms.

The result of this work will first be an $\mathcal{E}$ semantic function definition for each approach considered in this chapter. Such $\mathcal{E}$ semantic function definitions formally illustrates the outputs from the different expressions that can be constructed, that is, that they describe the effect of executing each available expression construct, (expressions are mapped to relations over interpretations and results). The use of such $\mathcal{E}$ semantic function definitions is proposed as a way of formally comparing different approaches to coping with partial terms. Justifications for the choice of LPF for the mechanisation aim of this work are put forward from the comparison work in this chapter.

It is beneficial to provide such semantic function definitions, as precise definitions of the different approaches to coping with partial terms can be provided, which are often not made clear. The semantic function definitions are used to derive some formal comparisons between the different approaches. Comparisons are made on the meaning of expressions in the different approaches, and on properties that hold in the different non-classical logic approaches. They are also used to illustrate how to move theorems between different approaches to coping with partial terms.

Consideration is given to the changes that need making to the $\mathcal{E}$ semantic function definition and to the Σ map to capture different approaches to coping with partial terms. It is interesting to note that only rather small changes need making to $\mathcal{E}$ and to Σ to move between the different approaches, but from these subtle changes different advantages and drawbacks occur. The definitions provide a way to *quickly* and *easily* be able to compare the meaning of different expressions that are written within the different approaches.

Since only small changes need making to the LPF $\mathcal{E}$ semantic function definition to define the semantic function definitions of the other approaches to coping with partial terms, this allows for the differences between the approaches to be explained in terms of changes to the $\mathcal{E}$ semantic function definitions and to the Σ map definitions. In effect the semantic function definitions precisely and succinctly capture the crucial points and the differences between the different approaches.

The $\mathcal{E}$ semantic function definition was created as a more concise alternative to the SOS definitions to pinpoint precisely the semantics of LPF. Additionally, such an $\mathcal{E}$ semantic function definition could be used as a basis under which to conduct proof. This is done in Chapter 6. It turned out that this $\mathcal{E}$ semantic function definition (along with the Σ map) provided a good semantic basis for formally illustrating the semantics of the different approaches to coping with partial terms, and with which to draw comparisons between the different approaches.

(Some initial collaborative work on a very early preliminary draft of a subset of Section 4.1 was done with my supervisors and is published in [JLS12b]. Specifically, a subset of the $\mathcal{E}^C$, $\mathcal{E}^D$, and the $\mathcal{E}^\exists$ semantic function definitions. Such semantic function definitions have been overhauled with significant changes and significant extensions in this chapter, including the modification of cases, and the inclusion of further detail such as function call and predicate call expression cases. A small subset of the $\mathcal{E}$ semantic function definition named $\mathcal{E}^L$ also appeared in that paper. The work is supplemented in this chapter with further semantic function definitions for numerous other approaches to coping with partial terms, and with comparisons and the identification of ways to move theorems between the different approaches.)

4.1 Alternative Semantic Definitions

Numerous semantic function definitions in the style of the $\mathcal{E}$ semantic function definition for LPF are presented in this section. Each formally captures the semantics of an approach to coping with logical formulae that can contain references to partial terms, which were introduced in Chapter 2.

The earlier subsections of this chapter present the different $\mathcal{E}$ semantic function definitions in the order that was used in Chapter 2. Section 4.1.8 pinpoints the changes that are made for each of the new $\mathcal{E}$ semantic function definitions compared to the LPF $\mathcal{E}$ semantic function definition, and the Σ variable and definition map.

The four different sequent interpretations are defined formally in Section 4.2, using the LPF Σ variable and definition map.

From these formal definitions comparisons between the different approaches in terms of the meaning of expressions in the different approaches, and in properties that hold in the different non-classical logic approaches are made. The semantic definitions are then used to show how to move theorems between the different approaches.

4.1.1 Relations

In this approach function applications are to be written in terms of the membership of the graph of the function, for instance, $f(x) = y$ is to be written as $(x, y) \in f$. This forces the result of a function application to be a defined Boolean value (true or false).

First, in the $\mathcal{E}$ semantic function definition the definition of Σ allows for propositional variables *Prop* to be absent from the domain of a σ to allow for undefined propositional identifiers to occur. But, all *Var* identifiers are already assumed always to denote. To ensure that all propositional variables denote let Σ^R be the set of mappings that contains denotations for all used elements of *Id*:

$$\Sigma^R = \{\sigma \mid \sigma\colon \Sigma \wedge \mathbf{dom}\,\sigma = \mathit{Id}\}$$

so all *Prop* and *Var* identifiers used in each $\sigma \in \Sigma^R$ map to an appropriate value. Additionally, all *Fn* and *Pr* identifiers map to a *Function* or *Predicate* respectively.

Secondly, the set of expressions needs modifying. Taking Expr^Δ as the starting point of building up the $\mathcal{E}^R$ semantics, the first step is to remove the δ logical operator and the Δ logical operator from consideration. Additionally, the function call expression construct and the predicate call expression construct need removing. Thus Expr_1^R is defined as Expr^Δ with the aforementioned expression constructs removed. Furthermore, the conditional expression is no longer considered in this chapter, and is removed from Expr_1^R.

FuncCall and *PredCall* need to be constructed in a different way in this approach:

$$\begin{aligned} \mathit{FuncMem} :: \mathit{function} &: \mathit{Fn} \\ \mathit{args} &: \mathit{Expr}^* \\ \mathit{result} &: \mathit{Value} \end{aligned}$$

and:

$$\begin{aligned} \mathit{PredMem} :: \mathit{predicate} &: \mathit{Pr} \\ \mathit{args} &: \mathit{Expr}^* \\ \mathit{result} &: \mathit{Value} \end{aligned}$$

Any *FuncMem* expression has the name of the function, the arguments to be passed into the function, as well as the expected result of the function,

$(\mathit{args}, \mathit{result}) \in \mathit{function}$, but again abstract syntax has been used. The context conditions follow in a similar way as for *FuncCall* and for *PredCall*, with the *result* being of the type integer or Boolean respectively.

Then Expr^R is defined so that:

$$\mathit{Expr}^R = \mathit{Expr}_1^R \mid \mathit{FuncMem} \mid \mathit{PredMem}$$

Compared to the $\mathcal{E}$ semantic function definition the function call case needs changing in the $\mathcal{E}^R$ semantic function definition to prevent "gaps" from arising. The function call case of the $\mathcal{E}^R$ semantic function definition is defined as:

$$\begin{array}{l}
\mathit{mk\_FuncMem}(f, al, r) \rightarrow \\
\qquad\qquad \{(\sigma, \mathbf{true}) \mid \sigma \in \Sigma^R \wedge \\
\qquad\qquad\qquad \forall i \colon \mathbf{inds}\ al \cdot (\sigma, vl(i)) \in \mathcal{E}^R(al(i)) \wedge \\
\qquad\qquad\qquad (vl, r) \in \sigma(f)\} \cup \\
\qquad\qquad \{(\sigma, \mathbf{false}) \mid \sigma \in \Sigma^R \wedge \\
\qquad\qquad\qquad \forall i \colon \mathbf{inds}\ al \cdot (\sigma, vl(i)) \in \mathcal{E}^R(al(i)) \wedge \\
\qquad\qquad\qquad (vl, r) \notin \sigma(f)\} \cup \\
\qquad\qquad \{(\sigma, \mathbf{false}) \mid \sigma \in \Sigma^R \wedge \\
\qquad\qquad\qquad \exists i \colon \mathbf{inds}\ al \cdot \sigma \notin \mathbf{dom}\, \mathcal{E}^R(al(i))\}
\end{array}$$

where the first set returns true if the arguments/result pair is a member of the graph of the function. The second set returns false if the arguments/result pair is not a member of the graph of the function. The third set returns false if an argument is a "gap". So, $\mathit{mk\_FuncMem}(\mathit{zero}, [-1], 0)$, $(([-1], 0) \in \mathit{zero})$, is false given the usual definition of the *zero* function.

If one is to write $f(g(y))$ the definition is more verbose as $g(y)$ needs to be written in terms of membership of a function (a Boolean result), and functions are still restricted to integer arguments only by the context conditions. So, $([y], r_1) \in g$ and $([r_1], r_2) \in f$ must be written.

If the result value is not known, then existential quantifiers must be used:

$$\exists r \colon \mathbb{Z} \cdot ([0], r) \in \mathit{zero}$$

and so on.

The predicate and arithmetic cases need changing in a similar way to avoid "gaps" from arising. The equality case needs no changes making to it because, the four causes of "gaps" in the definitions (propositional variables, functions, predicates, and arithmetic expressions) are now all total.

The $\mathcal{E}^R$ semantic function is presented in Figure 4.1.

Note that in the $\mathcal{E}^R$ semantic function definition (and in the $\mathcal{E}^C$, $\mathcal{E}^D$, $\mathcal{E}^\exists$, and $\mathcal{E}^{==}$ semantic functions definitions that follow), no change has been made

```
$\mathcal{E}^R : Expr^R \to \mathcal{P}(\Sigma^R \times Value)$
$\mathcal{E}^R(e) \quad \triangleq$
    $\mathbf{cases}\ e\ \mathbf{of}$
                  $e \in Value \to \{(\sigma, e) \mid \sigma \in \Sigma^R\}$
                   $e \in Prop \to \{(\sigma, \sigma(e)) \mid \sigma \in \Sigma^R\}$
                    $e \in Var \to \{(\sigma, \sigma(e)) \mid \sigma \in \Sigma^R\}$
          $mk\_Arith(a, op, b) \to *$
          $mk\_Equality(a, b) \to \{(\sigma, [\![=]\!](a', b')) \mid$
                                      $(\sigma, a') \in \mathcal{E}^R(a) \wedge (\sigma, b') \in \mathcal{E}^R(b)\}$
                 $mk\_Not(p) \to \{(\sigma, \mathbf{true}) \mid (\sigma, \mathbf{false}) \in \mathcal{E}^R(p)\} \cup$
                                  $\{(\sigma, \mathbf{false}) \mid (\sigma, \mathbf{true}) \in \mathcal{E}^R(p)\}$
                $mk\_Or(p, q) \to \{(\sigma, \mathbf{true}) \mid (\sigma, \mathbf{true}) \in \mathcal{E}^R(p)\} \cup$
                                  $\{(\sigma, \mathbf{true}) \mid (\sigma, \mathbf{true}) \in \mathcal{E}^R(q)\} \cup$
                                  $\{(\sigma, \mathbf{false}) \mid (\sigma, \mathbf{false}) \in \mathcal{E}^R(p) \wedge$
                                      $(\sigma, \mathbf{false}) \in \mathcal{E}^R(q)\}$
            $mk\_Exists(x, p) \to \{(\sigma, \mathbf{true}) \mid$
                                      $\sigma \in \Sigma^R \wedge \mathbf{true} \in$
                                      $\mathbf{rng}\,(\{\sigma \dagger \{x \mapsto i\} \mid i\colon \mathbb{Z}\} \lhd \mathcal{E}^R(p))\} \cup$
                                  $\{(\sigma, \mathbf{false}) \mid$
                                      $\sigma \in \Sigma^R \wedge$
                                      $\mathbf{rng}\,(\{\sigma \dagger \{x \mapsto i\} \mid i\colon \mathbb{Z}\} \lhd \mathcal{E}^R(p)) =$
                                        $\{\mathbf{false}\}\}$
    $mk\_FuncMem(f, al, r),$
     $mk\_PredMem(f, al, r) \to \{(\sigma, \mathbf{true}) \mid f \in (Fn \cup Pred) \wedge$
                                      $\sigma \in \Sigma^R \wedge$
                                      $\forall i\colon \mathbf{inds}\ al \cdot (\sigma, vl(i)) \in \mathcal{E}^R(al(i)) \wedge$
                                      $(vl, r) \in \sigma(f)\} \cup$
                                  $\{(\sigma, \mathbf{false}) \mid f \in (Fn \cup Pred) \wedge$
                                      $\sigma \in \Sigma^R \wedge$
                                      $\forall i\colon \mathbf{inds}\ al \cdot (\sigma, vl(i)) \in \mathcal{E}^R(al(i)) \wedge$
                                      $(vl, r) \notin \sigma(f)\} \cup$
                                  $\{(\sigma, \mathbf{false}) \mid f \in (Fn \cup Pred) \wedge$
                                      $\sigma \in \Sigma^R \wedge$
                                      $\exists i\colon \mathbf{inds}\ al \cdot \sigma \notin \mathbf{dom}\,\mathcal{E}^R(al(i))\}$
    $\mathbf{end}$
```

* The arithmetic case needs changing in a similar way to the function case to avoid any "gaps" from arising.

Figure 4.1: The $\mathcal{E}^R$ function definition which defines the semantics for the viewing function application in terms of the membership of a graph approach

to the semantics of the disjunction case other than the renaming of $\mathcal{E}$ with $\mathcal{E}^i$, where necessary. Since these five approaches all use the two-valued classical logic logical operators no change is necessary. The logical operators are all guarded from undefined truth values in this semantic function definition. Thus such a definition is equivalent to the following definition:

$$
\begin{aligned}
&\mathit{mk\_Or}(p, q) \rightarrow \\
&\qquad\qquad \{(\sigma, \mathbf{true}) \mid (\sigma, \mathbf{true}) \in \mathcal{E}^R(p) \wedge (\sigma, \mathbf{true}) \in \mathcal{E}^R(q)\} \cup \\
&\qquad\qquad \{(\sigma, \mathbf{true}) \mid (\sigma, \mathbf{true}) \in \mathcal{E}^R(p) \wedge (\sigma, \mathbf{false}) \in \mathcal{E}^R(q)\} \cup \\
&\qquad\qquad \{(\sigma, \mathbf{true}) \mid (\sigma, \mathbf{false}) \in \mathcal{E}^R(p) \wedge (\sigma, \mathbf{true}) \in \mathcal{E}^R(q)\} \cup \\
&\qquad\qquad \{(\sigma, \mathbf{false}) \mid (\sigma, \mathbf{false}) \in \mathcal{E}^R(p) \wedge (\sigma, \mathbf{false}) \in \mathcal{E}^R(q)\}
\end{aligned}
$$

and therefore the shorter definition has been used.

The logical operators can be defined in $\mathcal{E}^R$ as in $\mathcal{E}$ as the changes discussed above ensure that the logical operators will not be presented with any undefined operands. Notice that when all operands to the logical operators are defined the logical operator cases of the LPF $\mathcal{E}$ semantic function definition will return the same results as two-valued classical logic. In other words when all operands to the logical operators are defined the result obtained in LPF coincides with the result that would be obtained in two-valued classical logic.[1]

It can be shown that $\mathcal{E}^R$ never yields a "gap".

Lemma 5. For any expression $e \in \mathit{Expr}^R$ it is the case that $\mathcal{E}^R(e)$ is total, i.e. for every expression e and each $\sigma \in \Sigma^R$ there exists a tuple $(\sigma, v) \in \mathcal{E}^R(e)$.
Proof. The proof is similar to that of Lemma 6. □

Additionally, $\mathcal{E}^R$ is deterministic, see Lemma 1.

4.1.2 Forcing all Terms to Denote

These approaches ensure that the two-valued classical logic logical operators and quantifiers can be used by ensuring that all functions and predicates denote, that is, they yield a result for every member of their domain. As discussed earlier there are two approaches to achieving this.

Underspecification

The underspecification approach ensures that each function yields an integer for every member of its domain. A term that applies a partial function with arguments from outside of its defined domain should denote an unspecified but definite integer value. For instance, $\mathit{zero}(-1)$ is to denote an unspecified but

[1]What differs between the three-valued logics considered later in this chapter is how the logical operators cope with undefined operands.

definite integer value; it should not be possible to know, or to be able to prove which integer value is yielded.

To ensure that all propositional variables do denote, let Σ^C be the set of mappings that contain denotations for all used elements of *Id*:

$$\begin{aligned}\Sigma^C = \{\sigma \mid \sigma\colon \Sigma \wedge \mathbf{dom}\,\sigma = \mathit{Id} \wedge \\ \mathit{fun\_constraint}^C(\sigma) \wedge \mathit{pred\_constraint}^C(\sigma)\}\end{aligned}$$

The first conjunct ensures that all *Prop* and *Var* identifiers used in each $\sigma \in \Sigma^C$ map to an appropriate value, e.g. no undefined propositional variables can occur. Additionally, all *Fn* and *Pr* identifiers map to a *Function* or *Predicate* respectively.

*Function*s and *Predicate*s can still be partial thus any *Function* and *Predicate* referenced by any $i \in \mathbf{dom}\,\sigma$ for all $\sigma \in \Sigma^C$ needs underspecifying, (the graphs of the partial functions need extending), so that for any $\sigma \in \Sigma^C$ the following function is satisfied:

$$\mathit{fun\_constraint}^C : \Sigma^C \to \mathbb{B}$$

$$\begin{aligned}&\mathit{fun\_constraint}^C(\sigma) \quad \triangleq \\ &\quad \forall \mathit{id}\colon \mathbf{dom}\,\sigma \cdot \mathit{id} \in \mathit{Fn} \;\Rightarrow \\ &\qquad ((\mathbf{len}\,\mathit{params}(\sigma(\mathit{id})) = 0 \wedge \exists r\colon \mathbb{Z} \cdot ([], r) \in \sigma(\mathit{id})) \vee \\ &\qquad\quad (\mathbf{let}\ n = \mathbf{len}\,\mathit{params}(\sigma(\mathit{id}))\ \mathbf{in} \\ &\qquad\qquad \forall d\colon \{[i_1, \ldots, i_n] \mid i_1\colon \mathbb{Z}, \ldots, i_n\colon \mathbb{Z}\} \cdot \exists r\colon \mathbb{Z} \cdot (d, r) \in \sigma(\mathit{id})))\end{aligned}$$

which ensures that for any argument passed into any function there will be a defined result yielded. The function *params* used in the definition of $\mathit{fun\_constraint}^C$ returns the number of parameters of the function.

Up until now partial functions have been underspecified (or overspecified which is an approach discussed in the following subsection) without stating how. It is left to an implementation to define how it is done. Ensuring that only total functions and under/overspecified functions (partial functions modelled as total functions) are present has been performed by ensuring that only functions that satisfy $\mathit{fun\_constraint}^C$ are considered. How to under/overspecify partial functions relies on knowing the defined domain of partial functions. The defined domain of partial functions is not always obvious. An alternative definition of the $\mathcal{E}^C$ semantic function definition (and the $\mathcal{E}^D$ overspecification semantic function definition presented later) would have been to avoid the use of $\mathit{fun\_constraint}^C$, and to have an extra case for the function call case of the semantic function definitions, which would return an unknown but definite value $r \in \mathbb{Z}$ if no result is defined, and so on.

The function $\mathit{pred\_constraint}^C$ also needs to be satisfied in the definition of Σ^C. The function $\mathit{pred\_constraint}^C$ would be defined in a similar way to the $\mathit{fun\_constraint}^C$ function.

Thus the only functions and predicates that can be referenced in any $\sigma \in \Sigma^C$ are total functions and predicates, and those functions and predicates that are underspecified (and are thus made total), by ensuring that they yield a defined, but unspecified and definite, value for all arguments from outside of their defined domain.

Next Expr^Δ is taken as the starting point of building up this semantic definition. The first step is to remove the δ logical operator and the Δ logical operator from consideration. Additionally, the conditional expression is removed as it is no longer being considered in this chapter. Thus Expr^C is defined as Expr^Δ with the aforementioned expressions removed.

The $\mathcal{E}^C$ semantic function is presented in Figure 4.2. The big change made to define the $\mathcal{E}^C$ semantic function from the $\mathcal{E}$ semantic function other than the introduction of the constraints on Σ^C which take care of ensuring that all used variables are defined, and that all present functions and predicates return a value for all members of their domain, is to ensure that division by zero returns an unspecified definite value $r \in \mathbb{Z}$ (such a value should remain unknown):

$$\begin{aligned}
&\mathit{mk\_Arith}(a, op, b) \to \\
&\qquad \{(\sigma, [\![op]\!](a', b')) \mid (\sigma, a') \in \mathcal{E}^C(a) \wedge (\sigma, b') \in \mathcal{E}^C(b) \wedge \\
&\qquad\qquad op \in \{+, -, \times\}\} \cup \\
&\qquad \{(\sigma, [\![\div]\!](a', b')) \mid (\sigma, a') \in \mathcal{E}^C(a) \wedge (\sigma, b') \in \mathcal{E}^C(b) \wedge \\
&\qquad\qquad op = \div \wedge b' \neq 0\} \cup \\
&\qquad \{(\sigma, r) \mid (\sigma, a') \in \mathcal{E}^C(a) \wedge (\sigma, b') \in \mathcal{E}^C(b) \wedge \\
&\qquad\qquad op = \div \wedge b' = 0 \wedge r \in \mathbb{Z}\}
\end{aligned}$$

where $r \in \mathbb{Z}$ is to return a definite (but unspecified) value from the set $\mathbb{Z}$.

The rest of the definition is relatively the same as in the $\mathcal{E}$ semantic function, since Σ^C takes the brunt of the changes when formally defining the semantics for the underspecification approach to coping with logical formulae that can contain references to partial terms.

Notice how the equality predicate is guarded from undefined operands due to the underspecification of all partial functions and predicates, and the underspecification of the division arithmetic case. It is known that equality will always yield a defined result whenever all of its operands denote a proper defined value. In the underspecification approach the ability to continue using two-valued classical logic is maintained. No change needs making in $\mathcal{E}^C$ to the logical operators from how they were defined in $\mathcal{E}$.

The definition of $\mathcal{E}^C$ avoids "gaps" from being introduced and thus it can

```
$\mathcal{E}^C : Expr^C \to \mathcal{P}(\Sigma^C \times Value)$
$\mathcal{E}^C(e) \;\; \triangleq$
    **cases** $e$ **of**
                 $e \in Value \to \{(\sigma, e) \mid \sigma \in \Sigma^C\}$
                  $e \in Prop \to \{(\sigma, \sigma(e)) \mid \sigma \in \Sigma^C\}$
                   $e \in Var \to \{(\sigma, \sigma(e)) \mid \sigma \in \Sigma^C\}$
        $mk\_Arith(a, op, b) \to \{(\sigma, [\![op]\!](a', b')) \mid$
                                   $(\sigma, a') \in \mathcal{E}^C(a) \wedge (\sigma, b') \in \mathcal{E}^C(b) \wedge$
                                     $op \in \{+, -, \times\}\} \cup$
                               $\{(\sigma, [\![\div]\!](a', b')) \mid$
                                   $(\sigma, a') \in \mathcal{E}^C(a) \wedge (\sigma, b') \in \mathcal{E}^C(b) \wedge$
                                     $op = \div \wedge b' \neq 0\} \cup$
                               $\{(\sigma, r) \mid$
                                   $(\sigma, a') \in \mathcal{E}^C(a) \wedge (\sigma, b') \in \mathcal{E}^C(b) \wedge$
                                     $op = \div \wedge b' = 0 \wedge r \in \mathbb{Z}\}$
         $mk\_Equality(a, b) \to \{(\sigma, [\![=]\!](a', b')) \mid$
                                   $(\sigma, a') \in \mathcal{E}^C(a) \wedge (\sigma, b') \in \mathcal{E}^C(b)\}$
                $mk\_Not(p) \to \{(\sigma, \mathbf{true}) \mid (\sigma, \mathbf{false}) \in \mathcal{E}^C(p)\} \cup$
                               $\{(\sigma, \mathbf{false}) \mid (\sigma, \mathbf{true}) \in \mathcal{E}^C(p)\}$
               $mk\_Or(p, q) \to \{(\sigma, \mathbf{true}) \mid (\sigma, \mathbf{true}) \in \mathcal{E}^C(p)\} \cup$
                               $\{(\sigma, \mathbf{true}) \mid (\sigma, \mathbf{true}) \in \mathcal{E}^C(q)\} \cup$
                               $\{(\sigma, \mathbf{false}) \mid (\sigma, \mathbf{false}) \in \mathcal{E}^C(p) \wedge$
                                   $(\sigma, \mathbf{false}) \in \mathcal{E}^C(q)\}$
           $mk\_Exists(x, p) \to \{(\sigma, \mathbf{true}) \mid$
                                   $\sigma \in \Sigma^C \wedge \mathbf{true} \in$
                                   $\mathbf{rng}\,(\{\sigma \dagger \{x \mapsto i\} \mid i{:}\,\mathbb{Z}\} \lhd \mathcal{E}^C(p))\} \cup$
                               $\{(\sigma, \mathbf{false}) \mid$
                                   $\sigma \in \Sigma^C \;\; \wedge$
                                   $\mathbf{rng}\,(\{\sigma \dagger \{x \mapsto i\} \mid i{:}\,\mathbb{Z}\} \lhd \mathcal{E}^C(p)) =$
                                     $\{\mathbf{false}\}\}$
    $mk\_FuncCall(f, al),$
     $mk\_PredCall(f, al) \to \{(\sigma, r) \mid$
                                   $f \in (Fn \cup Pred) \wedge$
                                   $\sigma \in \Sigma^C \wedge$
                                   $\forall i{:}\,\mathbf{inds}\; al \cdot (\sigma, vl(i)) \in \mathcal{E}^C(al(i)) \wedge$
                                   $(vl, r) \in \sigma(f)\}$
    **end**
```

Figure 4.2: The $\mathcal{E}^C$ function definition which defines the semantics for the underspecification approach

be shown that $\mathcal{E}^C$ can never yield "gaps".

Lemma 6. For any expression $e \in \mathit{Expr}^C$ it is the case that $\mathcal{E}^C(e)$ is total, i.e. for every expression e and each $\sigma \in \Sigma^C$ there exists a tuple $(\sigma, v) \in \mathcal{E}^C(e)$.
Proof. By structural induction over Expr^C.

Base cases: By the definition of Expr^C there are five base cases to consider, $e \in \mathit{Value}$, $e \in \mathit{Prop}$, $e \in \mathit{Var}$, $\mathit{mk\_FuncCall}(f, al)$ where $f \in \mathit{Fn}$ and $\mathit{mk\_PredCall}(P, al)$ where $P \in \mathit{Pr}$:

1. $e \in \mathit{Value}$: the only constants defined in the language are the set of integer values $\{\ldots, -1, 0, 1, \ldots\}$ and the set of Boolean values $\{\mathbf{true}, \mathbf{false}\}$. These values are always defined and thus they are always defined in any given σ.
2. $e \in \mathit{Prop}$: Σ^C is restricted so that every used element of Id ($\mathit{Prop} \in \mathit{Id}$) is in the domain of each $\sigma \in \Sigma^C$. Thus it follows that every used Prop maps to a proper Boolean value.
3. $e \in \mathit{Var}$: follows in a similar way to case 2.
4. $\mathit{mk\_FuncCall}(f, al)$, where $f \in \mathit{Fn}$: it is known that $\mathit{Fn} \in \mathit{Id}$ and due to the restriction placed on Σ^C it is also known that each used Fn in the domain of each $\sigma \in \Sigma^C$ maps to a *Function* object. It is then the case that each *Function* in each $\sigma \in \Sigma^C$ yields an integer as a result for every argument in its domain (due to $\mathit{fun\_constraint}^C$), and thus every *Function* denotes irrespective of the argument(s) passed into the function.
5. $\mathit{mk\_PredCall}(P, al)$, where $P \in \mathit{Pr}$: follows in a similar way to case 4.

Inductive cases: By the definition of Expr^C, it is known that the inductive cases that need considering are arithmetic, equality, negation, disjunction and existential quantification. The proofs of the arithmetic cases, disjunction and existential quantification are presented below and similar reasoning can be applied to the equality and negation cases.

1. Consider the arithmetic cases. Clearly addition, subtraction and multiplication are defined for any two integers, and thus in these cases the arithmetic expression will be defined as by the induction hypothesis a and b are both defined. For the division case by the induction hypothesis a and b are both defined, and the two set unions for division cover

all cases, in the first case a defined result is returned by the standard division operators, and in the second case a defined result is returned through the arbitrary choice of an integer value.

2. Consider the $mk\_Or(p, q)$ case. By the induction hypothesis it is the case that p and q are both defined. By the context conditions it is known that both p and q are Boolean-valued expressions. Thus by the induction hypothesis p and q must denote one of two values, **true** and **false**. It is known that the definition of $\mathcal{E}^C$ defines a result for the $\vee$ logical operator whenever the operands denote one of the two values **true** or **false**. It therefore follows by the definition of $\mathcal{E}^C$ that the expression $mk\_Or(p, q)$ is defined as required.

3. Consider the $mk\_Exists(x, p)$ case. By the induction hypothesis it is the case that p is defined. If x is not free in p, since by the induction hypothesis p is defined and by the context conditions that p is a Boolean-valued expression, p must therefore always denote either **true** or **false** in any $\sigma \in \Sigma^C$. It therefore follows that for every σ, **true** or **false** will always be a member of $\mathbf{rng}\, \mathcal{E}^C(p)$. If x is not free in p by the definition of $\mathcal{E}^C$ it follows that $mk\_Exists(x, p)$ will always be defined as required, as the extra mapping for x in σ is irrelevant to the result.

   If x occurs free in p, then it must be the case that $x \in Var$, since the type Id is disjoint. It follows that the free variable x will subsequently be bound and it is known that quantification is only performed over the set of proper (i.e. defined) integer values. Thus by the definition of $\mathcal{E}^C$, when σ is updated to override the mapping for x, x will only ever be mapped to an integer value. Since by the induction hypothesis p is defined and since x is a defined integer variable (all Var in Σ^C denote), it must follow by the definition of $\mathcal{E}^C$ that $mk\_Exists(x, p)$ is defined as required.

□

The definition $\mathcal{E}^C$ is deterministic, since definite but unspecified defined values are assumed to be returned, see Lemma 1.

In Chapter 2, two underspecification approaches are discussed, returning a definite value (used to guide the discussion in this section so far), and returning an arbitrary value. To define this arbitrary value approach in the arithmetic case of $\mathcal{E}^C$, re-interpret $r \in \mathbb{Z}$ from returning a definite value to returning an arbitrary value, but such a value is still unknown/unspecified. Any partial functions and partial predicates need extending so that they still satisfy *fun_constraint*^{C} and *pred_constraint*^{C} respectively, but where it is to be that an arbitrary defined result will be yielded. With such an extension the $\mathcal{E}^C$ semantic function can still never yield any "gaps" (see Lemma 6), but the $\mathcal{E}^C$ semantic function will not be deterministic, since:

$$mk\_Equality(mk\_Arith(5, \div, 0), mk\_Arith(5, \div, 0))$$

can yield both true and false if an arbitrary but still unspecified value is returned. It is also the case that the law of the excluded middle does not hold, because:

$$\begin{array}{l} mk\_Or(\\ \qquad mk\_Equality(mk\_Arith(5, \div, 0), 0), \\ \qquad\quad mk\_Equality(mk\_Arith(5, \div, 0), 0)) \end{array}$$

can yield both true and false, if an arbitrary underspecified value is returned.

Overspecification

The alternative approach to forcing all terms to denote is to ensure that each partial function is overspecified, so that a default (known) value is returned whenever a function is applied with arguments from outside of its actual defined domain. For instance, $5/0 = 0$.

The Σ^C definition can be used in the definition of $\mathcal{E}^D$ but the way that the graph of the functions is extended will change.

The $\mathcal{E}^D$ semantic function is defined in Figure 4.3. As in the case of defining the semantics for the underspecification approach the brunt of the changes when defining the overspecification approach are made to Σ^C. In fact the $\mathcal{E}^D$ semantic function follows in basically the same way as the $\mathcal{E}^C$ semantic function except for the arithmetic cases, where for division by zero a definite/known integer value 0 is returned:

$$\begin{array}{l} mk\_Arith(a, op, b) \to \\ \qquad \{(\sigma, [\![op]\!](a', b')) \mid (\sigma, a') \in \mathcal{E}^D(a) \wedge (\sigma, b') \in \mathcal{E}^D(b) \wedge \\ \qquad\quad op \in \{+, -, \times\}\} \cup \\ \qquad \{(\sigma, [\![\div]\!](a', b')) \mid (\sigma, a') \in \mathcal{E}^D(a) \wedge (\sigma, b') \in \mathcal{E}^D(b) \wedge \\ \qquad\quad op = \div \wedge b' \neq 0\} \cup \\ \qquad \{(\sigma, 0) \mid (\sigma, a') \in \mathcal{E}^D(a) \wedge (\sigma, b') \in \mathcal{E}^D(b) \wedge \\ \qquad\quad op = \div \wedge b' = 0\} \end{array}$$

The choice to return 0 is just an arbitrary choice. Any defined value could have been chosen here to be returned for the division by zero case.

It can be shown that $\mathcal{E}^D$ never yields a "gap".

```
$\mathcal{E}^D : \mathit{Expr}^C \to \mathcal{P}(\Sigma^C \times \mathit{Value})$

$\mathcal{E}^D(e) \;\; \triangleq$
    $\textbf{cases } e \textbf{ of}$
                  $e \in \mathit{Value} \to \{(\sigma, e) \mid \sigma \in \Sigma^C\}$
                   $e \in \mathit{Prop} \to \{(\sigma, \sigma(e)) \mid \sigma \in \Sigma^C\}$
                    $e \in \mathit{Var} \to \{(\sigma, \sigma(e)) \mid \sigma \in \Sigma^C\}$
         $\mathit{mk\_Arith}(a, op, b) \to \{(\sigma, [\![op]\!](a', b')) \mid$
                                      $(\sigma, a') \in \mathcal{E}^D(a) \wedge (\sigma, b') \in \mathcal{E}^D(b) \wedge$
                                        $op \in \{+, -, \times\}\} \cup$
                                 $\{(\sigma, [\![\div]\!](a', b')) \mid$
                                      $(\sigma, a') \in \mathcal{E}^D(a) \wedge (\sigma, b') \in \mathcal{E}^D(b) \wedge$
                                        $op = \div \wedge b' \neq 0\} \cup$
                                 $\{(\sigma, 0) \mid$
                                      $(\sigma, a') \in \mathcal{E}^D(a) \wedge (\sigma, b') \in \mathcal{E}^D(b) \wedge$
                                        $op = \div \wedge b' = 0\}$
          $\mathit{mk\_Equality}(a, b) \to \{(\sigma, [\![=]\!](a', b')) \mid$
                                      $(\sigma, a') \in \mathcal{E}^D(a) \wedge (\sigma, b') \in \mathcal{E}^D(b)\}$
                  $\mathit{mk\_Not}(p) \to \{(\sigma, \textbf{true}) \mid (\sigma, \textbf{false}) \in \mathcal{E}^D(p)\} \cup$
                                 $\{(\sigma, \textbf{false}) \mid (\sigma, \textbf{true}) \in \mathcal{E}^D(p)\}$
               $\mathit{mk\_Or}(p, q) \to \{(\sigma, \textbf{true}) \mid (\sigma, \textbf{true}) \in \mathcal{E}^D(p)\} \cup$
                                 $\{(\sigma, \textbf{true}) \mid (\sigma, \textbf{true}) \in \mathcal{E}^D(q)\} \cup$
                                 $\{(\sigma, \textbf{false}) \mid (\sigma, \textbf{false}) \in \mathcal{E}^D(p) \wedge$
                                      $(\sigma, \textbf{false}) \in \mathcal{E}^D(q)\}$
           $\mathit{mk\_Exists}(x, p) \to \{(\sigma, \textbf{true}) \mid$
                                      $\sigma \in \Sigma^C \wedge \textbf{true} \in$
                                      $\textbf{rng}\,(\{\sigma \dagger \{x \mapsto i\} \mid i\colon \mathbb{Z}\} \lhd \mathcal{E}^D(p))\} \cup$
                                 $\{(\sigma, \textbf{false}) \mid$
                                      $\sigma \in \Sigma^C \;\; \wedge$
                                      $\textbf{rng}\,(\{\sigma \dagger \{x \mapsto i\} \mid i\colon \mathbb{Z}\} \lhd \mathcal{E}^D(p)) =$
                                        $\{\textbf{false}\}\}$
    $\mathit{mk\_FuncCall}(f, al),$
      $\mathit{mk\_PredCall}(f, al) \to \{(\sigma, r) \mid$
                                      $f \in (\mathit{Fn} \cup \mathit{Pred}) \wedge$
                                      $\sigma \in \Sigma^C \wedge$
                                      $\forall i\colon \textbf{inds}\; al \cdot (\sigma, vl(i)) \in \mathcal{E}^D(al(i)) \wedge$
                                      $(vl, r) \in \sigma(f)\}$
    $\textbf{end}$
```

Figure 4.3: The $\mathcal{E}^D$ semantic function definition which defines the semantics for the overspecification approach

Lemma 7. For any expression $e \in Expr^C$ it is the case that $\mathcal{E}^D(e)$ is total, i.e. for every expression e and each $\sigma \in \Sigma^C$ there exists a tuple $(\sigma, v) \in \mathcal{E}^D(e)$.
Proof. The proof is similar to that of Lemma 6. It is the case that Σ^C only has reference to functions and predicates that yield a result for every argument in their domain, (either a total function or a partial function overspecified). Additionally, the division by zero case for an arithmetic expression returns a defined value, namely 0. □

Additionally, $\mathcal{E}^D$ is deterministic, see Lemma 1.

In the $\mathcal{E}$ semantic function definition, functions are defined to return the same results whenever they are applied with the same arguments in a given $\sigma \in \Sigma$; $\mathcal{E}$ is deterministic. The definite value underspecification approach and the overspecification approach are both deterministic. However, the arbitrary value underspecification approach is not deterministic.

The Well-Definedness Approach

In this approach validity and well-definedness are proven separately. The LPF $\mathcal{E}$ semantic function definition is used as a basis to define the WD approach. It is ensured that $\mathcal{E}$ (only considering $Expr^C$ though) is total by only considering those expressions $e \in Expr^C$ such that:

$$\forall \sigma \colon \Sigma \cdot \exists v \colon Value \cdot (\sigma, v) \in \mathcal{E}(e)$$

4.1.3 Semi-Classical Approaches

Semantic function definitions for two related approaches, the existential equality approach, and the strong equality approach are both presented in this section.

Existential Equality

In this approach terms can be undefined. The aim is to catch such undefinedness at the predicate level, and in doing so guarding the logical operators from any undefinedness that may arise. Recall that the notion of equality used in the definition of $\mathcal{E}$ is weak/strict, that is, if either (or both) operands are undefined then the result of the equality is undefined. Existential equality is defined to return false if either operand is undefined, as discussed Chapter 2. Other predicates need treating in a similar way to that discussed for existential equality.

The set of expressions first needs extending, so $Expr^{\exists}$ is defined as:

$$\mathit{Expr}^{\exists} = \mathit{Expr}^{C} \mid \mathit{ExEquality}$$

where the abstract syntax and the context condition for *ExEquality* are defined in the same way as for *Equality*.

The Σ^R variable and definition map should be used in the definition of the $\mathcal{E}^{\exists}$ semantic function definition, since propositional variables should denote, to ensure that no "gaps" can arise in such Boolean expressions.

The $\mathcal{E}^{\exists}$ semantic function is presented in Figure 4.4 and in Figure 4.5.

In Figure 4.4, the notion of equality used is existential equality which is defined in $\mathcal{E}^{\exists}$ as:

$$\begin{array}{l}
\mathit{mk\_ExEquality}(a, b) \to \\
\qquad\qquad \{(\sigma, [\![=]\!](a', b')) \mid (\sigma, a') \in \mathcal{E}^{\exists}(a) \wedge (\sigma, b') \in \mathcal{E}^{\exists}(b)\} \cup \\
\qquad\qquad \{(\sigma, \mathbf{false}) \mid \sigma \in (\Sigma^R \setminus \mathbf{dom}\,\mathcal{E}^{\exists}(a))\} \cup \\
\qquad\qquad \{(\sigma, \mathbf{false}) \mid \sigma \in (\Sigma^R \setminus \mathbf{dom}\,\mathcal{E}^{\exists}(b))\}
\end{array}$$

where the first case is the usual equality case already presented. The second and the third case ensure that whenever one of the operands to *ExEquality* is undefined that a defined value false is returned.

The function case of $\mathcal{E}^{\exists}$ is the same as in $\mathcal{E}$. Additionally, the arithmetic case of $\mathcal{E}^{\exists}$ is the same as in $\mathcal{E}$. This is because "gaps" in this approach can still arise in terms, just not in predicates.

The predicate case of $\mathcal{E}^{\exists}$ though needs changing from what was presented in $\mathcal{E}$, to ensure that any undefinedness that arises does not propagate upwards to the logical operators:

$$\begin{array}{l}
\mathit{mk\_PredCall}(P, \mathit{al}) \to \\
\qquad\qquad \{(\sigma, r) \mid \sigma \in \Sigma^R \wedge \\
\qquad\qquad\qquad \forall i \colon \mathbf{inds}\ \mathit{al} \cdot (\sigma, \mathit{vl}(i)) \in \mathcal{E}^{\exists}(\mathit{al}(i)) \wedge \\
\qquad\qquad\qquad (\mathit{vl}, r) \in \sigma(P)\} \cup \\
\qquad\qquad \{(\sigma, \mathbf{false}) \mid \sigma \in \Sigma^R \wedge \\
\qquad\qquad\qquad \forall i \colon \mathbf{inds}\ \mathit{al} \cdot (\sigma, \mathit{vl}(i)) \in \mathcal{E}^{\exists}(\mathit{al}(i)) \wedge \\
\qquad\qquad\qquad \mathit{vl} \notin \mathbf{dom}\,\sigma(P)\} \cup \\
\qquad\qquad \{(\sigma, \mathbf{false}) \mid \sigma \in \Sigma^R \wedge \\
\qquad\qquad\qquad \exists i \colon \mathbf{inds}\ \mathit{al} \cdot \sigma \in (\Sigma^R \setminus \mathbf{dom}\,\mathcal{E}^{\exists}(\mathit{al}(i)))\}
\end{array}$$

The first set of this predicate call case of $\mathcal{E}^{\exists}$ returns the value r yielded by the predicate if all arguments are defined and the predicate yields a value when applied with those arguments. The second set ensures that the truth value false is returned if the predicate does not yield a result despite all arguments being applied to the predicate denoting, with the third set returning false if an argument applied to the predicate is undefined (does not denote a proper value).

```
$\mathcal{E}^{\exists} : \mathit{Expr}^{\exists} \to \mathcal{P}(\Sigma^R \times \mathit{Value})$
$\mathcal{E}^{\exists}(e) \quad \triangleq$
    **cases** $e$ **of**
                  $e \in \mathit{Value} \to \{(\sigma, e) \mid \sigma \in \Sigma^R\}$
                   $e \in \mathit{Prop} \to \{(\sigma, \sigma(e)) \mid \sigma \in \Sigma^R\}$
                    $e \in \mathit{Var} \to \{(\sigma, \sigma(e)) \mid \sigma \in \Sigma^R\}$
        $\mathit{mk\_Arith}(a, op, b) \to \{(\sigma, [\![op]\!](a', b')) \mid$
                                    $(\sigma, a') \in \mathcal{E}^{\exists}(a) \wedge (\sigma, b') \in \mathcal{E}^{\exists}(b) \wedge$
                                      $op \in \{+, -, \times\}\} \cup$
                                 $\{(\sigma, [\![\div]\!](a', b')) \mid$
                                    $(\sigma, a') \in \mathcal{E}^{\exists}(a) \wedge (\sigma, b') \in \mathcal{E}^{\exists}(b) \wedge$
                                      $op = \div \wedge b' \neq 0\}$
        $\mathit{mk\_Equality}(a, b) \to \{(\sigma, [\![=]\!](a', b')) \mid$
                                    $(\sigma, a') \in \mathcal{E}^{\exists}(a) \wedge (\sigma, b') \in \mathcal{E}^{\exists}(b)\}$
      $\mathit{mk\_ExEquality}(a, b) \to \{(\sigma, [\![=]\!](a', b')) \mid$
                                    $(\sigma, a') \in \mathcal{E}^{\exists}(a) \wedge (\sigma, b') \in \mathcal{E}^{\exists}(b)\} \cup$
                                 $\{(\sigma, \mathbf{false}) \mid \sigma \in (\Sigma^R \setminus \mathbf{dom}\, \mathcal{E}^{\exists}(a))\} \cup$
                                 $\{(\sigma, \mathbf{false}) \mid \sigma \in (\Sigma^R \setminus \mathbf{dom}\, \mathcal{E}^{\exists}(b))\}$
                 $\mathit{mk\_Not}(p) \to \{(\sigma, \mathbf{true}) \mid (\sigma, \mathbf{false}) \in \mathcal{E}^{\exists}(p)\} \cup$
                                 $\{(\sigma, \mathbf{false}) \mid (\sigma, \mathbf{true}) \in \mathcal{E}^{\exists}(p)\}$
               $\mathit{mk\_Or}(p, q) \to \{(\sigma, \mathbf{true}) \mid (\sigma, \mathbf{true}) \in \mathcal{E}^{\exists}(p)\} \cup$
                                 $\{(\sigma, \mathbf{true}) \mid (\sigma, \mathbf{true}) \in \mathcal{E}^{\exists}(q)\} \cup$
                                 $\{(\sigma, \mathbf{false}) \mid (\sigma, \mathbf{false}) \in \mathcal{E}^{\exists}(p) \wedge$
                                    $(\sigma, \mathbf{false}) \in \mathcal{E}^{\exists}(q)\}$
                          $\ldots$
    **end**
```

Figure 4.4: The $\mathcal{E}^{\exists}$ semantic function definition which defines the semantics for the approach of forcing all predicates to denote using existential equality (part 1)

```
$\mathcal{E}^{\exists} : \mathit{Expr}^{\exists} \to \mathcal{P}(\Sigma^R \times \mathit{Value})$
$\mathcal{E}^{\exists}(e) \quad \triangleq$
    $\mathbf{cases}\ e\ \mathbf{of}$
                    $\ldots$
        $\mathit{mk\_Exists}(x, p) \to \{(\sigma, \mathbf{true}) \mid$
                        $\sigma \in \Sigma^R \wedge$
                        $\mathbf{true} \in$
                          $\mathbf{rng}\,(\{\sigma \dagger \{x \mapsto i\} \mid i{:}\,\mathbb{Z}\} \lhd \mathcal{E}^{\exists}(p))\} \cup$
                    $\{(\sigma, \mathbf{false}) \mid$
                        $\sigma \in \Sigma^R \wedge$
                        $\mathbf{rng}\,(\{\sigma \dagger \{x \mapsto i\} \mid i{:}\,\mathbb{Z}\} \lhd \mathcal{E}^{\exists}(p)) =$
                          $\{\mathbf{false}\}\}$
    $\mathit{mk\_FuncCall}(f, al) \to \{(\sigma, r) \mid \sigma \in \Sigma^R \wedge$
                        $\forall i{:}\,\mathbf{inds}\,al \cdot (\sigma, vl(i)) \in \mathcal{E}^{\exists}(al(i)) \wedge$
                        $(vl, r) \in \sigma(f)\}$
    $\mathit{mk\_PredCall}(P, al) \to \{(\sigma, r) \mid \sigma \in \Sigma^R \wedge$
                        $\forall i{:}\,\mathbf{inds}\,al \cdot (\sigma, vl(i)) \in \mathcal{E}^{\exists}(al(i)) \wedge$
                        $(vl, r) \in \sigma(P)\} \cup$
                    $\{(\sigma, \mathbf{false}) \mid \sigma \in \Sigma^R \wedge$
                        $\forall i{:}\,\mathbf{inds}\,al \cdot (\sigma, vl(i)) \in \mathcal{E}^{\exists}(al(i)) \wedge$
                        $vl \notin \mathbf{dom}\,\sigma(P)\} \cup$
                    $\{(\sigma, \mathbf{false}) \mid \sigma \in \Sigma^R \wedge$
                        $\exists i{:}\,\mathbf{inds}\,al \cdot \sigma \in$
                          $(\Sigma^R \setminus \mathbf{dom}\,\mathcal{E}^{\exists}(al(i)))\}$
    $\mathbf{end}$
```

Figure 4.5: The $\mathcal{E}^{\exists}$ semantic function definition which defines the semantics for the approach of forcing all predicates to denote using existential equality (part 2)

Weak equality has been kept in the $\mathcal{E}^{\exists}$ semantic function definition because it still needs to be written in function definitions. The non-strict existential equality notion is defined in $\mathcal{E}^{\exists}$ to cope with any partial terms that can arise in logical formulae. Thus a user has to be aware of multiple notions of equality when reasoning about logical formulae that can contain reference to partial terms in such an approach. If the weak notion of equality (*Equality*) is removed from $Expr^{\exists}$ and thus as a case from $\mathcal{E}^{\exists}$, then it can be shown that $\mathcal{E}^{\exists}$ is total over Boolean expressions.

Unlike $\mathcal{E}^{C}$ and $\mathcal{E}^{D}$, $\mathcal{E}^{\exists}$ is not always total over integer expressions, since functions and the division arithmetic case can still cause "gaps", that is, integer terms can still be undefined in the $\mathcal{E}^{\exists}$ semantic function. However, all predicates are forced to denote other than the weak equality construct where a "gap" can still result, but the existential notion of equality is to be used for reasoning about logical formulae that can contain references to partial terms.

Lemma 8. For any Boolean expression e, where e excludes any reference to *Equality*, then $\mathcal{E}^{\exists}(e)$ is total, i.e. for every Boolean expression e and each $\sigma \in \Sigma^{R}$, there must exist a tuple $(\sigma, v) \in \mathcal{E}^{\exists}(e)$.
Proof. The proof is similar to that of Lemma 6. $\square$

$\mathcal{E}^{\exists}$ is also deterministic, see Lemma 1.

Strong Equality

Strong equality can be defined in a similar way to existential equality. Recall that the difference between strong equality and existential equality is that strong equality yields the truth value true when both of its operands do not denote, but existential equality in such a case would yield the truth value false when both of its operands do not denote.

As for the $\mathcal{E}^{\exists}$ semantic function, the $\mathcal{E}^{==}$ semantic function can be defined using Σ^{R}.

As for existential equality the set of expressions must be extended, so $Expr^{==}$ is defined as:

$Expr^{==} = Expr^{C} \mid StEquality$

where the abstract syntax and the context conditions for *StEquality* are the same as those for *Equality*.

In Figure 4.6 and in Figure 4.7 the full $\mathcal{E}^{==}$ semantic function is presented where the main change is the following case:

$$
\begin{aligned}
&\mathit{mk\_StEquality}(a, b) \rightarrow \\
&\qquad\qquad \{(\sigma, [\![=]\!](a', b')) \mid (\sigma, a') \in \mathcal{E}^{==}(a) \wedge (\sigma, b') \in \mathcal{E}^{==}(b)\} \cup \\
&\qquad\qquad \{(\sigma, \mathbf{true}) \mid \sigma \in (\Sigma^R \setminus \mathbf{dom}\, \mathcal{E}^{==}(a)) \wedge \\
&\qquad\qquad\qquad \sigma \in (\Sigma^R \setminus \mathbf{dom}\, \mathcal{E}^{==}(b))\} \cup \\
&\qquad\qquad \{(\sigma, \mathbf{false}) \mid \sigma \in (\Sigma^R \setminus \mathbf{dom}\, \mathcal{E}^{==}(a)) \wedge \sigma \in \mathbf{dom}\, \mathcal{E}^{==}(b)\} \cup \\
&\qquad\qquad \{(\sigma, \mathbf{false}) \mid \sigma \in \mathbf{dom}\, \mathcal{E}^{==}(a) \wedge \sigma \in (\Sigma^R \setminus \mathbf{dom}\, \mathcal{E}^{==}(b))\}
\end{aligned}
$$

The nature of the interpretation that is given to strong equality leads to a more complicated definition than that which was required for existential equality. For existential equality if any operand is undefined then false is returned, while for strong equality false is only to be returned if the one operand is not defined, if both operands are undefined then true is to be returned.

The predicate call case of the $\mathcal{E}^{==}$ semantic function also needs to ensure that no "gap" can arise, since a defined result must always be returned. The predicate call case of the $\mathcal{E}^{\exists}$ semantic function definition could be used as when a predicate does not denote then the value false is returned. However, consider a predicate $\neq\neq$ (strong inequality) being defined then when both arguments do not denote the value true would be expected to be returned. This leads to a more complicated predicate call case than that needed for the predicate call case of the $\mathcal{E}^{\exists}$ semantic function, since an extra set is needed to ensure that if all operands do not denote, then the truth value true should be returned, and additionally, the set that returned false needs extending, (the set that returned false when at least one operand did not denote a proper value). This set needs extending to ensure that not only does at least the one operand not denote a proper value, but that at least one operand does denotes a proper value, to ensure that no two sets that define the semantics for the $\mathcal{E}^{==}$ semantic function overlap. This is similar to the reason for the more complicated case definition that is needed for strong equality, compared to the case definition that was needed for existential equality.

The same notes about the weak equality case still being present that were discussed for the $\mathcal{E}^{\exists}$ semantic function definition also apply to the $\mathcal{E}^{==}$ semantic function definition. Unlike $\mathcal{E}^{C}$ and $\mathcal{E}^{D}$, $\mathcal{E}^{==}$ is not always total over integer expressions (like $\mathcal{E}^{\exists}$), since functions and the arithmetic division case can still cause "gaps" to arise.

Lemma 9. For any Boolean expression e, where e excludes any reference to *Equality* then, $\mathcal{E}^{==}(e)$ is total, i.e. for every Boolean expression e and each $\sigma \in \Sigma$, there must exist a tuple $(\sigma, v) \in \mathcal{E}^{==}(e)$.

Proof. The proof is similar to that of Lemma 6. □

```
$\mathcal{E}^{==} : \mathit{Expr}^{==} \to \mathcal{P}(\Sigma^R \times \mathit{Value})$

$\mathcal{E}^{==}(e) \quad \triangleq$
    **cases** $e$ **of**
                    $e \in \mathit{Prop} \to \{(\sigma, \sigma(e)) \mid \sigma \in \Sigma^R\}$
                     $e \in \mathit{Var} \to \{(\sigma, \sigma(e)) \mid \sigma \in \Sigma^R\}$
        $\mathit{mk\_Arith}(a, op, b) \to \{(\sigma, [\![op]\!](a', b')) \mid$
                                        $(\sigma, a') \in \mathcal{E}^{==}(a) \wedge$
                                          $(\sigma, b') \in \mathcal{E}^{==}(b) \wedge$
                                          $op \in \{+, -, \times\}\} \cup$
                                  $\{(\sigma, [\![\div]\!](a', b')) \mid$
                                        $(\sigma, a') \in \mathcal{E}^{==}(a) \wedge$
                                          $(\sigma, b') \in \mathcal{E}^{==}(b) \wedge$
                                          $op = \div \wedge b' \neq 0\}$
         $\mathit{mk\_Equality}(a, b) \to \{(\sigma, [\![=]\!](a', b')) \mid$
                                        $(\sigma, a') \in \mathcal{E}^{==}(a) \wedge (\sigma, b') \in \mathcal{E}^{==}(b)\}$
       $\mathit{mk\_StEquality}(a, b) \to \{(\sigma, [\![=]\!](a', b')) \mid$
                                        $(\sigma, a') \in \mathcal{E}^{==}(a) \wedge (\sigma, b') \in \mathcal{E}^{==}(b)\} \cup$
                                  $\{(\sigma, \mathbf{true}) \mid \sigma \in (\Sigma^R \setminus \mathbf{dom}\, \mathcal{E}^{==}(a)) \wedge$
                                          $\sigma \in (\Sigma^R \setminus \mathbf{dom}\, \mathcal{E}^{==}(b))\} \cup$
                                  $\{(\sigma, \mathbf{false}) \mid \sigma \in (\Sigma^R \setminus \mathbf{dom}\, \mathcal{E}^{==}(a)) \wedge$
                                          $\sigma \in \mathbf{dom}\, \mathcal{E}^{==}(b)\} \cup$
                                  $\{(\sigma, \mathbf{false}) \mid \sigma \in \mathbf{dom}\, \mathcal{E}^{==}(a) \wedge \sigma \in (\Sigma^R$
                                          $\setminus \mathbf{dom}\, \mathcal{E}^{==}(b))\}$
                 $\mathit{mk\_Not}(p) \to \{(\sigma, \mathbf{true}) \mid (\sigma, \mathbf{false}) \in \mathcal{E}^{==}(p)\} \cup$
                                  $\{(\sigma, \mathbf{false}) \mid (\sigma, \mathbf{true}) \in \mathcal{E}^{==}(p)\}$
               $\mathit{mk\_Or}(p, q) \to \{(\sigma, \mathbf{true}) \mid (\sigma, \mathbf{true}) \in \mathcal{E}^{==}(p)\} \cup$
                                  $\{(\sigma, \mathbf{true}) \mid (\sigma, \mathbf{true}) \in \mathcal{E}^{==}(q)\} \cup$
                                  $\{(\sigma, \mathbf{false}) \mid (\sigma, \mathbf{false}) \in \mathcal{E}^{==}(p) \wedge$
                                        $(\sigma, \mathbf{false}) \in \mathcal{E}^{==}(q)\}$
                              $\ldots$
    **end**
```

Figure 4.6: The $\mathcal{E}^{==}$ semantic function definition which defines the semantics for the approach of forcing all predicates to denote using strong equality (part 1)

```
$\mathcal{E}^{==} : \mathit{Expr}^{==} \to \mathcal{P}(\Sigma^R \times \mathit{Value})$
$\mathcal{E}^{==}(e) \quad \triangleq$
    $\textbf{cases } e \textbf{ of}$
                    $\ldots$
        $\mathit{mk\_Exists}(x, p) \to \{(\sigma, \textbf{true}) \mid$
                                    $\sigma \in \Sigma^R \wedge$
                                    $\textbf{true} \in$
                                      $\textbf{rng}\,(\{\sigma \dagger \{x \mapsto i\} \mid i{:}\,\mathbb{Z}\} \lhd \mathcal{E}^{==}(p))\} \cup$
                                $\{(\sigma, \textbf{false}) \mid$
                                    $\sigma \in \Sigma^R \wedge$
                                    $\textbf{rng}\,(\{\sigma \dagger \{x \mapsto i\} \mid i{:}\,\mathbb{Z}\} \lhd \mathcal{E}^{==}(p)) =$
                                      $\{\textbf{false}\}\}$
    $\mathit{mk\_FuncCall}(f, al) \to \{(\sigma, r) \mid \sigma \in \Sigma^R \wedge$
                                    $\forall i{:}\, \textbf{inds}\, al \cdot (\sigma, vl(i)) \in \mathcal{E}^{==}(al(i)) \wedge$
                                    $(vl, r) \in \sigma(f)\}$
    $\mathit{mk\_PredCall}(P, al) \to *$
    $\textbf{end}$
```

* Refer to the discussion on $\mathcal{E}^{==}$.

Figure 4.7: The $\mathcal{E}^{==}$ semantic function definition which defines the semantics for the approach of forcing all predicates to denote using strong equality (part 2)

$\mathcal{E}^{==}$ is also deterministic, see Lemma 1.

4.1.4 Weak Kleene Logic

In this approach if any operand to a logical operator is undefined then the entire expression is undefined. This is the strict interpretation that is given to the logical operators. Thus, defined results are only returned when both operands are defined.

Since partial functions and partial predicates can be defined in this approach the definition of Σ that is used in the LPF $\mathcal{E}$ semantic function can be used. The semantic function for this weak Kleene approach is presented in Figure 4.8.

The main changes made are to the disjunction logical operator case:

$$\begin{aligned}
&\mathit{mk\_Or}(p, q) \to \\
&\qquad \{(\sigma, \textbf{true}) \mid (\sigma, \textbf{true}) \in \mathcal{E}^W(p) \wedge \sigma \in \textbf{dom}\, \mathcal{E}^W(q)\} \cup \\
&\qquad \{(\sigma, \textbf{true}) \mid \sigma \in \textbf{dom}\, \mathcal{E}^W(p) \wedge (\sigma, \textbf{true}) \in \mathcal{E}^W(q)\} \cup \\
&\qquad \{(\sigma, \textbf{false}) \mid (\sigma, \textbf{false}) \in \mathcal{E}^W(p) \wedge (\sigma, \textbf{false}) \in \mathcal{E}^W(q)\}
\end{aligned}$$

since both operands p and q must denote in a given σ for a defined result to be returned.

Furthermore, the existential quantifier case changes:

$$
\begin{aligned}
&mk\_Exists(x, p) \to \\
&\qquad\qquad \{(\sigma, \mathbf{true}) \mid \sigma \in \Sigma \wedge \\
&\qquad\qquad\qquad \mathbf{let}\ d = \mathbf{rng}\,(\{\sigma \dagger \{x \mapsto i\} \mid i{:}\,\mathbb{Z}\} \lhd \mathcal{E}^W(p))\ \mathbf{in} \\
&\qquad\qquad\qquad d = \{\mathbf{true}\} \vee d = \{\mathbf{true}, \mathbf{false}\}\} \cup \\
&\qquad\qquad \{(\sigma, \mathbf{false}) \mid \sigma \in \Sigma \wedge \\
&\qquad\qquad\qquad \mathbf{rng}\,(\{\sigma \dagger \{x \mapsto i\} \mid i{:}\,\mathbb{Z}\} \lhd \mathcal{E}^W(p)) = \{\mathbf{false}\}\}
\end{aligned}
$$

since an undefined result ("gap") gives rise to a "gap" it is necessary to ensure that in the true case that p is always defined for any $i \in \mathbb{Z}$. The false case remains unchanged from the definition that it was given in the $\mathcal{E}$ semantic function definition.

The other logical operator that is present in the $\mathcal{E}^W$ semantic function, namely the negation logical operator case, does not need to be changed from what was presented in the $\mathcal{E}$ semantic function definition.

4.1.5 McCarthy's Conditional Operators

The first variable in the conditional expressions is usually referred to as the "inevitable variable" because, if it is undefined, then the entire expression is undefined since conditional expressions are strict in their first argument. This means that disjunction and conjunction are no longer commutative.

Additionally, quantifiers are problematic with respect to undefined values. Thus, $\exists i{:}\,\{0,1\} \cdot i/i = 1$ may not have the same truth value as $1/1 = 1 \vee 0/0 = 1$. While propositional logic operators are strict in their first operand, an order of evaluation for quantifiers is not at all obvious in McCarthy's conditional operator approach. One solution could be to take the strict interpretation of the quantifiers (as in the weak Kleene approach $\mathcal{E}^W$) to complete the semantic function for McCarthy's conditional operator approach, but the evaluation order for the quantifiers in McCarthy's conditional operator approach is generally devised to match the underlying application/program.

Since partial functions and partial predicates can be defined in this approach the definition of Σ that is used in the LPF $\mathcal{E}$ semantic function can be used. The semantic function for McCarthy's approach is presented in Figure 4.9.

The main change made for the definition of $\mathcal{E}^M$ is to the disjunction case:

$$
\begin{aligned}
&mk\_Or(p, q) \to \\
&\qquad\qquad \{(\sigma, \mathbf{true}) \mid (\sigma, \mathbf{true}) \in \mathcal{E}^M(p)\} \cup \\
&\qquad\qquad \{(\sigma, \mathbf{true}) \mid (\sigma, \mathbf{false}) \in \mathcal{E}^M(p) \wedge (\sigma, \mathbf{true}) \in \mathcal{E}^M(q)\} \cup \\
&\qquad\qquad \{(\sigma, \mathbf{false}) \mid (\sigma, \mathbf{false}) \in \mathcal{E}^M(p) \wedge (\sigma, \mathbf{false}) \in \mathcal{E}^M(q)\}
\end{aligned}
$$

to provide an operand evaluation policy that is strict in the first operand. Notice that the interpretation that is given to the negation logical operator is

```
$\mathcal{E}^W : Expr^C \to \mathcal{P}(\Sigma \times Value)$

$\mathcal{E}^W(e) \;\; \triangleq$
    cases $e$ of
                 $e \in Value \to \{(\sigma, e) \mid \sigma \in \Sigma\}$
                  $e \in Prop \to \{(\sigma, \sigma(e)) \mid \sigma \in \Sigma \wedge e \in \mathbf{dom}\, \sigma\}$
                   $e \in Var \to \{(\sigma, \sigma(e)) \mid \sigma \in \Sigma\}$
      $mk\text{-}Arith(a, op, b) \to \{(\sigma, [\![op]\!](a', b')) \mid$
                                    $(\sigma, a') \in \mathcal{E}^W(a) \wedge (\sigma, b') \in \mathcal{E}^W(b) \wedge$
                                      $op \in \{+, -, \times\}\} \cup$
                                $\{(\sigma, [\![\div]\!](a', b')) \mid$
                                    $(\sigma, a') \in \mathcal{E}^W(a) \wedge (\sigma, b') \in \mathcal{E}^W(b) \wedge$
                                      $op = \div \wedge b' \neq 0\}$
      $mk\text{-}Equality(a, b) \to \{(\sigma, [\![=]\!](a', b')) \mid$
                                    $(\sigma, a') \in \mathcal{E}^W(a) \wedge (\sigma, b') \in \mathcal{E}^W(b)\}$
               $mk\text{-}Not(p) \to \{(\sigma, \mathbf{true}) \mid (\sigma, \mathbf{false}) \in \mathcal{E}^W(p)\} \cup$
                                $\{(\sigma, \mathbf{false}) \mid (\sigma, \mathbf{true}) \in \mathcal{E}^W(p)\}$
             $mk\text{-}Or(p, q) \to \{(\sigma, \mathbf{true}) \mid (\sigma, \mathbf{true}) \in \mathcal{E}^W(p) \wedge$
                                    $\sigma \in \mathbf{dom}\, \mathcal{E}^W(q)\} \cup$
                                $\{(\sigma, \mathbf{true}) \mid \sigma \in \mathbf{dom}\, \mathcal{E}^W(p) \wedge$
                                    $(\sigma, \mathbf{true}) \in \mathcal{E}^W(q)\} \cup$
                                $\{(\sigma, \mathbf{false}) \mid (\sigma, \mathbf{false}) \in \mathcal{E}^W(p) \wedge$
                                    $(\sigma, \mathbf{false}) \in \mathcal{E}^W(q)\}$
        $mk\text{-}Exists(x, p) \to \{(\sigma, \mathbf{true}) \mid$
                                    $\sigma \in \Sigma \wedge$
                                    let $d =$
                                      $\mathbf{rng}\, (\{\sigma \dagger \{x \mapsto i\} \mid i{:}\, \mathbb{Z}\} \lhd \mathcal{E}^W(p))$
                                        in
                                          $d = \{\mathbf{true}\} \vee$
                                            $d = \{\mathbf{true}, \mathbf{false}\}\} \cup$
                                $\{(\sigma, \mathbf{false}) \mid$
                                    $\sigma \in \Sigma \wedge$
                                    $\mathbf{rng}\, (\{\sigma \dagger \{x \mapsto i\} \mid i{:}\, \mathbb{Z}\} \lhd \mathcal{E}^W(p)) =$
                                      $\{\mathbf{false}\}\}$
    $mk\text{-}FuncCall(f, al),$
     $mk\text{-}PredCall(f, al) \to \{(\sigma, r) \mid$
                                    $f \in (Fn \cup Pred) \wedge$
                                    $\sigma \in \Sigma \wedge$
                                    $\forall i{:}\, \mathbf{inds}\; al \cdot (\sigma, vl(i)) \in \mathcal{E}^W(al(i)) \wedge$
                                    $(vl, r) \in \sigma(f)\}$
    end
```

Figure 4.8: The $\mathcal{E}^W$ semantic function definition which defines the semantics of the weak Kleene approach

the same as in the $\mathcal{E}$ semantic function definition and as in the $\mathcal{E}^W$ semantic function definition.

4.1.6 Łukasiewicz's Logic

This approach differs from the LPF (strong Kleene) approach $\mathcal{E}$ for the propositional operators, since a different interpretation is given to the implication logical operator. Since $p \ \Rightarrow_{\text{Ł}} q$ is not logically equivalent to $\neg_{\text{Ł}} p \vee_{\text{Ł}} q$, the $\mathcal{E}^{\text{Ł}}$ semantic function will need to include an implication case.

First it is necessary to extend Expr^C to take into account the implication logical operator:

$$\mathit{Expr}^{\text{Ł}} = \mathit{Expr}^C \mid \mathit{Implies}$$

where the abstract syntax and the context condition for *Implies* is the same as for *Or*.

The $\mathcal{E}^{\text{Ł}}$ semantic function is presented in Figure 4.10 and in Figure 4.11. Since implication can no longer be defined as $\neg\, p \vee_{\text{Ł}} q$ this logical operator needs adding to $\mathcal{E}^{\text{Ł}}$. The implication case in the $\mathcal{E}^{\text{Ł}}$ semantic function is defined as:

$$\begin{array}{l}
\mathit{mk\_Implies}(p,q) \to \\
\qquad\qquad \{(\sigma, \mathbf{true}) \mid (\sigma, \mathbf{false}) \in \mathcal{E}^{\text{Ł}}(p)\} \cup \\
\qquad\qquad \{(\sigma, \mathbf{true}) \mid (\sigma, \mathbf{true}) \in \mathcal{E}^{\text{Ł}}(q)\} \cup \\
\qquad\qquad \{(\sigma, \mathbf{true}) \mid \sigma \in (\Sigma \backslash \mathbf{dom}\, \mathcal{E}^{\text{Ł}}(p)) \wedge \sigma \in (\Sigma \backslash \mathbf{dom}\, \mathcal{E}^{\text{Ł}}(q))\} \cup \\
\qquad\qquad \{(\sigma, \mathbf{false}) \mid (\sigma, \mathbf{true}) \in \mathcal{E}^{\text{Ł}}(p) \wedge (\sigma, \mathbf{false}) \in \mathcal{E}^{\text{Ł}}(q)\}
\end{array}$$

where the final true case considers every $\sigma \in \Sigma$ where both operands p and q are undefined.

A case does not need introducing for the non-monotone $\Leftrightarrow_{\text{Ł}}$ logical operator, since the syntactic definition to $(p \ \Rightarrow_{\text{Ł}} q) \wedge_{\text{Ł}} (q \ \Rightarrow_{\text{Ł}} p)$ holds. Recall that $\bot_{\mathbb{B}} \ \Leftrightarrow_{\text{Ł}} \bot_{\mathbb{B}}$ is true.

The rest of the expression cases in the $\mathcal{E}^{\text{Ł}}$ semantic function are the same as they are defined in the $\mathcal{E}$ semantic function, but with rewriting $\mathcal{E}$ to $\mathcal{E}^{\text{Ł}}$ in the different expression construct cases.

4.1.7 Bochvar's External Logic

In this approach the results of formulae are forced to take one of two values true or false.

Since partial functions and partial predicates can be defined in this approach the definition of Σ that is used in the LPF $\mathcal{E}$ semantic function can be used. The semantic function for Bochvar's external approach is presented in Figure 4.12 and in Figure 4.13, where Expr^B is defined as:

$$\mathit{Expr}^B = \mathit{Expr}^C \mid \lceil \mid \rceil$$

```
$\mathcal{E}^M : Expr^C \to \mathcal{P}(\Sigma \times Value)$
$\mathcal{E}^M(e) \quad \triangleq$
    **cases** $e$ **of**
                  $e \in Value \to \{(\sigma, e) \mid \sigma \in \Sigma\}$
                   $e \in Prop \to \{(\sigma, \sigma(e)) \mid \sigma \in \Sigma \land e \in \mathbf{dom}\, \sigma\}$
                    $e \in Var \to \{(\sigma, \sigma(e)) \mid \sigma \in \Sigma\}$
       $mk\_Arith(a, op, b) \to \{(\sigma, [\![op]\!](a', b')) \mid$
                                 $(\sigma, a') \in \mathcal{E}^M(a) \land (\sigma, b') \in \mathcal{E}^M(b) \land$
                                   $op \in \{+, -, \times\}\} \cup$
                             $\{(\sigma, [\![\div]\!](a', b')) \mid$
                                 $(\sigma, a') \in \mathcal{E}^M(a) \land (\sigma, b') \in \mathcal{E}^M(b) \land$
                                   $op = \div \land b' \neq 0\}$
        $mk\_Equality(a, b) \to \{(\sigma, [\![=]\!](a', b')) \mid$
                                 $(\sigma, a') \in \mathcal{E}^M(a) \land (\sigma, b') \in \mathcal{E}^M(b)\}$
                $mk\_Not(p) \to \{(\sigma, \mathbf{true}) \mid (\sigma, \mathbf{false}) \in \mathcal{E}^M(p)\} \cup$
                             $\{(\sigma, \mathbf{false}) \mid (\sigma, \mathbf{true}) \in \mathcal{E}^M(p)\}$
              $mk\_Or(p, q) \to \{(\sigma, \mathbf{true}) \mid (\sigma, \mathbf{true}) \in \mathcal{E}^M(p)\} \cup$
                             $\{(\sigma, \mathbf{true}) \mid (\sigma, \mathbf{false}) \in \mathcal{E}^M(p) \land$
                                 $(\sigma, \mathbf{true}) \in \mathcal{E}^M(q)\} \cup$
                             $\{(\sigma, \mathbf{false}) \mid (\sigma, \mathbf{false}) \in \mathcal{E}^M(p) \land$
                                 $(\sigma, \mathbf{false}) \in \mathcal{E}^M(q)\}$
          $mk\_Exists(x, p) \to *$
    $mk\_FuncCall(f, al),$
     $mk\_PredCall(f, al) \to \{(\sigma, r) \mid$
                                 $f \in (Fn \cup Pred) \land$
                                 $\sigma \in \Sigma \land$
                                 $\forall i{:}\, \mathbf{inds}\; al \cdot (\sigma, vl(i)) \in \mathcal{E}^M(al(i)) \land$
                                 $(vl, r) \in \sigma(f)\}$
    **end**
```

* Depends on the underlying application/program, or it could be defined as in the $mk\_Exists(x, p)$ expression case of the $\mathcal{E}^W$ semantic function.

Figure 4.9: The $\mathcal{E}^M$ semantic function definition which defines the semantics of McCarthy's conditional operators approach

$$
\begin{array}{l}
\mathcal{E}^{Ł} : \mathit{Expr}^{Ł} \to \mathcal{P}(\Sigma \times \mathit{Value}) \\[1ex]
\mathcal{E}^{Ł}(e) \quad \triangleq \\
\quad \textbf{cases } e \textbf{ of} \\
\quad\quad \begin{array}{rl}
e \in \mathit{Value} \to & \{(\sigma, e) \mid \sigma \in \Sigma\} \\
e \in \mathit{Prop} \to & \{(\sigma, \sigma(e)) \mid \sigma \in \Sigma \wedge e \in \mathbf{dom}\, \sigma\} \\
e \in \mathit{Var} \to & \{(\sigma, \sigma(e)) \mid \sigma \in \Sigma\} \\
\mathit{mk\_Arith}(a, op, b) \to & \{(\sigma, [\![op]\!](a', b')) \mid \\
& \qquad (\sigma, a') \in \mathcal{E}^{Ł}(a) \wedge (\sigma, b') \in \mathcal{E}^{Ł}(b) \wedge \\
& \qquad\quad op \in \{+, -, \times\}\} \cup \\
& \{(\sigma, [\![\div]\!](a', b')) \mid \\
& \qquad (\sigma, a') \in \mathcal{E}^{Ł}(a) \wedge (\sigma, b') \in \mathcal{E}^{Ł}(b) \wedge \\
& \qquad\quad \wedge\ op = \div \wedge b' \neq 0\} \\
\mathit{mk\_Equality}(a, b) \to & \{(\sigma, [\![=]\!](a', b')) \mid \\
& \qquad (\sigma, a') \in \mathcal{E}^{Ł}(a) \wedge (\sigma, b') \in \mathcal{E}^{Ł}(b)\} \\
\mathit{mk\_Not}(p) \to & \{(\sigma, \mathbf{true}) \mid (\sigma, \mathbf{false}) \in \mathcal{E}^{Ł}(p)\} \cup \\
& \{(\sigma, \mathbf{false}) \mid (\sigma, \mathbf{true}) \in \mathcal{E}^{Ł}(p)\} \\
\mathit{mk\_Or}(p, q) \to & \{(\sigma, \mathbf{true}) \mid (\sigma, \mathbf{true}) \in \mathcal{E}^{Ł}(p)\} \cup \\
& \{(\sigma, \mathbf{true}) \mid (\sigma, \mathbf{true}) \in \mathcal{E}^{Ł}(q)\} \cup \\
& \{(\sigma, \mathbf{false}) \mid (\sigma, \mathbf{false}) \in \mathcal{E}^{Ł}(p) \wedge \\
& \qquad (\sigma, \mathbf{false}) \in \mathcal{E}^{Ł}(q)\} \\
\ldots &
\end{array} \\
\quad \textbf{end}
\end{array}
$$

Figure 4.10: The $\mathcal{E}^{Ł}$ semantic function definition which defines the semantics of Lukasiewicz's approach (part 1)

```
$\mathcal{E}^{Ł} : \mathit{Expr}^{Ł} \to \mathcal{P}(\Sigma \times \mathit{Value})$

$\mathcal{E}^{Ł}(e) \quad \triangleq$
    $\mathbf{cases}\ e\ \mathbf{of}$
                     $\ldots$
      $\mathit{mk\_Implies}(p, q) \to \{(\sigma, \mathbf{true}) \mid (\sigma, \mathbf{false}) \in \mathcal{E}^{Ł}(p)\} \cup$
                              $\{(\sigma, \mathbf{true}) \mid (\sigma, \mathbf{true}) \in \mathcal{E}^{Ł}(q)\} \cup$
                              $\{(\sigma, \mathbf{true}) \mid \sigma \in (\Sigma \setminus \mathbf{dom}\,\mathcal{E}^{Ł}(p)) \wedge$
                                   $\sigma \in (\Sigma \setminus \mathbf{dom}\,\mathcal{E}^{Ł}(q))\} \cup$
                              $\{(\sigma, \mathbf{false}) \mid (\sigma, \mathbf{true}) \in \mathcal{E}^{Ł}(p) \wedge$
                                   $(\sigma, \mathbf{false}) \in \mathcal{E}^{Ł}(q)\}$
       $\mathit{mk\_Exists}(x, p) \to \{(\sigma, \mathbf{true}) \mid$
                                   $\sigma \in \Sigma \wedge$
                                   $\mathbf{true} \in$
                                     $\mathbf{rng}\,(\{\sigma \dagger \{x \mapsto i\} \mid i : \mathbb{Z}\} \lhd \mathcal{E}^{Ł}(p))\} \cup$
                              $\{(\sigma, \mathbf{false}) \mid$
                                   $\sigma \in \Sigma \wedge$
                                   $\mathbf{rng}\,(\{\sigma \dagger \{x \mapsto i\} \mid i : \mathbb{Z}\} \lhd \mathcal{E}^{Ł}(p)) =$
                                     $\{\mathbf{false}\}\}$
    $\mathit{mk\_FuncCall}(f, al)$,
     $\mathit{mk\_PredCall}(f, al) \to \{(\sigma, r) \mid$
                                   $f \in (\mathit{Fn} \cup \mathit{Pred}) \wedge$
                                   $\sigma \in \Sigma \wedge$
                                   $\forall i : \mathbf{inds}\ al \cdot (\sigma, vl(i)) \in \mathcal{E}^{Ł}(al(i)) \wedge$
                                   $(vl, r) \in \sigma(f)\}$
    $\mathbf{end}$
```

Figure 4.11: The $\mathcal{E}^{Ł}$ semantic function definition which defines the semantics of Lukasiewicz's approach (part 2)

where the abstract syntax and the context conditions for both $\lceil$ and $\rceil$ are defined as for δ and Δ.

In the $\mathcal{E}^B$ semantic function $\lceil$ is defined as:

$$
\begin{array}{l}
mk\text{-}\lceil(p) \rightarrow \\
\qquad \{(\sigma, \mathbf{true}) \mid (\sigma, \mathbf{true}) \in \mathcal{E}^B(p)\} \cup \\
\qquad \{(\sigma, \mathbf{false}) \mid (\sigma, \mathbf{false}) \in \mathcal{E}^B(p)\} \cup \\
\qquad \{(\sigma, \mathbf{false}) \mid \sigma \in (\Sigma \setminus \mathbf{dom}\, \mathcal{E}^B(p))\}
\end{array}
$$

where true is returned when p is true, and false is returned in the other two possible interpretations for p, that is, when p is false or undefined. The $\rceil$ operator is defined in a similar way.

The negation, disjunction, and the existential quantifier expression cases all need re-interpreting in the $\mathcal{E}^B$ semantic function definition, since no "gaps" should arise from these operators. For instance, the negation logical operator requires a third set:

$$
\begin{array}{l}
mk\text{-}Not(p) \rightarrow \\
\qquad \{(\sigma, \mathbf{true}) \mid (\sigma, \mathbf{false}) \in \mathcal{E}^B(p)\} \cup \\
\qquad \{(\sigma, \mathbf{true}) \mid \sigma \in (\Sigma \setminus \mathbf{dom}\, \mathcal{E}^B(p))\} \cup \\
\qquad \{(\sigma, \mathbf{false}) \mid (\sigma, \mathbf{true}) \in \mathcal{E}^B(p)\}
\end{array}
$$

The alternative would be to define negation in the same way as the weak Kleene/Bochvar's internal system negation operator in the $\mathcal{E}^W$ semantic function definition, but with the use of $\lceil$:

$$
\begin{array}{l}
mk\text{-}Not(p) \rightarrow \\
\qquad \{(\sigma, \mathbf{true}) \mid (\sigma, \mathbf{false}) \in \mathcal{E}^B(mk\text{-}\lceil(p))\} \cup \\
\qquad \{(\sigma, \mathbf{false}) \mid (\sigma, \mathbf{true}) \in \mathcal{E}^B(mk\text{-}\lceil(p))\}
\end{array}
$$

This alternate approach defines such expression constructs in the $\mathcal{E}^B$ semantic function as they are defined in [Boc81] through the definitions of weak Kleene/Bochvar's internal system and the use of $\lceil$. For instance, $p \vee_B q$ can be defined as: $\lceil p \vee_W \lceil q$.

Disjunction could be defined either as:

$$
\begin{aligned}
&mk\_Or(p,q) \rightarrow \\
&\qquad\qquad \{(\sigma,\mathbf{true}) \mid (\sigma,\mathbf{true}) \in \mathcal{E}^B(p)\} \cup \\
&\qquad\qquad \{(\sigma,\mathbf{true}) \mid (\sigma,\mathbf{true}) \in \mathcal{E}^B(q)\} \cup \\
&\qquad\qquad \{(\sigma,\mathbf{false}) \mid (\sigma,\mathbf{false}) \in \mathcal{E}^B(p) \wedge \\
&\qquad\qquad\qquad \sigma \notin \mathbf{dom}\,\mathcal{E}^B(q)\} \cup \\
&\qquad\qquad \{(\sigma,\mathbf{false}) \mid \sigma \notin \mathbf{dom}\,\mathcal{E}^B(p) \wedge \\
&\qquad\qquad\qquad (\sigma,\mathbf{false}) \in \mathcal{E}^B(q)\} \cup \\
&\qquad\qquad \{(\sigma,\mathbf{false}) \mid (\sigma,\mathbf{false}) \in \mathcal{E}^B(p) \wedge \\
&\qquad\qquad\qquad (\sigma,\mathbf{false}) \in \mathcal{E}^B(q)\} \cup \\
&\qquad\qquad \{(\sigma,\mathbf{false}) \mid \sigma \in (\Sigma \setminus \mathbf{dom}\,\mathcal{E}^B(p)) \wedge \\
&\qquad\qquad\qquad \sigma \in (\Sigma \setminus \mathbf{dom}\,\mathcal{E}^B(q))\}
\end{aligned}
$$

or alternatively through the use of the $\lceil$ logical operator in a way close to the definition of disjunction in the $\mathcal{E}$ semantic function definition:

$$
\begin{aligned}
&mk\_Or(p,q) \rightarrow \\
&\qquad\qquad \{(\sigma,\mathbf{true}) \mid (\sigma,\mathbf{true}) \in \mathcal{E}^B(mk\_\lceil(p))\} \cup \\
&\qquad\qquad \{(\sigma,\mathbf{true}) \mid (\sigma,\mathbf{true}) \in \mathcal{E}^B(mk\_\lceil(q))\} \cup \\
&\qquad\qquad \{(\sigma,\mathbf{false}) \mid (\sigma,\mathbf{false}) \in \mathcal{E}^B(mk\_\lceil(p)) \wedge \\
&\qquad\qquad\qquad (\sigma,\mathbf{false}) \in \mathcal{E}^B(mk\_\rceil(q))\}
\end{aligned}
$$

Additionally, the quantifier case will need changing as illustrated in Figure 4.13.

A $p \leftrightarrow_B q$ logical operator is also available which is true when p has the same strength as q. This logical operator is similar to $p \;\Leftrightarrow\; q$ (in LPF) except that formulae such as $\mathbf{true} \leftrightarrow_B \bot_{\mathbb{B}}$ are false, and formulae such as $\bot_{\mathbb{B}} \leftrightarrow_B \mathbf{false}$ and $\bot_{\mathbb{B}} \leftrightarrow_B \bot_{\mathbb{B}}$ are true. The logical operator $\leftrightarrow_B$ can be defined in the standard way through $(p \Rightarrow_B q) \wedge_B (q \Rightarrow_B p)$.

4.1.8 Concluding Remarks

Only small changes are needed to be made to the LPF $\mathcal{E}$ semantic function definition presented in Section 3.4 to be able to formally define the semantics of the other approaches to coping with partial terms. The structure of $\mathcal{E}$ has been maintained in each of the $\mathcal{E}^i$ semantic function definitions.

To further illustrate the cases that needed changing, Table 4.1 outlines the cases that changed for each $\mathcal{E}^i$ semantic function definition from what was originally presented in the $\mathcal{E}$ semantic function definition.

Recall that:

- $\mathcal{E}^R$ - Relations;
- $\mathcal{E}^C$ - Underspecification;

```
$\mathcal{E}^B : \mathit{Expr}^B \to \mathcal{P}(\Sigma \times \mathit{Value})$
$\mathcal{E}^B(e) \quad \triangleq$
    cases $e$ of
                $e \in \mathit{Value} \to \{(\sigma, e) \mid \sigma \in \Sigma\}$
                 $e \in \mathit{Prop} \to \{(\sigma, \sigma(e)) \mid \sigma \in \Sigma \land e \in \mathbf{dom}\, \sigma\}$
                  $e \in \mathit{Var} \to \{(\sigma, \sigma(e)) \mid \sigma \in \Sigma\}$
    $\mathit{mk\_Arith}(a, op, b) \to \{(\sigma, [\![op]\!](a', b')) \mid$
                                    $(\sigma, a') \in \mathcal{E}^B(a) \land (\sigma, b') \in \mathcal{E}^B(b) \land$
                                      $op \in \{+, -, \times\}\} \cup$
                              $\{(\sigma, [\![\div]\!](a', b')) \mid$
                                    $(\sigma, a') \in \mathcal{E}^B(a) \land (\sigma, b') \in \mathcal{E}^B(b) \land$
                                      $op = \div \land b' \neq 0\}$
    $\mathit{mk\_Equality}(a, b) \to \{(\sigma, [\![=]\!](a', b')) \mid$
                                    $(\sigma, a') \in \mathcal{E}^B(a) \land (\sigma, b') \in \mathcal{E}^B(b)\}$
                  $\mathit{mk\_}\lceil(p) \to \{(\sigma, \mathbf{true}) \mid (\sigma, \mathbf{true}) \in \mathcal{E}^B(p)\} \cup$
                              $\{(\sigma, \mathbf{false}) \mid (\sigma, \mathbf{false}) \in \mathcal{E}^B(p)\} \cup$
                              $\{(\sigma, \mathbf{false}) \mid \sigma \in (\Sigma \setminus \mathbf{dom}\, \mathcal{E}^B(p))\}$
                  $\mathit{mk\_}\rceil(p) \to \{(\sigma, \mathbf{true}) \mid (\sigma, \mathbf{false}) \in \mathcal{E}^B(p)\} \cup$
                              $\{(\sigma, \mathbf{false}) \mid (\sigma, \mathbf{true}) \in \mathcal{E}^B(p)\} \cup$
                              $\{(\sigma, \mathbf{false}) \mid \sigma \in (\Sigma \setminus \mathbf{dom}\, \mathcal{E}^B(p))\}$
             $\mathit{mk\_Not}(p) \to \{(\sigma, \mathbf{true}) \mid (\sigma, \mathbf{false}) \in \mathcal{E}^B(p)\} \cup$
                              $\{(\sigma, \mathbf{true}) \mid \sigma \in (\Sigma \setminus \mathbf{dom}\, \mathcal{E}^B(p))\} \cup$
                              $\{(\sigma, \mathbf{false}) \mid (\sigma, \mathbf{true}) \in \mathcal{E}^B(p)\}$
                      ...
    end
```

Figure 4.12: The $\mathcal{E}^B$ semantic function definition which defines the semantics of Bochvar's External approach (part 1)

```
$\mathcal{E}^B : Expr^B \to \mathcal{P}(\Sigma \times Value)$
$\mathcal{E}^B(e) \;\; \triangleq$
    **cases** $e$ **of**
                  $\ldots$
          $mk\_Or(p, q) \to \{(\sigma, \mathbf{true}) \mid (\sigma, \mathbf{true}) \in \mathcal{E}^B(p)\} \cup$
                          $\{(\sigma, \mathbf{true}) \mid (\sigma, \mathbf{true}) \in \mathcal{E}^B(q)\} \cup$
                          $\{(\sigma, \mathbf{false}) \mid (\sigma, \mathbf{false}) \in \mathcal{E}^B(p) \wedge$
                              $\sigma \notin \mathbf{dom}\, \mathcal{E}^B(q)\} \cup$
                          $\{(\sigma, \mathbf{false}) \mid \sigma \notin \mathbf{dom}\, \mathcal{E}^B(p) \wedge$
                              $(\sigma, \mathbf{false}) \in \mathcal{E}^B(q)\} \cup$
                          $\{(\sigma, \mathbf{false}) \mid (\sigma, \mathbf{false}) \in \mathcal{E}^B(p) \wedge$
                              $(\sigma, \mathbf{false}) \in \mathcal{E}^B(q)\} \cup$
                          $\{(\sigma, \mathbf{false}) \mid \sigma \in (\Sigma \setminus \mathbf{dom}\, \mathcal{E}^B(p)) \wedge$
                              $\sigma \in (\Sigma \setminus \mathbf{dom}\, \mathcal{E}^B(q))\}$
        $mk\_Exists(x, p) \to \{(\sigma, \mathbf{true}) \mid$
                              $\sigma \in \Sigma \wedge$
                              $\mathbf{true} \in$
                                $\mathbf{rng}\,(\{\sigma \dagger \{x \mapsto i\} \mid i \colon \mathbb{Z}\} \lhd \mathcal{E}^B(p))\} \cup$
                          $\{(\sigma, \mathbf{false}) \mid$
                              $\sigma \in \Sigma \wedge$
                              $\mathbf{true} \notin$
                                $\mathbf{rng}\,(\{\sigma \dagger \{x \mapsto i\} \mid i \colon \mathbb{Z}\} \lhd \mathcal{E}^B(p))\}$
    $mk\_FuncCall(f, al),$
     $mk\_PredCall(f, al) \to \{(\sigma, r) \mid$
                              $f \in (Fn \cup Pred) \wedge$
                              $\sigma \in \Sigma \wedge$
                              $\forall i \colon \mathbf{inds}\; al \cdot (\sigma, vl(i)) \in \mathcal{E}^B(al(i)) \wedge$
                              $(vl, r) \in \sigma(f)\}$
    **end**
```

Figure 4.13: The $\mathcal{E}^B$ semantic function definition which defines the semantics of Bochvar's External approach (part 2)

	$\mathcal{E}^R$	$\mathcal{E}^C$	$\mathcal{E}^D$	$\mathcal{E}^\exists$	$\mathcal{E}^{==}$	$\mathcal{E}^W$	$\mathcal{E}^M$	$\mathcal{E}^Ł$	$\mathcal{E}^B$
Σ change	✓	✓	✓	✓	✓				
Expr change$^{a)}$	✓			✓	✓			✓	✓
$e \in$ *Value*									
$e \in$ *Prop*	✓	✓	✓	✓	✓				
$e \in$ *Var*									
mk_Arith(a, op, b)	✓	✓	✓						
mk_Equality$(a, b)^{b)}$				✓	✓				
mk_Not(p)									✓
mk_Or(p, q)						✓	✓		✓
mk_Exists(x, p)						✓	✓		✓
mk_FuncCall(f, al)	✓								
mk_PredCall(P, al)	✓			✓	✓				
Additional$^{c)}$								✓	✓

a) Not considering the omission of the δ logical operator, the Δ logical operator, and the conditional expression construct.
b) Includes the addition of other equality expression constructs.
c) Are any additional expression constructs needed, excluding any additional equality constructs as this is considered in another case.

Table 4.1: An outline of the changes made to the LPF semantic function definition to define the other semantic function definitions

- $\mathcal{E}^D$ - Overspecification;
- $\mathcal{E}^\exists$ - Existential equality;
- $\mathcal{E}^{==}$ - Strong equality;
- $\mathcal{E}^W$ - Weak Kleene logic (Bochvar's Internal logic operators);
- $\mathcal{E}^M$ - McCarthy's conditional operators;
- $\mathcal{E}^Ł$ - Łukasiewicz's logic; and
- $\mathcal{E}^B$ - Bochvar's External logic.

4.2 Comparing the Sequent Interpretations

In this section a formal semantic comparison of the different interpretations that can be given to a sequent $\vdash$, is presented. The LPF $\mathcal{E}$ semantic function definition and the definition of Σ used in the $\mathcal{E}$ semantic function definition are used to perform the comparison. Recall that the four different interpretations that can be given to a sequent are *SS*, *SW*, *WW* and *WS*. These four different sequent interpretations are briefly introduced in Section 2.2.5.

In terms of the $\mathcal{E}$ semantic function, logical consequence ($\Gamma \vdash e$) can be defined through the following set definitions, where Γ (a set of assumptions) is a possibly empty set of expressions, that is, $\Gamma = \{e_1, \ldots, e_n\}$, and where e (the goal) is a single formula. Two sets S and W will first be presented as they are used in the definition of multiple sequent interpretations.

The set S represents all interpretations where each assumption is true:

$$S = \{\sigma \mid \sigma \in \Sigma \land (\sigma, \mathbf{true}) \in \mathcal{E}(e_1 \land \ldots \land e_n)\}$$

The set W represents all interpretations where the assumptions are undefined:

$$W = \{\sigma \mid \sigma \in (\Sigma \setminus \mathbf{dom}\, \mathcal{E}(e_1 \land \ldots \land e_n))\}$$

4.2.1 SS

This is the sequent interpretation that is used in LPF. Given all of the interpretations where each assumption is true, then e should be true in all of those interpretations, that is, there exists no σ such that all formulae in Γ are true, while the formula e is false or undefined:[2]

$$S \subseteq \{\sigma \mid \sigma \in \Sigma \land (\sigma, \mathbf{true}) \in \mathcal{E}(e)\}$$

4.2.2 SW

With the SW sequent interpretation the formula e should be true or undefined in all of those interpretations where each assumption is true S:

$$S \subseteq (\{\sigma \mid \sigma \in \Sigma \land (\sigma, \mathbf{true}) \in \mathcal{E}(e)\} \cup \{\sigma \mid \sigma \in (\Sigma \setminus \mathbf{dom}\, \mathcal{E}(e))\})$$

4.2.3 WW

With the WW sequent interpretation the same condition as in the SW sequent interpretation must hold. But additionally, the following condition must hold. Given all of those interpretations where the assumptions are undefined, then e should be true or undefined in all of those interpretations:

$$W \subseteq (\{\sigma \mid \sigma \in \Sigma \land (\sigma, \mathbf{true}) \in \mathcal{E}(e)\} \cup \{\sigma \mid \sigma \in (\Sigma \setminus \mathbf{dom}\, \mathcal{E}(e))\})$$

Thus (non-formally):

$$(SW) \land (WW)$$

[2]While in two-valued classical logic this is equivalent to the assertion $e_1 \land \ldots \land e_n \Rightarrow e$, this does not hold in LPF, cf. $\Rightarrow$-I.

4.2.4 WS

With the *WS* sequent interpretation the same condition as in the *SS* sequent interpretation must hold. But additionally, the following condition must hold. Given all of those interpretations where the assumptions are undefined, then e should be true in all of those interpretations:

$W \subseteq \{\sigma \mid \sigma \in \Sigma \wedge (\sigma, \mathbf{true}) \in \mathcal{E}(e)\}$

Thus:

$$(SS) \wedge (WS)$$

4.3 Comparisons between the Different Approaches to Coping with Partial Terms

In this section two comparisons are presented on the different approaches to coping with logical formulae that can contain references to partial terms. The first comparison is on the values that are denoted in each of the different approaches for a number of different expressions, including a term expression, predicate expressions, as well as quantified expressions. The second comparison is performed on the non-classical logic approaches considered. Here different properties of the non-classical logics are compared, which include checking whether for example the logics are monotone and whether the standard CNF transformations still hold/are still compatible with the semantics of the non-classical logics that are being considered. The $\mathcal{E}^{[i]}$ semantic function definitions aided greatly with performing such comparisons.

4.3.1 Comparison 1: The Values Denoted in the Different Approaches

The expressions used in this comparison are:

1. $zero(-1)$;
2. $zero(-1) = 0$;
3. $zero(-1) = zero(-1)$;
4. Property 1.1 (the *zero* function implication example):

$$\forall i{:}\, \mathbb{Z} \cdot i \geq 0 \;\Rightarrow\; zero(i) = 0$$

5. Property 1.3 (the *zero* function disjunction example):

$$\forall i{:}\, \mathbb{Z} \cdot zero(i) = 0 \vee zero(-i) = 0.$$

This set of five expressions ensures that term expressions, predicate expressions and quantified expressions are all accounted for. Additionally, these expressions are adequate to illustrate the main differences between the various approaches to coping with logical formulae that can contain references to partial terms.

Table 4.2 illustrates the effect of coping with undefinedness in different expression constructs. This is done by illustrating the resulting values of the five different expressions listed above which contain references to partial terms in each of the different approaches considered.

Note that any changes to expressions 1 to 5 are made as necessary for the different approaches that are considered. For example, when the quantifier bounds approach is considered the quantification becomes over the set of natural numbers $\mathbb{N}$, and when the strong equality approach is considered the notion of equality becomes $==$.

	1	2	3	4	5
Relations	**false**	**false**	**false**	**true**	**true**
Quantifier Bounds	$\bot_{\mathbb{Z}}^{a)}$	$\bot_{\mathbb{B}}$	$\bot_{\mathbb{B}}$	**true**	*denied*$^{b)}$
The WD approach	*rejected*$^{c)}$	*rejected*	*rejected*	**true**	**true**
Underspecification (determined value)	$\top_{\mathbb{Z}}^{d)}$	$\top_{\mathbb{B}}$	**true**	**true**	**true**
Underspecification (arbitrary value)	$\top_{\mathbb{Z}}$	$\top_{\mathbb{B}}$	$\top_{\mathbb{B}}$	**true**	**true**
Overspecification	$0^{e)}$	**true**	**true**	**true**	**true**
PFOL (plus the existential equality approach)	$\bot_{\mathbb{Z}}$	**false**	**false**	**true**	**true**
Strong equality	$\bot_{\mathbb{Z}}$	**false**	**true**	**true**	**true**
LCF	$\bot_{\mathbb{Z}}$	**false**	**true**	**true**	**true**
Predicate Underspecification	$\bot_{\mathbb{Z}}$	$\top_{\mathbb{B}}$	**true**	**true**	**true**
Weak Kleene (Bochvar Internal)	$\bot_{\mathbb{Z}}$	$\bot_{\mathbb{B}}$	$\bot_{\mathbb{B}}$	$\bot_{\mathbb{B}}$	$\bot_{\mathbb{B}}$
McCarthy's Conditional Operators	$\bot_{\mathbb{Z}}$	$\bot_{\mathbb{B}}$	$\bot_{\mathbb{B}}$	**true**	$\bot_{\mathbb{B}}$
LPF (strong Kleene logic)	$\bot_{\mathbb{Z}}$	$\bot_{\mathbb{B}}$	$\bot_{\mathbb{B}}$	**true**	**true**
Łukasiewicz's approach	$\bot_{\mathbb{Z}}$	$\bot_{\mathbb{B}}$	$\bot_{\mathbb{B}}$	**true**	**true**
Bochvar's External approach	$\bot_{\mathbb{Z}}$	$\bot_{\mathbb{B}}$	$\bot_{\mathbb{B}}$	**true**	**true**

a) $\bot_T$: Stands for a "gap" where T is the type of the expression.
b) Since i is of the type natural number $\mathbb{N}$, $-i$ cannot be expressed. If $-i$ ($i \in \mathbb{N}$), is defined to be 0 for example, then this example is **true**.
c) *rejected*: The Well-definedness conditions cannot be proved.
d) $\top_T$: Stands for an unspecified/undetermined value of the type T. Such an unspecified value is defined but the actual value is not known.
e) Assuming that here the value that is returned when the *zero* function is applied with an argument from outside of its defined domain ($i < 0$) is 0.

Table 4.2: A comparison of numerous approaches to coping with partial terms

4.3.2 Comparison 2: Non-classical Logic Comparisons

All of the non-classical logic approaches discussed are now considered under the assumption that the sequent interpretation used is the *SS* sequent interpretation (justified in Section 2.3) as is used in LPF. The comparison is performed without any use of the Δ logical operator. Recall that:

- $\mathcal{E}^W$ - Weak Kleene logic (Bochvar's Internal logic operators);
- $\mathcal{E}^M$ - McCarthy's conditional operators;
- $\mathcal{E}$ - LPF (strong Kleene logic);
- $\mathcal{E}^{Ł}$ - Łukasiewicz's logic; and
- $\mathcal{E}^B$ - Bochvar's External logic.

The results are presented in Table 4.3, where the terms used in the table are described below:

- $\Rightarrow$ contrapositive: Is $p \Rightarrow q$ equivalent to $\neg\, q \Rightarrow \neg\, p$?
- $\vee$ and $\wedge$ commutativity: Are the disjunction and conjunction operators commutative?
- Law of the excluded middle: Does the law of the excluded middle hold?
- Quantifiers: Can the quantifiers be written in terms of disjunctions and conjunctions?
- Deduction theorem: Does the deduction theorem hold?
- Trivial sequent: Does the trivial sequent hold?
- CNF transformations: Do the standard CNF transformations hold?
- PNF transformations: Do the standard PNF transformations hold?
- The standard syntactic definitions: Do the standard syntactic definitions: $p \wedge q$ being equivalent to $\neg\,(\neg\, p \vee \neg\, q)$; $p \Rightarrow q$ being equivalent to $\neg\, p \vee q$; $p \Leftrightarrow q$ being equivalent to $(p \Rightarrow q) \wedge (q \Rightarrow p)$; and $\forall i \cdot p$ being equivalent to $\neg\, \exists i \cdot \neg\, p$ hold?
- Tautologies: Are there any tautologies in the language?
- Monotone operators: Are the logical operators monotone?

	$\mathcal{E}^W$	$\mathcal{E}^M$	$\mathcal{E}$	$\mathcal{E}^{\text{Ł}}$	$\mathcal{E}^B$
$\Rightarrow$ contrapositive	✓		✓	✓	✓
$\vee$ and $\wedge$ commutativity	✓		✓	✓	✓
Law of the excluded middle					✓
Quantifiers	✓	a)	✓	✓	✓
Deduction theorem					✓
CNF transformations	✓		✓	✓ b)	✓ c)
PNF transformations	✓		✓	✓	✓
The standard syntactic definitions	✓	✓	✓		
Tautologies			d)	✓	✓
Monotone operators	✓	✓	✓		
Uniformity		✓ e)	✓	✓	✓
Normality	✓	✓	✓	✓	✓

a) Depends on the underlying order of evaluation that is given to the quantifiers. If the strict interpretation is given then this follows in $\mathcal{E}^M$ as it does in $\mathcal{E}^W$.
b) Except that replacing $p \Rightarrow_{\text{Ł}} q$ with $\neg_{\text{Ł}} p \vee_{\text{Ł}} q$ cannot be done, and a more sophisticated translation is needed, refer to Section 4.4.
c) For instance, p is not logically equivalent to $\neg_B \neg_B p$, $p \vee_B p$ is not logically equivalent to p, and the absorption properties do not maintain logical equivalence. Undefinedness is treated as false when given as an operand to a logical operator.
d) Recall that there can be no tautologies in $\mathcal{E}$, without the use of the Δ logical operator.
e) In a restricted way compared to for example $\mathcal{E}$, (the first operand only in $\mathcal{E}^M$).

Table 4.3: A comparison of three-valued logic approaches to coping with partial terms

- Uniformity: Are the results as defined as possible, that is, can a result be determined from a single operand where possible (for example, the truth of one disjunct is sufficient for the truth of the disjunction)? and

- Normality: When all operands to the logical operators are defined, that is, true and false, do the logical operators yield the same result as the two-valued classical logical operators will yield?

Note that there is no case when a logical value can be determined for the $\Leftrightarrow$ logical operator from a single logical value (operand); thus this logical operator is uniform by default.

As can be seen from Table 4.3, with the $\mathcal{E}^M$ semantic function fewer of the desired properties hold in comparison to the other approaches. Basic logical properties/laws are lost, which means that using familiar notations such as clausal form becomes more complicated due to the sequential (strict in the first operand) semantics of the logical operators in this approach. Questions are also undoubtedly raised about quantifiers in this approach, as discussed earlier.

The other approaches maintain more of the standard logical laws. However, with the $\mathcal{E}^W$ approach the logical operators are strict (undefinedness in any operand leads to overall undefinedness), while the other approaches employ non-strict logical operators. Thus, the other approaches allow for more defined results to be returned, which is undoubtedly favoured.

The $\mathcal{E}^{\mathbf{L}}$ approach is the closest approach to the $\mathcal{E}$ approach. What differs is that $\bot_{\mathbb{B}} \;\Rightarrow_{\mathbf{L}} \bot_{\mathbb{B}}$ is true in the $\mathcal{E}^{\mathbf{L}}$ approach to coping with partial terms. This change alone brings about a number of distinct differences between the two approaches.

In the $\mathcal{E}^{\mathbf{L}}$ approach, tautologies (due to the implication logical operator) can be constructed. However, this is at the expense of standard syntactic definitions not holding and the loss of monotonicity. A loss of standard syntactic definitions holding, ensures that any reasoning that takes place in such a three-valued logic becomes even less familiar for a user who may be familiar with two-valued classical logic. The loss of monotonicity is a drawback as results can be contradicted through further evaluations.

The $\mathcal{E}^B$ approach looks pleasing, but again suffers from the lack of monotonicity.

The $\mathcal{E}$ approach appears like a favourable compromise. The logical operators are the strongest possible monotonic extension of the familiar two-valued classical logic logical operators. Standard logical laws are maintained in this approach. This ensures that while the three-valued logic is unfamiliar, it is not a too drastic change from that of two-valued classical logic, compared to the other approaches.

4.4 Relationships between the Different Approaches to Coping with Partial Terms and LPF

The $\mathcal{E}^{[i]}$ semantic function definitions provided above can be made use of to aid in pinpointing relationships between the different approaches to coping with logical formulae that can contain references to partial terms, in particular how formulae can be translated between the different approaches. The focus is on relationships between the different approaches considered and the preferred approach of LPF.

This is beneficial because for instance, McCarthy's conditional operator logic approach (with the sequential interpretation of binary logic operators) is used in tool support such as in the VDM Toolset [ELL94] and in the Overture toolset [LBF+10] (as discussed in Section 2.4), and LPF (with the parallel evaluation of the binary logic operators) has been favoured here for proof support. LPF as shown has more desirable logical properties to be used for proof support, as documented in Table 4.3 in Section 4.3.

Recall that all of the logical operators in the non-classical logic approaches considered, coincide with the corresponding two-valued classical logic logical operators when all of their corresponding operands are defined.

4.4.1 Relationship with Two-Valued Classical Logic using Overspecification

Recall that the overspecification approach is where a known defined value is returned by a partial function, when it is applied with arguments from outside of its defined domain.

Since all of the proof rules in LPF are also valid proof rules in two-valued classical logic and since LPF is normal, it follows that any theorem of LPF is a theorem in two-valued classical logic, providing every partial function and so on is overspecified. In other words if $\Gamma \vdash p$ in LPF, then $\Gamma \vdash p$ in two-valued classical logic.

However, it does not hold the other way around. For instance:

$$p \vee \neg p$$

is not a theorem of LPF.

A theorem of two-valued classical logic with the overspecification semantics needs translating from:

$$p$$

to:

$$p \vee \neg \Delta p$$

For instance, if:

$$1/0 = 0$$

is a theorem in the overspecification approach in two-valued classical logic then it needs translating to:

$$1/0 = 0 \vee \neg \Delta(1/0 = 0)$$

to be a theorem in LPF.

This is the same as stating:

$$\delta p \vdash p$$

The relationship with the underspecification approach follows in a similar way.

However, such a relationship does not follow in the same way when considering Łukasiewicz's logic. A theorem of Łuksiewicz's logic may not be a theorem of two-valued classical logic with overspecified functions. Consider, 1/0 being undefined the following is a theorem in Łukasiewicz's approach:

$$1/0 = 0 \ \Rightarrow_{\text{Ł}} 1/0 = 1$$

If division is overspecified so that 1/0 yields 0, then in classical logic:

$$\mathbf{true} \ \Rightarrow \ \mathbf{false}$$

which is false.

4.4.2 Relationship with the Well-Definedness Approach

A formula is valid in the WD approach (see Section 2.2.2) if $WD(e)$ is valid in two-valued logic and e is valid in two-valued logic. In the WD approach one has to prove WD and validity separately. In LPF one proves validity and definedness at the same time.

4.4.3 Relationship with Two-Valued Classical Logic using Strong Equality

In [FJ08] the authors explore the relationship between theorems provable in LPF using weak equality and two-valued classical logic using existential equality, providing the translations from the one approach to the other and vice versa (note that they use δ instead of Δ, but give δ the semantics of Δ). Since in this thesis weak equality, existential equality, and strong equality are discussed the translations between theorems provable in LPF using weak equality and two-valued classical logic using strong equality are defined.

Consider the formula:

$$\mathbf{true} \vee 1/0 = 1/0$$

which is a theorem in LPF. If the weak equality is rewritten as a strong equality, then the strong equality will guard the logical operator from a non-denoting truth value ($1/0 == 1/0$ will denote true), and thus this will also be a theorem in two-valued classical logic with strong equality. Therefore, to convert a theorem of LPF into two-valued classical logic with strong equality just rewrite any weak equality used to the strong equality equivalent. Doing so cannot cause the value of a formula to change from true to false.

Since the strong equality will always yield true or false the logical operators will be guarded from any partial terms. Of course the same is required for every relational operator/predicate and not just for the equality relational operator that is being considered in the examples.

However, not all theorems of two-valued classical logic with strong equality are theorems in LPF so "extra" work is required when converting the other way around into LPF. Consider:

$$\textbf{true} \Rightarrow 1/0 == 1/0$$

which evaluates to:

$$\textbf{true} \Rightarrow \textbf{true}$$

which further evaluates to **true**. However, in LPF (with weak equality) this example:

$$\textbf{true} \Rightarrow 1/0 = 1/0$$

evaluates to:

$$\textbf{true} \Rightarrow \perp_{\mathbb{B}}$$

which evaluates to $\perp_{\mathbb{B}}$.

When translating a theorem of two-valued classical logic with strong equality into LPF each strong equality:

$$a == b$$

requires rewriting to:

$$(a = b \wedge \Delta(a = b)) \vee (\neg\,\Delta(a = a) \wedge \neg\,\Delta(b = b))$$

to be a theorem of LPF.

4.4.4 Relationship with the Weak Kleene Approach

It should be clear that the weak Kleene logic is a weaker logic than LPF. In the weak Kleene logic if undefinedness occurs then undefinedness results. However, in the strong Kleene LPF logic undefinedness can be masked if enough information in certain circumstances is available from the other operand. For quantified expressions in the weak Kleene logic if the quantified formula is ever undefined then the quantified formula is undefined. This is not always the case for LPF, as illustrated in the $\mathcal{E}$ semantic function definition.

It follows that any theorem in the weak Kleene approach must be a theorem in the LPF approach. For example, $p \vee_W q$ is true if $(\sigma, \mathbf{true}) \in \mathcal{E}^W(p)$ and $(\sigma, \mathbf{true}) \in \mathcal{E}^W(q)$, or $(\sigma, \mathbf{true}) \in \mathcal{E}^W(p)$ and $(\sigma, \mathbf{false}) \in \mathcal{E}^W(q)$, or when $(\sigma, \mathbf{false}) \in \mathcal{E}^W(p)$ and when $(\sigma, \mathbf{true}) \in \mathcal{E}^W(q)$. In LPF $p \vee q$ is true if either $(\sigma, \mathbf{true}) \in \mathcal{E}(p)$ or when $(\sigma, \mathbf{true}) \in \mathcal{E}(q)$. It follows that $\delta p \wedge \delta q \wedge (p \vee q)$ must be true whenever $p \vee_W q$ is true, and the disjunction cases of $\mathcal{E}^W$ and $\mathcal{E}$ return the same result whenever both operands are defined.

Both $p \vee_W q$ and $p \vee q$ are false in the same cases when $(\sigma, \mathbf{false}) \in \mathcal{E}/\mathcal{E}^W(p)$ and when $(\sigma, \mathbf{false}) \in \mathcal{E}/\mathcal{E}^W(q)$. Additionally, if a quantifier is assigned a value of either true or false in $\mathcal{E}^W$, then $p(i)$ must be true or false for every $i \in \mathbb{Z}$, and $\mathcal{E}$ is defined to return the same value in such cases.

However, any theorem in the LPF approach may not be a theorem in the weak Kleene approach, as the logical operators are strict in the weak Kleene approach, but are non-strict in the LPF approach. In $\mathcal{E}$ defined results can be returned even in the presence of undefined operands in some cases.

Such logical operators are also those from the internal Bochvar approach. The relationship between Bochvar's external operators and Bochvar's internal operators are presented in [Boc81]. Each external logical operator can be defined in terms of the internal logical operators through use of the $\lceil$ logical operator and the $\rceil$ logical operator.

4.4.5 Relationship with the McCarthy Conditional Operator Approach

Consider the propositional subset of LPF and McCarthy's conditional operator approach. The two approaches differ as the former has a parallel evaluation of the binary logical operators, and the latter has a sequential evaluation of the binary logical operators. The two approaches have the same monotone ordering given to the logical operators, and the two approaches have the same definition for the negation logical operator.

If $p \vee_M q$ is true then $p \vee q$ is also true in LPF. In LPF $p \vee q$ is true if either $(\sigma, \mathbf{true}) \in \mathcal{E}(p)$ or $(\sigma, \mathbf{true}) \in \mathcal{E}(q)$. In McCarthy's approach, however,

$p \vee_M q$ is true if either $(\sigma, \mathbf{true}) \in \mathcal{E}^M(p)$, or both $(\sigma, \mathbf{false}) \in \mathcal{E}^M(p)$ and $(\sigma, \mathbf{true}) \in \mathcal{E}^M(q)$. Therefore, the proposition p must be defined if $p \vee_M q$ is true in McCarthy's approach ($\delta p \wedge (p \vee q)$ must be true). The LPF $\mathcal{E}$ semantic function definition is designed to return true in the two cases that the disjunction logical operator returns the value true in the $\mathcal{E}^M$ semantic function definition. The false cases match in the $\mathcal{E}^M$ semantic function definition and in the $\mathcal{E}$ semantic function definition.

Additionally, if the weak quantifier interpretation is used in McCarthy's approach then the theorem follows in the LPF approach as stated in the weak Kleene section above. Furthermore, no matter what quantifier interpretation is used in McCarthy's $\mathcal{E}^M$ approach then if a defined result is returned in $\mathcal{E}^M$ it will also be returned in the LPF $\mathcal{E}$ approach as the LPF quantifier interpretation is stronger. The false case of the existential quantifier must match in $\mathcal{E}$ and $\mathcal{E}^M$. If the quantified formula is always true then the existential quantifier cases in $\mathcal{E}$ and $\mathcal{E}^M$ must return the same result, as the order of evaluation in such an approach becomes irrelevant. However, if the quantified formula is ever undefined for some value, and it is true for at least the one value then the order of evaluation devised for the interpretation in $\mathcal{E}^M$ will determine whether the existential quantifier is true or undefined in the $\mathcal{E}^M$ semantic function definition. However, no matter what order of evaluation is given to the existential quantifier case in the $\mathcal{E}^M$ semantic function definition, if true is returned in $\mathcal{E}^M$ then true will be returned in $\mathcal{E}$ for the existential quantifier case, as $\mathcal{E}$ will return true if $p(i)$ is true for any $i \in \mathbb{Z}$. So, $p(i)$ can be undefined for any $i \in \mathbb{Z}$ as long as for at least one $i \in \mathbb{Z}$, $p(i)$ is true.

In summary an expression evaluation order may be present in $\mathcal{E}^M$, but no such order is devised in $\mathcal{E}$. Thus if a defined result true or false, is returned from the existential quantifier case in $\mathcal{E}^M$ then that same defined result true or false will be returned from the existential quantifier case in $\mathcal{E}$.

Consider the existential quantifier in McCarthy's approach to have the weak Kleene interpretation, then in LPF the following is a theorem:

$$\exists i \colon \mathbb{Z} \cdot \mathit{zero}(i) = 0$$

but it may not be a theorem in McCarthy's approach, since for example, $\mathit{zero}(-1) = 0$ is undefined. This forces the whole existentially quantified expression to be undefined, despite the fact that $\mathit{zero}(0) = 0$.

It follows that any theorem in McCarthy's approach must be a theorem in the LPF approach. However, any theorem in the LPF approach may not be a theorem in the McCarthy approach, due to the parallel evaluation nature of

LPF.

4.4.6 Relationship with Łukasiewicz's Approach

A difference between the $\mathcal{E}$ semantic function and the $\mathcal{E}^{Ł}$ semantic function comes down to the interpretation that is given to the implication logical operator. Recall that $\perp_{\mathbb{B}} \Rightarrow \perp_{\mathbb{B}}$ is undefined in $\mathcal{E}$, but $\perp_{\mathbb{B}} \Rightarrow_{Ł} \perp_{\mathbb{B}}$ is true in $\mathcal{E}^{Ł}$.

Therefore to translate a theorem of LPF into Łukasiewicz' approach no change is necessary. Every theorem of LPF is a theorem in Łukasiewicz's approach. However, when translating a theorem of Łukasiewicz's approach into a theorem of LPF, it is the case that every $p \Rightarrow_{Ł} q$ needs translating to the following formula:

$$(\neg\, p) \vee (q) \vee (\neg\, \Delta p \wedge \neg\, \Delta q)$$

Unfortunately, the monotonic δ operator cannot be used in this translation, because of translating to the weaker LPF logic.

In order to translate $p \Leftrightarrow_{Ł} q$ to LPF, since the propositional logical operator $\Leftrightarrow_{Ł}$ is not monotone in Łukasiewicz's approach, translate first to:

$$(p \Rightarrow_{Ł} q) \wedge_{Ł} (q \Rightarrow_{Ł} p)$$

and then to LPF as before.

4.4.7 Relationship with Bochvar's External Approach

Recall that after applying a logical operator in LPF undefinedness can result. However, in Bochvar's external approach after applying a logical operator a defined result is always returned.

Consider the propositional case first. Whenever a defined result true or false is returned through the $\mathcal{E}$ semantic function definition, the same defined result true or false will be returned through the $\mathcal{E}^{B}$ semantic function definition. Thus any propositional theorem of LPF is a theorem in Bochvar's external approach. However, the converse does not hold because $\mathcal{E}^{B}$ would return true for $\neg_{B}(0/0 = 0)$, while a "gap" would result in the $\mathcal{E}$ semantic function definition. The formula $\neg_{B} p$ needs translating to $\neg\, p \vee \neg\, (\Delta p)$ in LPF.

Now consider the predicate case. If the existentially quantified expression $p(i)$ is true for any $i \in \mathbb{Z}$ then the $\mathcal{E}$ semantic function definition will return true and the $\mathcal{E}^{B}$ semantic function definition will also return true. The quantified expression $p(i)$ needs to be false for all $i \in \mathbb{Z}$ for the truth value false to be returned by the $\mathcal{E}$ semantic function definition. However, in the existential quantifier case of $\mathcal{E}^B$ false can be returned in more cases. If a defined result is returned through the existential quantifier case of $\mathcal{E}$ then the same defined result will be returned through the existential quantifier case of $\mathcal{E}^B$. However, the converse may not hold.

4.4.8 Concluding Remarks

A trend between the non-classical logic approaches can be identified. The weak Kleene (Bochvar internal) approach is weaker than McCarthy's logic, which itself is weaker than the strong Kleene (LPF) logic, which is weaker than Łukasiewicz's logic, which is weaker than Bochvar's external approach logic. Theorems can easily be translated from a weaker logic to a stronger logic. However, the situation changes if moving theorems the other way around.

Any theorem of weak Kleene logic (where undefinedness gives rise to undefinedness) must also be a theorem of McCarthy's logic, but the converse does not hold. Consider first the propositional case, where only the binary operators change. In the former **true** $\vee \perp_{\mathbb{B}}$ is not a theorem but it is in the latter. Thus one cannot take an arbitrary theorem from McCarthy's approach and put it straight into the weak Kleene approach. In the McCarthy approach the propositional operators are *strengthened* so that results can be determined in certain circumstances if the first operand is defined even if the second operand is undefined.

Consider the predicate case, in the weak Kleene approach if the quantified formula (the predicate) is undefined for at least the one quantified value then the entire quantified expression is undefined. This interpretation is sometimes given in McCarthy's approach. However, in McCarthy's approach another order of evaluation could be given to the quantifiers. Thus again one cannot translate immediately from McCarthy's approach into the weak Kleene approach. But if a defined result is given to an existential quantifier in the weak Kleene approach then no matter what value is assigned to the quantified variable the predicate must be defined. Thus a theorem involving a quantifier in the weak Kleene is a theorem in the McCarthy approach as well, but the converse does not hold.

A theorem of the McCarthy approach must also be a theorem of LPF, but the converse does not hold. Consider the propositional case, the difference between the two approaches to coping with partial terms is that the former employs a sequential interpretation and the latter employs a parallel evaluation of the logical operators. Thus while $\perp_{\mathbb{B}} \vee$ **true** is a theorem of LPF it is not a theorem of McCarthy's approach. Any theorem of McCarthy's approach must be a theorem of LPF, because for the set of all true cases of the logical operators in McCarthy's approach as defined in the $\mathcal{E}^M$ semantic function definition are a proper subset of the true cases in the $\mathcal{E}$ semantic function definition, ($\mathcal{E}$ will return true/false in all cases that $\mathcal{E}^M$ returns true/false, it is just that $\mathcal{E}$ can return true/false in more cases).

Consider, the predicate case. An existential quantifier must be false in the same way in both the $\mathcal{E}^M$ and the $\mathcal{E}$ approaches. However, an existential quantifier is true in LPF regardless of whether for some value assigned to the quantified variable the predicate is undefined, as long as the predicate is true for some quantified value. Thus no matter what order of evaluation is given to the quantifiers in McCarthy's approach the stronger LPF interpretation of the quantifiers will ensure that the same result is returned. However, the contrary cannot hold.

A theorem of the LPF approach must be a theorem of Łukasiewicz's approach, but the converse does not hold. The $\mathcal{E}$ semantic function and the $\mathcal{E}^{\text{Ł}}$ semantic function are the same other than for the implication expression evaluation case. The $\mathcal{E}^{\text{Ł}}$ semantic function implication case is stronger than in the $\mathcal{E}$ semantic function since $\perp_{\mathbb{B}} \ \Rightarrow_{\text{Ł}} \perp_{\mathbb{B}}$ is a theorem in $\mathcal{E}^{\text{Ł}}$ but $\perp_{\mathbb{B}} \ \Rightarrow \ \perp_{\mathbb{B}}$ is not a theorem in $\mathcal{E}$.

A theorem of LPF and Łuksiewicz's approach must also be a theorem of Bochvar's external approach, but the converses do not hold. Consider the LPF case first. Whenever $\mathcal{E}$ returns a defined result then $\mathcal{E}^B$ returns the same defined result. The cases when the $\mathcal{E}^B$ semantic function definition returns a defined result can be seen to be a proper subset of the cases when the $\mathcal{E}$ semantic function definition returns a defined result. This applies not only to the propositional cases, but to the predicate cases.

For the Łukasiewicz's case, as mentioned the difference is the implication case that needed defining in $\mathcal{E}^{\text{Ł}}$. In the $\mathcal{E}^B$ semantic function definition $\perp_{\mathbb{B}} \Rightarrow_{\text{Ł}} \perp_{\mathbb{B}}$ is true and the same result will be returned by the $\mathcal{E}^B$ semantic function definition. Such an implication case did not need defining in $\mathcal{E}^B$ because $p \Rightarrow_B q$ can be defined as $\neg_B p \vee_B q$. This is because the negation of undefined is true, and true or anything is true in the $\mathcal{E}^B$ semantic function definition.

In summary if a defined result is returned in $\mathcal{E}^W$ then this same result will be returned in $\mathcal{E}^M$. Additionally, if a defined result is returned in $\mathcal{E}^M$ then this same result will be returned in $\mathcal{E}$. A result from a weaker monotonic logic will also be returned in a stronger monotonic logic. Thus these three monotonic logics are ordered.

Furthermore, if a defined result is returned in $\mathcal{E}$ then this same result will be returned in $\mathcal{E}^{\text{Ł}}$. Additionally, if a defined result is returned in $\mathcal{E}^{\text{Ł}}$ then this same result will be returned in $\mathcal{E}^B$. The difference comes down to the fact that both $\mathcal{E}^{\text{Ł}}$ and $\mathcal{E}^{B}$ are not monotonic. For instance, $\perp_{\mathbb{B}} \vee_{B}$ **false** is false, but if functions complete leaving **true** $\vee_{B}$ **false**, then the result will change from false to true. Also, $\perp_{\mathbb{B}} \;\Rightarrow_{\text{Ł}} \perp_{\mathbb{B}}$ is true, but if functions complete leaving **true** $\Rightarrow_{B}$**false**, then a result can change from true to false. This is a major reason against utilising a non-monotonic logic for the work in Chapter 6 (and for reasoning about the properties of partial functions in general). The stronger of the monotonic logics $\mathcal{E}$ is deemed to be the most satisfactory, as the logic is as defined *as possible*.

4.5 Conclusions

This chapter showed how the $\mathcal{E}$ semantic function definition alongside the Σ variable and definition map, which formally capture the semantics of LPF, can be adapted to formally capture the semantics of numerous other approaches to coping with partial terms. The use of such definitions is proposed as a way of formally comparing the different approaches.

These $\mathcal{E}^{[i]}$ semantic function definitions provide a way of easily identifying the differences between the different approaches to coping with partial terms. They allow for the differences between the different approaches to be explained in terms of changes to $\mathcal{E}$ and to Σ. This is helped by only small changes needing to be made to move between the different $\mathcal{E}^{[i]}$ semantic function definitions.

The $\mathcal{E}^{[i]}$ semantic function definitions and the $\Sigma^{[i]}$ definitions were used as a basis with which to conduct some comparisons between the different approaches. Specifically, comparing the meaning of expressions written in different approaches, and comparing properties that hold in different non-classical logic approaches. Such definitions have greatly aided in performing such comparisons, since they in effect precisely and succinctly capture the crucial points and differences between the different approaches.

Issues regarding different logics can arise for example when combining different formal methods. Different formal methods are based upon different logics, and therefore utilise different approaches to coping with partial terms. The work in this chapter has looked at answering not only questions of how different approaches to coping with partial terms compare, but identifying how theorems can be moved between different approaches. Being able to move theorems between different proof tools/formal methods relies on identifying the differences between the approaches, and this work has focused on overcoming any mismatches in respect to the different treatments of coping with partial terms.

Chapter 6 takes the preferred approach of LPF and investigates applying proof procedures to it. The LPF $\mathcal{E}$ semantic function definition is used as the underlying basis to precisely define concepts, illustrate issues, and to conduct proofs of the modifications made to the proof procedures to cover LPF. It would be useful to also use these non-classical logic $\mathcal{E}^i$ semantic function definitions to aid in the modification of the proof procedures for the other non-classical logics considered, and to compare the amount of extra work that is brought into these proof procedures against that for LPF, due to the different semantics of the non-classical logic approaches. This topic is discussed further in Section 7.2 on future work. Mechanisations of the favoured LPF approach in Maude and in Isabelle are also considered in Chapter 5

Chapter 5

Mechanising LPF Semantic Definitions

Contents

5.1 Maude Mechanisation 141

5.1.1 An Introduction to Maude 142

5.1.2 Mechanising the SOS Definitions 144

5.2 Isabelle Mechanisation 169

5.3 Conclusions 182

Mechanisations of the SOS definitions in both the Maude term-rewriting system, and in the Isabelle proof assistant are considered in this chapter. The Maude term-rewriting mechanisation allows for expressions to be evaluated according to the semantics of LPF by tool. The Isabelle mechanisation allows for proofs of key properties to be conducted in LPF.

5.1 Maude Mechanisation

Both the big-step LPF SOS semantic definition and the small-step LPF SOS semantic definition can be used to form a basis of a mechanisation in the term rewriting system Maude [CDE$^+$07]. The focus of this section is on mechanising the SOS definitions presented in Section 3.3 in the Maude term rewriting system to allow for expressions to be evaluated by a tool, according to the semantics of LPF. The full big-step and small-step LPF semantic definitions are presented in Appendix A.

The work in this section on the mechanisation of the SOS definitions presented earlier provides some assurance that the definitions are *correct*, that is, that they accurately capture the semantics of LPF. Expressions can be evaluated by a tool and it can be shown that expressions evaluated through the tool do evaluate to the expected values, or in the case that undefined expressions are evaluated that no result, that is, a “gap” results. Up until this point example evaluations have only been performed completely manually as in Figure 3.7.

This section illustrates how to cross over from the theoretical definitions into a concrete mechanisation, and illustrates further how “gaps” are coped with, and the problems that "gaps" bring about. Further evidence as to the need for a small-step SOS definition to accurately define the semantics of LPF for evaluating expressions is provided by the work in this section. However, the use of the small-step SOS definition is at the expense of further rewrite steps/rule applications needing to be made compared to when using the big-step SOS definition. The extent of this is shown through considering the evaluation of a wide range of examples.

5.1.1 An Introduction to Maude

A brief introduction to Maude is presented first. For a more detailed overview of Maude the reader should refer to [CDE^{+}07]. Only the parts of Maude that are used in mechanising the SOS definitions are introduced here.

Two types of modules are used here: functional modules (`fmod`) and system modules (`mod`). A module may consist of a collection of sorts, operations, and equations. Additionally, a system module can contain a collection of rewrite rules.

Modules in Maude can import other modules to re-use operations etc. Modules are imported here by either:

```
protecting MODULE
```

or by:

```
including MODULE
```

The `protecting` keyword is used when no change is to be made to the imported module.

A sort defines a type of data and subsort relations can also be specified, e.g.:

```
sort INT .
sort NAT .

subsort NAT < INT .
```

Kinds are associated with sorts, where the kinds also contain any associated error terms, they are error supersorts.

Operations are declared in modules like:

```
op _+_ : INT INT -> INT .
```

Operations that are constructors are declared using the attribute `[ctor]` before the space before the period at the end of the line. Operations can also be flagged as associative and commutative by specifying the attributes `assoc` and `comm` respectively in between `[]` which are to occur again before the space before the period at the end of the line.

When defining equations and rewrite rules variables may be used which are declared in the following way:

```
var I : INT .
```

Variables are placeholders, they do not store specific values.

Equations are used to simplify expressions and a set of equations should be terminating and confluent. The equational logic underlying Maude is membership equational logic. Equations are defined in the following way:

```
eq 0 + I = I .
```

Conditional equations ensure that an equation is used for simplification only if its condition is satisfied, e.g.:

```
ceq 0 <= s(I) if 0 <= I .
```

Equations that have been defined in modules can be reduced by Maude using the `red` command, e.g.:

```
red 5 + -5 .
result Zero: 0
```

A set of rewrite rules does not need to be terminating and confluent. A rewrite rule is used to describe a transition between states, e.g.:

```
sort EXAMPLE .
op term1 : -> EXAMPLE .
op term2 : -> EXAMPLE.
rl [LABEL] : term1 => term2 .
```

Conditional rewrite rules are declared in the following way:

```
crl [LABEL] : term1 => term2 if BOOLEAN_CONDITION .
```

The `rew` command can be used to get Maude to execute rewrite rules:

```
rew term1 .
result EXAMPLE: term2
```

5.1.2 Mechanising the SOS Definitions

The mechanisation of the SOS rules into Maude follows the way they were introduced above. First the syntax will be defined, followed by defining the context conditions. This is followed by the semantic rules, first the big-step SOS rules and then the small-step SOS rules.

This has not been the first attempt at mechanising SOS definitions. The authors in [VMO06] mechanise a number of SOS definitions in Maude. This work extends that work in providing a comparison between the big-step and the small-step SOS definitions in terms of the number of rewrite comparisons that are necessary, mechanising definitions that define the semantics of a three-valued logic (showing how partial functions can be defined and coped with, and considering interpretations that need applying to rules for instance, those that define the disjunction logical operator, to define the parallel evaluation nature of LPF), and through the consideration of mechanising context conditions.

Constant Values

The first task is to define the two constant types used in the SOS definitions (both the big-step SOS definitions and the small-step SOS definitions). The Boolean values are defined in their own module and this is done to separate the LPF Boolean values from the default Boolean values:

```
fmod LPFBool is
  sort LPFBOOL .

  op LPFtrue : -> LPFBOOL [ctor] .
  op LPFfalse : -> LPFBOOL [ctor] .
endfm
```

where no explicit undefined value is defined, since undefinedness is represented as a “gap”.

The `LPFInt` module is implemented making use of the built-in `Int` module in Maude, where a new sort (type) `LPFINT` is defined. The equality operation is overwritten to return a `LPFBOOL` value instead of one of the default Boolean values, and this will also need doing to similar operators such $\geq$ etc. if they were part of the semantic definitions being implemented. Addition and subtraction etc. are already defined in the `Int` module and thus are available to be used with `LPFInt` operands.

```
fmod LPFInt is
  protecting LPFBool .
  protecting INT .

  sort LPFINT .

  subsort Int < LPFINT .

  op _eq_ : LPFINT LPFINT -> LPFBOOL [comm] .

  var m : LPFINT .
  var n : LPFINT .

  ceq m eq n = LPFtrue if (m == n) == true .
  ceq m eq n = LPFfalse if (m == n) == false .
endfm
```

Now the two basic types of the language are defined all of the operations that are called from within the Strachey brackets in the semantic definitions are available for use by the semantic rules that follow later.

Syntax

The `LPFExpr` module defines the syntax of all the expression constructs in *Expr*. A sort is given for `LPFEXPR`, where the values and the identifiers are defined as subsorts of this sort.

```
fmod LPFExpr is
  protecting STRING .
  protecting LPFBool .
  protecting LPFInt .

  sort LPFEXPR .

  sort VALUE .
  sort ID .

  subsort LPFBOOL < VALUE .
  subsort LPFINT < VALUE .

  subsort VALUE < LPFEXPR .

  sort VARID .
  sort PROPID .
  sort FUNID .
  sort PREDID .

  sort VALUEIDS .
  sort DEFINITIONIDS .

  subsort VARID < VALUEIDS .
  subsort PROPID < VALUEIDS .
  subsort FUNID < DEFINITIONIDS .
  subsort PREDID < DEFINITIONIDS .

  subsort VALUEIDS < ID .
  subsort DEFINITIONIDS < ID .

  subsort ID < LPFEXPR .

  ...
endfm
```

The identifiers are defined as strings in `LPFExpr`, with operations to go from strings to identifiers:

```
  op V(_) : String -> VARID .
```

where `V` is for variable (integer) identifiers. Operations are defined similarly for `P`, `F` and `Pr` for propositional identifiers, function identifiers, and predicate identifiers respectively. The use of `V`, `P`, `F`, and `Pr` takes care of the issue of ensuring that the four types of identifiers are disjoint. So, for instance, an integer variable `x` is written as `V("x")`, and a function identifier `zero` as `F("zero")`.

The syntax of the different expression constructs are then defined in the `LPFExpr` module as:

```
  sort ARITHOP .

  op PLUS : -> ARITHOP .
  op MINUS : -> ARITHOP .
  op MULT : -> ARITHOP .
  op DIV : -> ARITHOP .
  op ARITH(_, _, _) : LPFEXPR ARITHOP LPFEXPR -> LPFEXPR .
  op NOT(_) : LPFEXPR -> LPFEXPR .
  op OR(_, _) : LPFEXPR LPFEXPR -> LPFEXPR .
  op AND(_, _) : LPFEXPR LPFEXPR -> LPFEXPR .
  op EXISTS(_, _) : VARID LPFEXPR -> LPFEXPR .
  op FORALL(_, _) : VARID LPFEXPR -> LPFEXPR .
  op FUNCCALL(_, _) : FUNID LPFEXPR -> LPFEXPR .
```

and so on. Functions and predicates are here restricted to the one argument each. This helps to simply the mechanisation, but still allows for the issues surrounding undefinedness to be adequately illustrated. Removing such a restriction will be straightforward using the `List` module available in Maude.

The syntactic definitions are defined using equations in `LPFExpr`, e.g.:

```
  var V : VARID .
  var P : LPFEXPR .
  var Q : LPFEXPR .

  eq AND(P, Q) = NOT(OR(NOT(P), NOT(Q))) .
  eq FORALL(V, P) = NOT(EXISTS(V, NOT(P))) .
```

and so on.

Another module is used to define the function and the predicate definitions:

```
fmod LPFDefinitions is
  protecting LPFBool .
  protecting LPFExpr .

  sort FUNCDEF .
  sort DEFINITIONS .

  subsort FUNCDEF < DEFINITIONS .

  op FUNC(_, _) : VARID LPFEXPR -> FUNCDEF .

  var v : VARID .
  var e : LPFEXPR .

  op getParams(_) : DEFINITIONS -> VARID .
  eq getParams(FUNC(v, e)) = v .
  op getExpression(_) : DEFINITIONS -> LPFEXPR .
  eq getExpression(FUNC(v, e)) = e .

  op zeroFunction : -> FUNCDEF .
  eq zeroFunction =
      FUNC(V("i"),
           COND(EQUALITY(V("i"), 0),
           0,
           FUNCCALL(F("zero"), ARITH(V("i"), MINUS, 1)))) .
endfm
```

where predicates are defined in a similar way.

The operation `zeroFunction` is an example function definition that has been defined, and can be used later. Other function definitions can be defined in a similar way.

Context Conditions

The context conditions are now defined. First the names of the two types INT and BOOL are defined in a module:

```
fmod LPFTypes is
  sort BOOLTYPE .
  sort INTTYPE .
  sort TYPE .

  subsort BOOLTYPE < TYPE .
  subsort INTTYPE < TYPE .

  op BTYPE : -> BOOLTYPE .
  op ITYPE : -> INTTYPE .
  op ERROR : -> [TYPE] .
endfm
```

where the `BTYPE` operator and the `ITYPE` operator name the corresponding type, and the `ERROR` type is defined as an error term.

The module for the type map is defined using the `Map` module, as a map from sort `VALUEIDS` to the sort `TYPE`:

```
fmod LPFTypeMap is
  protecting LPFExpr .
  protecting LPFTypes .
  protecting MAP{VALUEIDS, TYPE} .

  op typeMap1 : -> Map{VALUEIDS, TYPE} .
  eq typeMap1 = insert(P("p"), BTYPE,
                insert(V("i"), ITYPE, empty)) .
endfm
```

The operation `typeMap1` is an example type map that has been defined for use later. Further type maps can be defined in a similar way.

In order to use `VALUEIDS` in a map the following definition:

```
view VALUEIDS from TRIV to LPFExpr is
  sort Elt to VALUEIDS .
endv
```

is required to allow for the instantiation of parameterised modules. This is required for all other modules that are used in a map and a list.

The `Def` (definitions) map is defined inside another module using another map:

```
fmod LPFDefMap is
  protecting LPFExpr .
  protecting LPFDefinitions .
  protecting MAP{DEFINITIONIDS, DEFINITIONS} .

  op defMap1 : -> Map{DEFINITIONIDS, DEFINITIONS} .
  eq defMap1 = insert(F("zero"), zeroFunction, empty) .
endfm
```

The operation `defMap1` is an example definition map defined for use later, which uses the `zeroFunction` function definition defined as an operation from earlier.

Now the context conditions can be defined. First a module that defines the context conditions for expressions is defined by defining a `wf-Expr` operation:

```
fmod LPFContextConditionsExpressions is
  protecting LPFBool .
  protecting LPFInt .
  protecting LPFExpr .
  protecting LPFDefinitions .
  protecting LPFTypes .
  protecting LPFTypeMap .
  protecting LPFDefMap .

  op wf-Expr(_, _, _) : LPFEXPR
                        Map{VALUEIDS, TYPE}
                        Map{DEFINITIONIDS, DEFINITIONS}
                            -> [TYPE] .
  ...
endfm
```

where all of the different expression cases that define the total `wf-Expr` function are defined using equations with the use of variables:

```
  var b : LPFBOOL .
  var i : LPFINT .

  var p : PROPID .
  var v : VARID .
  var f : FUNID .

  var e1 : LPFEXPR .
  var e2 : LPFEXPR .
  var e3 : LPFEXPR .

  var op : ARITHOP .

  var vars : Map{VALUEIDS, TYPE} .
  var defs : Map{DEFINITIONIDS, DEFINITIONS} .

  eq wf-Expr(b, vars, defs) = BTYPE .
  eq wf-Expr(i, vars, defs) = ITYPE .

  eq wf-Expr(p, vars, defs) =
     if $hasMapping(vars, p) then
        BTYPE
     else
        ERROR
     fi .

  eq wf-Expr(v, vars, defs) =
     if $hasMapping(vars, v) then
        ITYPE
     else
        ERROR
     fi .

  eq wf-Expr(ARITH(e1, op, e2), vars, defs) =
     if wf-Expr(e1, vars, defs) == ITYPE and
        wf-Expr(e2, vars, defs) == ITYPE then
    ITYPE
     else
        ERROR
     fi .

  eq wf-Expr(EQUALITY(e1, e2), vars, defs) =
     if wf-Expr(e1, vars, defs) == ITYPE and
        wf-Expr(e2, vars, defs) == ITYPE then
        BTYPE
     else
        ERROR
     fi .

  eq wf-Expr(COND(e1, e2, e3), vars, defs) =
     if wf-Expr(e1, vars, defs) == BTYPE and
        wf-Expr(e2, vars, defs) == ITYPE and
        wf-Expr(e3, vars, defs) == ITYPE then
        ITYPE
     else
        ERROR
     fi .

  eq wf-Expr(NOT(e1), vars, defs) =
     if wf-Expr(e1, vars, defs) == BTYPE then
        BTYPE
     else
        ERROR
     fi .

  eq wf-Expr(DELTA(e1), vars, defs) =
     if wf-Expr(e1, vars, defs) == BTYPE then
       BTYPE
     else
       ERROR
     fi .

   eq wf-Expr(OR(e1, e2), vars, defs) =
     if wf-Expr(e1, vars, defs) == BTYPE and
       wf-Expr(e2, vars, defs) == BTYPE then
       BTYPE
     else
       ERROR
     fi .

   eq wf-Expr(EXISTS(v, e1), vars, defs) =
     if wf-Expr(e1, insert(v, ITYPE, vars), defs) == BTYPE then
       BTYPE
     else
       ERROR
     fi .

   eq wf-Expr(FUNCCALL(f, e1), vars, defs) =
     if $hasMapping(defs, f) and
       wf-Expr(e1, vars, defs) == ITYPE then
       ITYPE
     else
       ERROR
     fi .
```

The `PREDCALL` case follows in a similar way.

The `wf-Func` operation is defined in another module:

```
fmod LPFContextConditionsDefinitions is
  protecting BOOL .
  protecting LPFBool .
  protecting LPFInt .
  protecting LPFExpr .
  protecting LPFTypes .
  protecting LPFTypeMap .
  protecting LPFDefMap .
  protecting LPFContextConditionsExpressions .

  var fun : FUNCDEF .
  var f : FUNID .
  var defs : Map{DEFINITIONIDS, DEFINITIONS} .

  op wf-Func(_, _) : FUNCDEF
                     Map{DEFINITIONIDS, DEFINITIONS} ->
                        Bool .
  eq wf-Func(fun, defs) =
      wf-Expr(getExpression(fun),
              insert(getParams(fun), ITYPE, empty),
              defs) == ITYPE .
endfm
```

and the `wf-Pred` context condition should be defined in the same way as the `wf-Func` context condition in the `LPFContextConditionsDefinitions` module.

The context conditions must be satisfied before using the semantic rules that follow. It is assumed that any expression, function, and predicate definition used with the following SOS rules satisfy the context conditions, that is, for any expression `e`:

```
  wf-Expr(e, vars, defs)
```

evaluates to `ITYPE`, or evaluates to `BTYPE`.

Additionally, any function definition `f` used:

```
  wf-Func(f, defs)
```

evaluates to `true`, and any predicate definition `P` used:

```
  wf-Pred(P, defs)
```

evaluates to `true`.

Note that a type map and a `Defs` map are assumed to coincide with a σ map, but a propositional identifier can be absent from the domain of a σ to allow for undefined propositional variables to be present, but such a propositional identifier must map to `BTYPE` in the corresponding type map to ensure the correct checking of expressions.

The following examples show the context conditions in action:

```
red wf-Expr(LPFtrue,typeMap1,defMap1) .
result BOOLTYPE: BTYPE

red wf-Expr(FUNCCALL(F("zero"), 0), typeMap1, defMap1) .
result INTTYPE: ITYPE

red wf-Expr(OR(LPFtrue, 1), typeMap1, defMap1) .
result [TYPE]: ERROR

red wf-Expr(FUNCCALL(F("zero"), LPFtrue), typeMap1, defMap1) .
result [TYPE]: ERROR

red wf-Func(zeroFunction, defMap1) .
result Bool: true
```

Semantic Objects

The semantic object σ is defined in a similar way to the other maps introduced:

```
fmod LPFSigma is
  protecting LPFExpr .
  protecting MAP{VALUEIDS, VALUE} .

  op sigma1 : -> Map{VALUEIDS, VALUE} .
  eq sigma1 = insert(V("i"), 1,
              insert(P("p"), LPFtrue, empty)) .
endfm
```

The decision has been taken to have a map σ that maps propositional identifiers to Boolean values and variable identifiers to integer variables, and to use the separate (already defined above) `Defs` map to map function identifiers to function definitions, and predicate identifiers to predicate definitions. The four maps defined in Σ in Section 3.3.2 are now split across two different maps, but the same data is still represented. The use of two maps simplifies this mechanisation, as a module already defined above for the context conditions can be re-used without any changes needing to be made.

Semantic Rules

The modules defined up to now have been functional modules. The SOS rules for both the big-step SOS rules and the small-step SOS rules will be defined in system modules. This is because each SOS rule is going to be defined as a rewrite rule. Recall that the set of SOS rules are not total, that is, they may not terminate when presented with "gaps". The rewrite rules are non-deterministic.

The big-step SOS rules are now introduced, where the semantic relation is defined as:

```
mod LPFBigStepSemantics is
  protecting LPFInt .
  protecting LPFBool .
  protecting LPFExpr .
  protecting LPFDefinitions .
  protecting LPFDefMap .
  protecting LPFSigma .

  sort BIGSTEPRELATION .

  subsort VALUE < BIGSTEPRELATION .

  op (_, _, _) -be> : LPFEXPR
      Map{VALUEIDS, VALUE}
      Map{DEFINITIONIDS, DEFINITIONS} -> [BIGSTEPRELATION] .
  ...
```

Recall that $\xrightarrow{e}$ in the big-step SOS definition is from $(Expr \times \Sigma)$ to *Value*.

In the `LPFBigStepSemantics` module numerous variable placeholders need introducing to allow for the SOS rules to be defined in Maude as rewrite rules:

```
  var a : LPFEXPR .
  var a' : LPFEXPR .
  var b : LPFEXPR .
  var b' : LPFEXPR .
  var p : LPFEXPR .
  var p' : LPFEXPR .
  var q : LPFEXPR .
  var q' : LPFEXPR .

  var v1 : VALUE .
  var v1' : VALUE .
  var v2 : VALUE .
  var v2' : VALUE .
```

```
var vId : VARID .
var pId : PROPID .
var fId : FUNID .
var PId : PREDID .

var sigma : Map{VALUEIDS, VALUE} .
var defs : Map{DEFINITIONIDS, DEFINITIONS} .
```

The big-step SOS rules are then defined as rewrite rules as follows:

```
rl [Value_E] : (v1, sigma, defs) -be> => v1 .

crl [Prop_E] : (pId, sigma, defs) -be> => sigma[pId]
     if $hasMapping(sigma, pId) .
rl [Var_E] : (vId, sigma, defs) -be> => sigma[vId] .

crl [Arith_E1] : (ARITH(a, PLUS, b), sigma, defs)
         -be> => (a' + b')
     if (a, sigma, defs) -be> => a' /\
        (b, sigma, defs) -be> => b' .

crl [ARITH_E2] : (ARITH(a, MINUS, b), sigma, defs)
         -be> => (a' - b')
     if (a, sigma, defs) -be> => a' /\
        (b, sigma, defs) -be> => b' .

crl [ARITH_E3] : (ARITH(a, MULT, b), sigma, defs)
         -be> => (a' * b')
     if (a, sigma, defs) -be> => a' /\
        (b, sigma, defs) -be> => b' .

crl [ARITH_E4] : (ARITH(a, DIV, b), sigma, defs)
         -be> => (a' quo b')
     if (a, sigma, defs) -be> => a' /\
        (b, sigma, defs) -be> => b' /\ b' =/= 0 .

crl [Equality_E] : (EQUALITY(a, b), sigma, defs)
         -be> => (a' eq b')
     if (a, sigma, defs) -be> => a' /\
        (b, sigma, defs) -be> => b' .
```

```
crl [Cond_E1] : (COND(p, a, b), sigma, defs) -be> => a’
     if (p, sigma, defs) -be> => LPFtrue /\
        (a, sigma, defs) -be> => a’ .

crl [Cond_E2] : (COND(p, a, b), sigma, defs) -be> => b’
     if (p, sigma, defs) -be> => LPFfalse /\
        (b, sigma, defs) -be> => b’ .

crl [Not_E1] : (NOT(p), sigma, defs) -be> => LPFfalse
     if (p, sigma, defs) -be> => LPFtrue .

crl [Not_e2] : (NOT(p), sigma, defs) -be> => LPFtrue
     if (p, sigma, defs) -be> => LPFfalse .

crl [delta_E1] : (DELTA(p), sigma, defs) -be> => LPFtrue
     if (p, sigma, defs) -be> => LPFtrue .

crl [delta_E2] : (DELTA(p), sigma, defs) -be> => LPFtrue
     if (p, sigma, defs) -be> => LPFfalse .

crl [Or_E1] : (OR(p, q), sigma, defs) -be> => LPFtrue
    if (p, sigma, defs) -be> => LPFtrue .

crl [Or_E2] : (OR(p, q), sigma, defs) -be> => LPFtrue
    if (q, sigma, defs) -be> => LPFtrue .

crl [Or_E3] : (OR(p, q), sigma, defs) -be> => LPFfalse
    if (p, sigma, defs) -be> => LPFfalse /\
       (q, sigma, defs) -be> => LPFfalse .

crl [FuncCall_E] : (FUNCCALL(fId, a), sigma, defs)
                   -be> => b’
    if (a, sigma, defs) -be> => a’ /\
       (getExpression(defs[fId]),
        insert(getParams(defs[fId]), a’, sigma), defs)
              -be> => b’ .

...
```

```
endm
```

The `PredCall_E` rule follows in a similar way to the `FuncCall_E` rule.

In the LPF SOS definitions an infinite number of premises are used to define the existential quantifier. Obviously, an infinite number of premises cannot be used in such a setting of a term-rewriting system. Thus here quantification is only over a finite set of integer values. Only the values `-1`, `0`, and `1` are used. As mentioned earlier the existential quantifier is defined essentially as a disjunction:

```
  crl [EXISTS_Ta] : (EXISTS(vId, p), sigma, defs)
                   -be> => LPFtrue
      if (p, insert(vId, -1, sigma), defs) -be> => LPFtrue .


  ...

  crl [EXISTS_F] : (EXISTS(vId, p), sigma, defs)
                  -be> => LPFfalse
      if (p, insert(vId, -1, sigma), defs) -be> => LPFfalse /\
         (p, insert(vId, 0, sigma), defs) -be> => LPFfalse /\
         (p, insert(vId, 1, sigma), defs) -be> => LPFfalse .
```

where the rules `EXISTS-Tb` and `EXISTS-Tc` follow closely to the `EXISTS-Ta` rule, but with `0` and `1` instead. Due to the evaluation nature of the big-step semantics the existential quantifier has the same problem as the disjunction logical operator, that was mentioned earlier. This is illustrated at the end of this section when comparing the big-step semantics in Maude with the small-step semantics in Maude. This ad-hoc big-step existential quantifier definition is only presented to be able to get some comparison results between the big-step semantics and the small-step semantics. Again the small-step semantic version is needed because control can get stuck with evaluating a term that fails to denote a proper value.

Some sample expression evaluations are:

```
rew (FUNCCALL(F("zero"), 0), sigma1, defMap1) -be> .
result Zero: 0

rew (OR(
        EQUALITY(FUNCCALL(F("zero"), 1), 0),
        EQUALITY(FUNCCALL(F("zero"), -1), 0)),
            sigma1, defMap1) -be> .
result LPFBOOL: LPFtrue
```

```
rew (OR(
        EQUALITY(FUNCCALL(F("zero"), -1), 0),
        EQUALITY(FUNCCALL(F("zero"), 1), 0)),
            sigma1, defMap1) -be> .
Fatal error: stack overflow.
```

The latter result is because of the big-step SOS definition that ensures that once the evaluation of an operand has begun, there is no way to move control (the evaluation) over to the other operand, and the evaluation of the non-denoting operand has started to be evaluated. This is the problem that was alluded to earlier when the big-step SOS definition was introduced to define the semantics of LPF. This is the problem that the small-step SOS definition overcomes. The small-step SOS definition is the preferred and necessary way of defining the semantics of LPF for such a mechanisation to be faithful to the parallel evaluation nature of LPF.

Also notice that in some cases no value is output since no further semantic rule can be applied:

```
rew (ARITH(1, DIV, 0), sigma1, defMap1) -be> .
result [FindResult,LPFEXPR,BIGSTEPRELATION]:
       (ARITH(1,DIV,0), ...) -be>
```

Trying to evaluate the expression `ARITH(1, DIV, 0)` terminates, but no value is given to the expression, the evaluation is *stuck*.

Before being able to define the small-step SOS rules, additional expression constructs need introducing:

- `FUNCINTER`;
- `PREDINTER` (virtually the same as `FUNCINTER`);
- `EXISTSINTER`; and
- `EXISTSPAIR`.

```
fmod LPFInterExpr is
  protecting LPFBool .
  protecting LPFInt .
  including LPFExpr .

  sort LPFINTEREXPR .
  sort FUNCINTER .

  subsort FUNCINTER < LPFINTEREXPR .
  subsort LPFINTEREXPR < LPFEXPR .

  op FUNCINTER(_, _, _) :
     LPFEXPR VARID LPFEXPR -> LPFINTEREXPR .

  *** Used later.
  sort EXISTSPAIR .
endfm
```

PredInter is defined just like FuncInter.

```
fmod LPFInterExprCont is
  protecting LPFBool .
  protecting LPFInt .
  including LPFExpr .
  including LPFInterExpr .
  protecting LIST{EXISTSPAIR} .

  sort EXISTSINTER .

  subsort EXISTSINTER < LPFINTEREXPR .

  op EXISTSPAIR(_, _) : LPFINT LPFEXPR -> EXISTSPAIR .
  op EXISTSINTER(_, _) : VARID List{EXISTSPAIR} -> EXISTSINTER .
endfm
```

The small-step SOS semantic relation is defined as:

```
mod LPFSmallStepSemantics is
  protecting LPFInt .
  protecting LPFBool .
  protecting LPFExpr .
  protecting LPFDefinitions .
  protecting LPFDefMap .
  protecting LPFSigma .
  protecting LPFInterExpr .
  protecting LPFInterExprCont .
  sort SMALLSTEPRELATION .
  sort SIDE .

  subsort VALUE < SMALLSTEPRELATION .
  subsort LPFEXPR < SMALLSTEPRELATION .

  op (_, _, _) : LPFEXPR
             Map{VALUEIDS, VALUE}
             Map{DEFINITIONIDS, DEFINITIONS} -> SIDE .

  op _ -se> : SIDE -> [SMALLSTEPRELATION] .
  ...
endm
```

The application of the small-step semantic relation `-se>` performs the one single rewrite. The reflexive, transitive closure is also needed which is defined as:

```
  op _ -E> : SIDE -> [SMALLSTEPRELATION] .

  crl [Base_Case] : (v1, sigma, defs) -E> => v2
      if (v1, sigma, defs) -se> => v2 /\ v1 == v2 .
  crl [Step_Case] : (a, sigma, defs) -E> => b
      if (a, sigma, defs) -se> => a' /\
         (a', sigma, defs) -E> => b .
```

where prior to this necessary variable placeholders are defined:

```
  var a : LPFEXPR .
  var a' : LPFEXPR .
  var b : LPFEXPR .
  var b' : LPFEXPR .

  var p : LPFEXPR .
  var p' : LPFEXPR .
  var q : LPFEXPR .
  var q' : LPFEXPR .

  var v1 : VALUE .
  var v2 : VALUE .
```

```
var pId : PROPID .
var vId : VARID .
var fId : FUNID .
var PId : PREDID .

var i1 : LPFINT .
var i2 : LPFINT .
var i3 : LPFINT .

var op : ARITHOP .

var sigma : Map{VALUEIDS, VALUE} .
var defs : Map{DEFINITIONIDS, DEFINITIONS} .
```

The small-step SOS rules can then be defined as:

```
rl [Value_E] : (v1, sigma, defs) -se> => v1 .

crl [Prop_E] : (pId, sigma, defs) -se> => sigma[pId]
     if $hasMapping(sigma, pId) .
rl [Var_E] : (vId, sigma, defs) -se> => sigma[vId] .

rl [Arith_E1] : (ARITH(v1, PLUS, v2), sigma, defs) -se> =>
                 (v1 + v2) .

rl [Arith_E2] : (ARITH(v1, MINUS, v2), sigma, defs) -se> =>
                 (v1 - v2) .

rl [Arith_E3] : (ARITH(v1, MULT, v2), sigma, defs) -se> =>
                 (v1 * v2) .

crl [Arith_E5] : (ARITH(v1, DIV, v2), sigma, defs) -se> =>
                 (v1 quo v2)
     if v2 =/= 0 .

crl [Arith_A] : (ARITH(a, op, b), sigma, defs) -se> =>
                ARITH(a', op, b')
     if (a, sigma, defs) -se> => a' /\
        (b, sigma, defs) -se> => b' .
```

```
rl [Equality_E] : (EQUALITY(v1, v2), sigma, defs)
        -se> => (v1 eq v2) .

crl [Equality_A] : (EQUALITY(a, b), sigma, defs)
        -se> => EQUALITY(a', b')
     if (a, sigma, defs) -se> => a' /\
        (b, sigma, defs) -se> => b' .

rl [Cond_E1] : (COND(LPFtrue, a, b), sigma, defs) -se> => a .

rl [Cond_E2] : (COND(LPFfalse, a, b), sigma, defs) -se> => b .

crl [Cond_A] : (COND(p, a, b), sigma, defs)
        -se> => COND(p', a, b)
     if (p, sigma, defs) -se> => p' .

rl [Not_E1] : (NOT(LPFtrue), sigma, defs) -se> => LPFfalse .

rl [Not_E2] : (NOT(LPFfalse), sigma, defs) -se> => LPFtrue .

crl [Not_A] : (NOT(p), sigma, defs) -se> => NOT(p')
     if (p, sigma, defs) -se> => p' .

rl [delta_E1] : (DELTA(LPFtrue), sigma, defs)
                -se> => LPFtrue .

rl [delta_E2] : (DELTA(LPFfalse), sigma, defs)
                -se> => LPFtrue .

crl [delta_A] : (DELTA(p), sigma, defs) -se> => DELTA(p')
     if (p, sigma, defs) -se> => p' .

rl [Or_E1] : (OR(LPFtrue, q), sigma, defs)
             -se> => LPFtrue .

rl [Or_E2] : (OR(p, LPFtrue), sigma, defs)
             -se> => LPFtrue .

rl [Or_E3] : (OR(LPFfalse, LPFfalse), sigma, defs)
```

```
              -se> => LPFfalse .

  crl [Or_A] : (OR(p, q), sigma, defs) -se> => OR(p', q')
      if (p, sigma, defs) -se> => p' /\
         (q, sigma, defs) -se> => q' .

  rl [FuncCall_E] : (FUNCCALL(fId, v1), sigma, defs)
         -se> =>
     FUNCINTER(getExpression(defs[fId]),
              getParams(defs[fId]), v1) .

  crl [FuncCall_A] : (FUNCCALL(fId, a), sigma, defs)
         -se> => FUNCCALL(fId, a')
      if (a, sigma, defs) -se> => a' .

  rl [FuncInter_E] : (FUNCINTER(v1, vId, v2), sigma, defs)
         -se> => v1 .

  crl [FuncInter_A] : (FUNCINTER(a, vId, v1), sigma, defs)
         -se> => FUNCINTER(a', vId, v1)
      if (a, insert(vId, v1, sigma), defs) -se> => a' .
  ...
```

The predicate call and the predicate inter rules follow in virtually the same way as the function call and the function inter rules. Existential quantification is considered below.

Notice that the `Or_A` rule is different to what was presented earlier in this chapter, where:

```
  crl [Or_L] : (OR(p, q), sigma, defs) -se> => OR(p', q)
      if (p, sigma, defs) -se> => p' .

  crl [Or_R] : (OR(p, q), sigma, defs) -se> => OR(p, q')
      if (q, sigma, defs) -se> => q' .
```

should have been expected. However, the rewrite (SOS) rules are nondeterministic. Some strategy to applying the rewrite rules is needed in a mechanisation to ensure that both operands get a chance to be evaluated. In LPF the truth tables are to be viewed as a parallel lazy evaluation of the operands. The use of the `Or_A` semantic rule ensures that both the `p` and the `q` operands get a chance to be evaluated. The p operand undergoes one rewrite step, and then the q operand undergoes one rewrite step during the application of the Or_A rewriting rule. This combining of the Or_L and Or_R rewriting rules into an Or_A rewriting rule provides the necessary control over the rewriting strategy in the face of the non-deterministic rule selection choice, that still allows for true to be returned even in the presence of a "gap" in either of the other operands, and in a way that is still faithful to the small-step SOS rules presented in Section 3.3.

The existential quantifier rewrite rules follow, where as discussed above a subset of the integer values is used:

```
op ExistsTrue(_) : EXISTSINTER -> Bool .
eq ExistsTrue(EXISTSINTER(vId, nil)) = false .
eq ExistsTrue(EXISTSINTER(vId, EXISTSPAIR(i1, p) pairs)) =
    if (p == LPFtrue) then true
    else ExistsTrue(EXISTSINTER(vId, pairs)) fi .

op ExistsFalse(_) : EXISTSINTER -> Bool .
eq ExistsFalse(EXISTSINTER(vId, nil)) = true .
eq ExistsFalse(EXISTSINTER(vId, EXISTSPAIR(i1, p) pairs)) =
    if (p =/= LPFfalse) then false
    else ExistsFalse(EXISTSINTER(vId, pairs)) fi .

rl [EXISTS-E] : (EXISTS(vId, p), sigma, defs) -se> =>
   EXISTSINTER(vId,
    (EXISTSPAIR(-1, p) EXISTSPAIR(0, p) EXISTSPAIR(1, p))) .

crl [EXISTS-T] : (inter, sigma, defs) -se> => LPFtrue
    if ExistsTrue(inter) .

crl [EXISTS-F] : (inter, sigma, defs) -se> => LPFfalse
    if ExistsFalse(inter) .

crl [EXISTSINTER-A] : (EXISTSINTER(vId,
  EXISTSPAIR(i1, a) EXISTSPAIR(i2, b) EXISTSPAIR(i3, p)),
            sigma, defs) -se> =>
    (EXISTSINTER(vId,
  EXISTSPAIR(i1, a') EXISTSPAIR(i2, b') EXISTSPAIR(i3, p')))
     if (a, insert(vId, i1, sigma), defs) -se> => a' /\
        (b, insert(vId, i2, sigma), defs) -se> => b' /\
          (p, insert(vId, i3, sigma), defs) -se> => p' .
```

as for the big-step semantics, only three quantified variables are used, but this provides an adequate basis for illustration. The use of a list will improve this mechanisation further. Like the small-step disjunction rules overcome the problem mentioned earlier with the big-step disjunction rules with undefinedness, the small-step existential quantifier rules overcome the problem with the big-step existential quantifier rules.

When using the small-step relation directly to evaluate the expression *zero*(0) the following results:

```
rew (FUNCCALL(F("zero"), 0), sigma1, defMap1) -se> .
LPFINTEREXPR: FUNCINTER(
                      COND(
                           EQUALITY(V("i"),0),
                              0,
                              FUNCCALL(F("zero"),
                                 ARITH(V("i"),MINUS,1))
                           ),
                      V("i"),
                      0)
```

but using the reflexive, transitive closure results in:

```
rew (FUNCCALL(F("zero"), 0), sigma1, defMap1) -E> .
result Zero: 0
```

also:

```
rew (OR(
        EQUALITY(FUNCCALL(F("zero"), 1), 0),
        EQUALITY(FUNCCALL(F("zero"), -1), 0)),
            sigma1, defMap1) -E> .
result LPFBOOL: LPFtrue
```

and:

```
rew (OR(
        EQUALITY(FUNCCALL(F("zero"), -1), 0),
        EQUALITY(FUNCCALL(F("zero"), 1), 0)),
            sigma1, defMap1) -E> .
result LPFBOOL: LPFtrue
```

Comparisons

In the big-step SOS definition the evaluation could get stuck in evaluating an undefined expression. This is overcome in the small-step SOS definition, due such a definition allowing for the interleaving of expressions in different branches. However, this is at a cost. The small-step SOS definition requires a much greater number of rewrites to take place than the big-step SOS definition requires on the same expression. This is illustrated by comparing the number of rewrites taken by each definition in Maude on the expressions presented in Figure 5.1. The results are presented in Table 5.1. The expressions from Figure 5.1 are evaluated with respect to `sigma1` and to `defMap1` which were introduced earlier in this section.

The SOS definitions were designed to illustrate the process of evaluating expressions in LPF, with efficiency not playing a key role, but rather expressing how expressions are to be evaluated.

Note that only the default rewriting strategy in Maude is being used. Thus when a stack overflow occurs in the results in Table 5.1 it is because the `Or_E1` rewriting rule has been selected, and the first operand contains the term that fails to denote and the weak equality relational operator. In the big-step SOS definition once the evaluation of an operand starts, control is in effect stuck in evaluating that operand. This is the issue that the small-step SOS definition overcomes. If the order of the `Or_E1` semantic rule and the `Or_E2` semantic rule are swapped around in the definition file then some of the results will be alternated.

Notice that for expression number 5, both definitions return `LPFtrue` despite the fact that the first operand to the disjunction operator is undefined. Division is defined by a conditional equation, and no further arithmetic rewrite rule can be applied on this first undefined operand. This differs from the results obtained for expression number 10, since a function needs evaluating, and thus further rewriting can take place on the undefined operand.

The most alarming differences between the number of rewrites being needed between the big-step SOS rules and the small-step SOS rules occur when functions are being evaluated. This is partly due to the use of `FuncInter` which stores necessary information which must be retrieved each time that a rewrite of the function definition being evaluated occurs. Such a technique is necessary to ensure that the intended result for LPF is returned as interleaving steps in different expression branches is necessary to express the parallel evaluation nature of LPF.

The denotational semantic definitions $\mathcal{E}$ could also be used as a basis for performing such a mechanisation of LPF in a term rewriting system like Maude.

```
 1. LPFtrue

 2. P("p")

 3. ARITH(1, PLUS, 1)

 4. OR(LPFfalse, LPFtrue)

 5. OR(EQUALITY(ARITH(1, DIV, 0), 1),
       EQUALITY(ARITH(1, DIV, 1), 1))

 6. COND(EQUALITY(ARITH(1, DIV, 1), 1), 1, 2)

 7. FUNCCALL(F("zero"), 0)

 8. FUNCCALL(F("zero"), 1)

 9. OR(EQUALITY(FUNCCALL(F("zero"), 1), 0),
       EQUALITY(FUNCCALL(F("zero"), -1), 0))

10. OR(EQUALITY(FUNCCALL(F("zero"), -1), 0),
       EQUALITY(FUNCCALL(F("zero"), 1), 0))

11. EXISTS(V("i"),
       OR(EQUALITY(FUNCCALL(F("zero"), 1), 0),
         EQUALITY(FUNCCALL(F("zero"), -1), 0)))

12. EXISTS(V("i"),
       OR(EQUALITY(FUNCCALL(F("zero"), -1), 0),
         EQUALITY(FUNCCALL(F("zero"), 1), 0)))

13. EXISTS(V("i"),
       OR(
         EQUALITY(FUNCCALL(F("zero"), V("i")), 0),
         EQUALITY(FUNCCALL(F("zero"),
                      ARITH(0, MINUS, V("i"))), 0)))

14. EXISTS(V("i"),
       OR(
         EQUALITY(FUNCCALL(F("zero"),
                      ARITH(0, MINUS, V("i"))), 0),
         EQUALITY(FUNCCALL(F("zero"), V("i")), 0)))
```

Figure 5.1: The expressions used for the SOS rewrite rule comparison

	Big-Step SOS		Small-Step SOS	
	Result	Number of Rewrites	Result	Number of Rewrites
1	LPFtrue	7	LPFtrue	9
2	LPFtrue	11	LPFtrue	15
3	2	10	2	12
4	LPFtrue	9	LPFtrue	11
5	LPFtrue	20	LPFtrue	36
6	1	18	1	24
7	0	27	0	46
8	0	70	0	127
9	LPFtrue	76	LPFtrue	341
10	Stack Overflow	n/a	LPFtrue	341
11	LPFtrue	80	LPFtrue	1306
12	Stack Overflow	n/a	LPFtrue	1306
13	Stack Overflow	n/a	LPFtrue	670
14	LPFtrue	86	LPFtrue	670

Table 5.1: A number of rewrite comparisons between the big-step Structural Operational Semantic and the small-step Structural Operational Semantic definitions

A lot of the code presented here for the mechanisation of the SOS definitions in Maude could be re-used in such a mechanisation.

5.2 Isabelle Mechanisation

In [AGM92, §4] the author suggests that instead of attempting to write your own theorem prover, it may be a better idea to try to capitalise upon an existing tool and build an extension on top of that. This advice is followed here.

The focus in this section is on a mechanisation of the LPF big-step SOS rules into Isabelle/HOL [NWP02]. Isabelle is an interactive generic theorem prover, which provides a meta-logic (Pure) which allows the formalisation of object logics. Different object logics have been predefined for Isabelle, one of which is Higher-Order Logic (HOL). This mechanisation of the LPF big-step SOS definition into Isabelle/HOL can be used to prove assertions in a way that is faithful to LPF. This preliminary work is to show how some interactive proof support for LPF can be provided/can be derived from the LPF big-step SOS definition.

For evaluating expressions (for instance, in a term-rewriting system), the small-step SOS semantics are necessary to precisely capture the semantics of LPF. For evaluating expressions to capture the parallel evaluation nature of LPF the interleaving of steps in different expression branches is necessary and thus for evaluating expressions the small-step SOS semantics are necessary as discussed earlier. In this proof setting the big-step SOS semantics are used. This is because of the nature of the proofs that will be conducted in showing that a given goal can be derived from given assumptions, by the user interactively guiding the proof by selecting the appropriate expression branch to follow where necessary.

The LPF big-step SOS definition were introduced in Section 3.3.1, and they are presented in full in Appendix A.

(Gudmund Grov helped me to get started with Isabelle/HOL, and collaboratively a first definition of LPF in Isabelle/HOL was written using my LPF big-step SOS definition. This definition has since been extensively modified and extended by myself which is presented in this section. All Isabelle/HOL proofs in this section are my own.)

Certain constructs provided in Isabelle cannot be reused here. As an example, `1 div 0` is `0` in Isabelle. In LPF it is undefined, that is, a "gap". It is not known which rules provided by Isabelle can be carried over to LPF, thus it is assumed that only the LPF rules provided below like `Value_E_I` and `Or_E1` are used in proofs. Additionally, the Isabelle simplifier `simp` can be used in restricted circumstances as will be illustrated in the example proofs that follow. As also mentioned in [AF97] (discussed in Section 2.4) the classical reasoning package appears not easy to use with LPF. The resolution proof procedure (not in the context of Isabelle) is considered for LPF in Chapter 6.

Since all functions are total in Isabelle/HOL the standard way of writing functions is not used here. The notion of functions being total is deeply embedded into numerous theorem proving systems. Functions instead are written using the *Func* construct, called using the *FuncCall* construct, and can be evaluated using the *FuncCall_E* semantic rule.

Only a subset of the big-step SOS definition are defined in Isabelle/HOL here for simplicity. The subset defined here though is sufficient to allow for proofs of the following two properties to be conducted, where i is an integer:

$$i \geq 0 \;\Rightarrow\; zero(i) = 0 \tag{5.1}$$

and:

$$zero(i) = 0 \vee zero(-i) = 0 \tag{5.2}$$

to be discharged are defined in Isabelle/HOL here. Adding the other expression definitions and semantic rules can be done by taking the same approach as to what is discussed below.

The definition file is defined as follows:

```
theory LPF
imports Main
begin
...
end
```

Main is a theory, which is the union of the predefined theories such as arithmetic.

Some expressions are then defined using the datatype construct:

```
datatype Expr =
   B "bool"
 | I "int"
 | P "string"
 | V "string"
 | Minus Expr Expr
 | Division Expr Expr
 | Equality Expr Expr
 | GreaterThan Expr Expr
 | GreaterThanEqual Expr Expr
 | Cond Expr Expr Expr
 | Not Expr
 | Or Expr Expr
 | Implies Expr Expr
 | FuncCall "string" Expr
```

This datatype defines the set of expressions that semantic rules are defined for. The first two lines `B` and `I` are the constant Boolean values and constant integer values respectively. The following two lines `P` and `V` represent the propositional variables and the integer variables respectively. The identifiers themselves are just strings. For instance, the Boolean value true is `B True`, and the propositional identifier `p` is `P ''p''`. An equality expression is to be written as `Equality a b`, where `a` and `b` are expressions (*Expr*), and so on.

It is assumed that any expression written will pass through the context conditions. The context conditions are not defined here in Isabelle, they are expressed formally in Section 3.2, and they are defined in Maude in Section 5.1. The big-step SOS rules that are introduced later ensure that if given `Equality a b`, that `a` must evaluate to `a'`, where `a'` is an integer, and that `b` must evaluate to `b'`, where `b'` is an integer, because both operands must be defined for equality. Without the context conditions one can write

`Or (B True) (I 1)` which since only the one operand must be true for the disjunction to be true, `Or (B True) (I 1)` can be proven to be true in the following. Such a formula though would be rejected immediately by the context conditions of Section 3.2. It is thus assumed that any expression that is to be proved would pass through the context conditions, and that a function body expression is an integer expression, and that a predicate body expression is a Boolean expression.

A function definition and a predicate definition are also defined as datatypes:

```
datatype Function = Func Expr
datatype Predicate = Pred Expr
```

A function definition is written as `Func e`, where `e` is the expression representing the body of the function.

The memory store Σ is defined as:

```
datatype PropStore = σProp "bool"
datatype VarStore = σVar "int"
datatype FuncStore = σFn Function
datatype PredStore = σPr Predicate
```

The four stores that make up a Σ are separated here. Notice that these are not maps. Each can only store a single item, for instance, a single propositional variable identifier, or a single function definition, and so on. If, for instance, more than the one function definition is to be used within a proof then these definitions will need extending, as at the minute only the one function definition can be present (the same applies to predicates and so on). This simplifies the following definitions, but still allows for the issues surrounding undefinedness to be adequately illustrated, and for proofs of Property 5.1 and Property 5.2 to be discharged.

The semantic rules are defined making use of the following notation:

```
datatype Data_B = D_B
  "(Expr ×
    PropStore × VarStore × FuncStore × PredStore ×
    bool)"

datatype Data_I = D_I
  "(Expr ×
    PropStore × VarStore × FuncStore × PredStore ×
    int)"
```

The datatype `Data_B` and the datatype `Data_I` enclose the necessary information, the expression to be evaluated, the four memory stores, and the expected result value, either a Boolean value or an integer value.

To illustrate how the semantic rules are defined in Isabelle, consider the following definition of the *Value_E* semantic rule but only for an integer value (`Value_E_I`), (the lemma for a Boolean value –`Value_E_B`– is virtually the same, but would use `eBS_B` and `D_B` instead); the use of `eBS_B` (`e` Big-Step Boolean) and `eBS_I` (`e` Big-Step integer) will be discussed below:

```
Value_E_I :
  "eBS_I (D_I (I v, σProp pro, σVar var, σFn fn, σPr pr, v))"
```

The `Prop_E` semantic rule is defined as:

```
Prop_E :
  "eBS_B (D_B (P iden, σProp pro, σVar var,
                        σFn fn, σPr pr, pro))"
```

The use of `P` ensures that a propositional identifier is being used. The result `pro` is the given Boolean value stored at σ`Prop`. The *Var_E* rule is virtually the same, but using `V iden` and `var` as the first and last arguments respectively, as well as using `eBS_I` and `D_I` since it is an integer expression.

The division semantic rule is defined as:

```
Division_E :
  "⟦eBS_I (D_I (a, σProp pro, σVar var, σFn fn, σPr pr, a')) ;
     eBS_I (D_I (b, σProp pro, σVar var, σFn fn, σPr pr, b')) ;
     b' ≠ 0 ;
     v = (a' div b')⟧
  ⟹ eBS_I (D_I (Division a b, σProp pro, σVar var,
              σFn fn, σPr pr, v))"
```

The assumptions are enclosed in ⟦⟧, which are separated by a `;`. The symbol $\Longrightarrow$ is meta-implication and is used to separate the assumptions from the conclusion. Notice that if `a` and `b` have been shown to be defined, that is, that they have evaluated to constant integer values `a'` and `b'` respectively, then the Isabelle division operator is used to compute the result `v`. The guard on `b'` (an integer value) not being equal to `0` is of the upmost importance here, because `a' div 0` is `0` in Isabelle, while it LPF this is reagrded as undefined. The subtraction rule `Minus_E` is defined virtually the same, but obviously with no such `b'` not being equal to `0` guard.

The equality elimination semantic rule `Equality_E` is defined in virtually the same way as:

```
Equality_E :
  "⟦eBS_I (D_I (a, σProp pro, σVar var, σFn fn, σPr pr, a')) ;
     eBS_I (D_I (b, σProp pro, σVar var, σFn fn, σPr pr, b')) ;
     v = (a' = b')⟧
  ⟹ eBS_B (D_B (Equality a b, σProp pro, σVar var,
             σFn fn, σPr pr, v))"
```

The greater than rule (`GreaterThan_E`) and the greater than equal rule (`GreaterThanEqual_E`) are both defined in virtually the same way as the `Equality_E` rule.

The disjunction lemmas are defined as:

```
Or_E1 :
  "⟦eBS_B (D_B (p, σProp pro, σVar var,
            σFn fn, σPr pr, True))⟧
  ⟹ eBS_B (D_B (Or p q, σProp pro, σVar var,
            σFn fn, σPr pr, True))"
Or_E2 :
  "⟦eBS_B (D_B (q, σProp pro, σVar var,
            σFn fn, σPr pr, True))⟧
  ⟹ eBS_B (D_B (Or p q, σProp pro, σVar var,
            σFn fn, σPr pr, True))"
Or_E3 :
  "⟦eBS_B (D_B (p, σProp pro, σVar var,
            σFn fn, σPr pr, False)) ;
     eBS_B (D_B (q, σProp pro, σVar var,
            σFn fn, σPr pr, False))⟧
  ⟹ eBS_B (D_B (Or p q, σProp pro, σVar var,
            σFn fn, σPr pr, False))"
```

The negation rules (`Not_E1` and `Not_E2`) and the conditional expression rules (`Cond_E1` and `Cond_E2`) are both defined in virtually the same way.

Implication is defined in terms of a negation and a disjunction:

```
Impl_E :
  "⟦eBS_B (D_B (Or (Not p) q, σProp pro, σVar var,
            σFn fn, σPr pr, True))⟧
  ⟹ eBS_B (D_B (Implies p q, σProp pro, σVar var,
            σFn fn, σPr pr, True))"
```

The function call elimination rule is defined as:

```
FuncCall_E :
  "⟦eBS_I (D_I (arg, σProp pro, σVar var,
             σFn (Func res), σPr pr, arg')) ;
     eBS_I (D_I (res, σProp pro, σVar arg',
             σFn (Func res), σPr pr, res'))⟧
  ⟹ eBS_I (D_I (FuncCall iden arg, σProp pro, σVar var,
             σFn (Func res), σPr pr, res'))"
```

An inductive predicate `eBS_B` is defined as:

```
inductive eBS_B :: "Data_B ⇒ bool"
```

and an inductive predicate `eBS_I` for integer expressions is defined as:

```
inductive eBS_I :: "Data_I ⇒ bool"
```

The `eBS_B` definition and the `eBS_I` definition are then completed using the rules that have been presented above, like:

```
where
  ϕ₁
| ...
| ϕₙ
```

where ϕ_i is one of the rules given above, such as `Value_E_I` and `FuncCall_E`. The Boolean expression rules are added to the definition of `eBS_B`, and the integer expression rules are added to the definition of `eBS_I`, as stated in the alternative *Expr* definition in Section 3.2.

Proofs can now be conducted that make use of these big-step SOS rules. A proof will here take the following form:

```
lemma name : "goal"
apply rule₁
...
apply ruleₙ
done
```

where rules are applied until the goal has been proved.

As an introductory example consider a proof of the LPF inference rule:

$$\boxed{\neg\text{-}\neg\text{-}I}\,\frac{p}{\neg\neg\, p}$$

This proof applies the Not_E1 rule, followed by the Not_E2 rule, which leaves a goal to be proved of p. This follows immediately by the given assumption:

```
lemma Not_Not_I :
  "⟦eBS_B (D_B (p, σProp pro, σVar var,
           σFn fn, σPr pr, True))⟧
  ⟹ eBS_B (D_B (Not (Not p), σProp pro, σVar var,
           σFn fn, σPr pr, True))"
apply (rule Not_E2)
apply (rule Not_E1)
apply assumption
done
```

As another example the proof of Division (I 1) (I 1) being 1 follows.

```
lemma "eBS_I (D_I (Division (I 1) (I 1),
          σProp pro, σVar var, σFn fn, σPr pr, 1))"
apply (rule Division_E)
apply (rule Value_E_I)
apply (rule Value_E_I)
apply simp
apply simp
done
```

The application of the Division_E rule forces four subgoals to be proved. The first two relate to showing that the two operands are defined and denote integer values, which follows immediately by use of the Value_E_I rule, since both operands are the integer value I 1. Since the two operands have been shown to be defined the Isabelle simplifier simp is used to calculate the value of the division since the operands have been shown to be defined. The simplifier also discharges the b not being equal to 0 subgoal.

The following is an example of a proof that cannot be discharged using the LPF rules provided. The integer variable V ''i'' will be assigned the integer value 1 from the integer value store σVar. After the Isabelle simplifier on the penultimate line is executed the value False is inferred, since the second defined operand to the Division operator is 0.

```
lemma "eBS_B (D_B (Equality (Division (V ''i'') (I 0)) (I 0),
          σProp pro, σVar 1, σFn fn, σPr pr, True))"
apply (rule Equality_E)
apply (rule Division_E)
apply (rule Var_E)
apply (rule Value_E_I)
apply simp
```

The above proof attempt has not been able to be completed.

In the following `zeroFunction` refers to the following:

```
σFn (Func
      (Cond (Equality (V ''i'') (I 0))
              (I 0)
              (FuncCall ''zero'' (Minus (V ''i'') (I 1)))))
```

which is the definition of the *zero* function introduced earlier.

The following proofs shows that *zero*(0) evaluates to 0:

```
lemma "eBS_I (D_I (FuncCall ''zero'' (I 0),
          σProp pro, σVar var, zeroFunction, σPr pr, 0))"
apply (rule FuncCall_E)
apply (rule Value_E_I)
apply (rule Cond_E1)
apply (rule Equality_E)
apply (rule Var_E)
apply (rule Value_E_I)
apply simp
apply (rule Value_E_I)
done
```

So, for example after applying the `FuncCall_E` semantic rule in the above proof you are left with two subgoals to prove:

```
1. eBS_I
     (D_I (I (0::int), σProp pro, σVar var,
           σFn (Func
               (Cond (Equality (V ''i'') (I (0::int)))
                     (I (0::int))
                     (FuncCall ''zero''
                       (Minus (V ''i'') (I (1::int)))))),
           σPr pr, ?arg'))
2. eBS_I
     (D_I (Cond (Equality (V ''i'') (I (0::int))) (I (0::int))
            (FuncCall ''zero'' (Minus (V ''i'') (I (1::int)))),
           σProp pro, σVar ?arg',
           σFn (Func
                (Cond (Equality (V ''i'') (I (0::int)))
                      (I (0::int))
                      (FuncCall ''zero''
                        (Minus (V ''i'') (I (1::int)))))),
           σPr pr, 0::int))
variables:
  pr :: Predicate
  ?arg', var :: int
  pro :: bool
```

In the above proof the subgoals were proved in the order listed. The first relates to showing that the argument is defined. The second relates to showing that this application of the *zero* function evaluates to 0.

The following proof shows that that *zero*(1) also evaluates to 0, where this proof requires an extra application of the `FuncCall_E` rule:

```
lemma "eBS_I (D_I (FuncCall ''zero'' (I 1),
          σProp pro, σVar var, zeroFunction, σPr pr, 0))"
apply (rule FuncCall_E)
apply (rule Value_E_I)
apply (rule Cond_E2)
apply (rule Equality_E)
apply (rule Var_E)
apply (rule Value_E_I)
apply simp
apply (rule FuncCall_E)
apply (rule Minus_E)
apply (rule Var_E)
apply (rule Value_E_I)
apply simp
apply (rule Cond_E1)
apply (rule Equality_E)
apply (rule Var_E)
apply (rule Value_E_I)
apply simp
apply (rule Value_E_I)
done
```

Now Property 5.1 and Property 5.2 will be proved. In order to prove Property 5.1 two inference rules are used:

$$\boxed{zero\_b}\frac{}{zero(0) = 0}$$

$$\boxed{zero\_i}\frac{\begin{array}{c} i{:}\,\mathbb{Z}; \\ i \neq 0; \\ zero(i-1) = k \end{array}}{zero(i) = k}$$

which are assumed to be true, refer to Section 2.3 where these two inference rules were first used in this thesis.

The following two lemmas represent the above two properties. The `sorry` command is used to abandon the proof attempt since these are lemmas that are being "assumed" true here for this work and are not being proved. The `sorry` command allows the user to continue work in the proof file.

```
lemma zero_b :
  "eBS_B (D_B (Equality (FuncCall ''zero'' (I 0)) (I 0),
          σProp pro, σVar var, zeroFunction, σPr pr, True))"
sorry

lemma zero_i :
  "⟦eBS_B (D_B (Not (Equality (I i) (I 0)),
            σProp pro, σVar var, zeroFunction, σPr pr, True)) ;
     eBS_B (D_B (Equality
                     (FuncCall ''zero'' (Minus (I i) (I 1))) (I k),
            σProp pro, σVar var, zeroFunction, σPr pr, True))⟧
  ⟹ eBS_B (D_B (Equality (FuncCall ''zero'' (I i)) (I k),
            σProp pro, σVar var, zeroFunction, σPr pr, True))"
sorry
```

The proof will follow by using natural number induction. Since the only numeric datatype that is being used is integer values the induction rule needs an assumption that $i \geq 0$:

$$\boxed{zero\_Natural\_Number\_Induction}\frac{\begin{array}{c} i \geq 0; \\ zero(0) = 0; \\ zero(i-1) = k \vdash i > 0 \ \Rightarrow \ zero(i) = k \end{array}}{zero(i) = k}$$

```
lemma zero_Natural_Number_Induction :
  "⟦eBS_B (D_B (GreaterThanEqual (I i) (I 0),
           σProp pro, σVar var, zeroFunction, σPr pr, True)) ;
     eBS_B (D_B (Equality (FuncCall ''zero'' (I 0)) (I 0),
           σProp pro, σVar var, zeroFunction, σPr pr, True)) ;
     (eBS_B (D_B (Equality
                 (FuncCall ''zero'' (Minus (I i) (I 1))) (I k),
            σProp pro, σVar var, zeroFunction, σPr pr, True))
     ⟹ eBS_B (D_B (Implies (GreaterThan (I i) (I 0))
                     (Equality (FuncCall ''zero'' (I i)) (I k)),
           σProp pro, σVar var, zeroFunction, σPr pr, True)))⟧
  ⟹ eBS_B (D_B (Equality (FuncCall ''zero'' (I i)) (I k),
            σProp pro, σVar var, zeroFunction, σPr pr, True))"
sorry
```

The proof of Property 5.1 follows where case distinctions are made using `case_tac`. After the first case distinction the case that $i < 0$ is dealt with, which follows since **false** $\Rightarrow$ p is true. This case is concluded with the first use of `simp`. After this the case that $i \geq 0$ is dealt with, which follows by induction. The subgoal $i \geq 0$ and the base case subgoal are then discharged, the former follows trivially and the latter follows by applying `zero_b`. To conclude the step case another case distinction is made on $i > 0$. To discharge the $i > 0$ case an application of `zero_i` is made, and the subgoal then follows by the assumption. The $i \leq 0$ case follows trivially.

```
lemma zero_Implication_Example :
  "eBS_B (D_B (Implies (GreaterThanEqual (I i) (I 0))
                (Equality (FuncCall ''zero'' (I i)) (I 0)),
          σProp pro, σVar var, zeroFunction, σPr pr, True))"
apply (case_tac "i < (0::int)")
apply (rule Impl_E)
apply (rule Or_E1)
apply (rule Not_E2)
apply (rule GreaterThanEqual_E)
apply (rule Value_E_I)
apply (rule Value_E_I)
apply simp
apply (rule Impl_E)
apply (rule Or_E2)
apply (rule zero_Natural_Number_Induction)
apply (rule GreaterThanEqual_E)
apply (rule Value_E_I)
apply (rule Value_E_I)
apply simp
apply (rule zero_b)
apply (case_tac "i > (0::int)")
apply (rule Impl_E)
apply (rule Or_E2)
apply (rule zero_i)
apply (rule Not_E2)
apply (rule Equality_E)
apply (rule Value_E_I)
apply (rule Value_E_I)
apply simp
apply assumption
apply (rule Impl_E)
apply (rule Or_E1)
apply (rule Not_E2)
apply (rule GreaterThan_E)
apply (rule Value_E_I)
apply (rule Value_E_I)
apply simp
done
```

In order to discharge the proof of Property 5.2 the $\Rightarrow$ $\_E\_L$ inference rule is made use of:

$$\boxed{\Rightarrow\ \_E\_L}\ \frac{p\ \Rightarrow\ q; p}{q}$$

```
lemma Impl_E_L :
  "⟦eBS_B (D_B (Implies p q, σProp pro, σVar var,
            σFn fn, σPr pr, True)) ;
     eBS_B (D_B (p, σProp pro, σVar var,
            σFn fn, σPr pr, True))⟧
   ⟹ eBS_B (D_B (q, σProp pro, σVar var,
            σFn fn, σPr pr, True))"
sorry
```

The proof of Property 5.2 then follows by making a case distinction on $i < 0$, and through applying the `zero_Implication_Example` rule.

```
lemma zero_Disjunction_Example :
  "eBS_B (D_B (Or (Equality (FuncCall ''zero'' (I i)) (I 0))
                (Equality (FuncCall ''zero'' ((I (-i)))) (I 0)),
          σProp pro, σVar var, zeroFunction, σPr pr, True))"
apply (case_tac "i < (0::int)")
apply (rule Or_E2)
apply (rule Impl_E_L)
apply (rule zero_Implication_Example)
apply (rule GreaterThanEqual_E)
apply (rule Value_E_I)
apply (rule Value_E_I)
apply simp
apply (rule Or_E1)
apply (rule Impl_E_L)
apply (rule zero_Implication_Example)
apply (rule GreaterThanEqual_E)
apply (rule Value_E_I)
apply (rule Value_E_I)
apply simp
done
```

This work provided some interactive proof support for LPF, but it is by no means a complete definition, but it does provide the foundation to facilitate the further development of interactive proof support for LPF. Furthermore, this preliminary work on mechanised proof support for LPF in Isabelle has shown that Isabelle is a useful tool for providing proof support for non-classical logics. Unfortunately due to the embedding used, large logical formulae need to be written. Also in this preliminary work on mechanised proof support for LPF there is not much automation available; the resolution proof procedure is investigated in Chapter 6.

It would be useful to extend this *partial* mechanisation further to include support for VDM-SL datatypes etc., as well as to address the simplifications that have been made in this mechanisation.

5.3 Conclusions

The Maude term-rewriting system has been used to provide a mechanisation of the big-step SOS definition for LPF, and of the small-step SOS definition for LPF. This has provided some assurance that the semantic definitions accurately formalise the semantics of LPF, as well as providing a case study on how to incorporate SOS definitions into Maude. This mechanisation has shown precisely why the small-step SOS semantic definition is required to accurately capture the semantics of LPF for evaluating expressions. Also a comparison between the small-step and the big-step SOS definition implementations has been made, comparing the number of rewrite rules executed in evaluating different expressions, showing the scale of how many more rewrites need to take place with the small-step definition.

The LPF big-step SOS definition has been used as the foundation of providing interactive proof support to allow for reasoning about logical formulae that can contain references to partial terms in LPF, using the Isabelle proof assistant. Key properties of logical formulae that contain references to partial functions have been discharged.

Chapter 6

Investigating Proof Procedures in LPF

Contents

6.1 Validity and Satisfiability 185

6.1.1 Two-Valued Classical Logic Recap 185

6.1.2 LPF 186

6.2 The Method of Truth Tables 187

6.3 Clausal Form 188

6.3.1 Two-Valued Classical Logic Recap 188

6.3.2 LPF 189

6.4 Unification 198

6.5 Resolution 199

6.5.1 Two-Valued Classical Logic Recap 199

6.5.2 LPF 201

6.6 Refutation Procedures 204

6.6.1 Two-Valued Classical Logic Recap 204

6.6.2 LPF 205

6.7 Conclusions 215

This chapter presents an investigation into the applicability of mechanised proof support for LPF, by focusing on the basic but fundamental two-valued classical logic proof procedure resolution and the associated technique proof by contradiction. An investigation of the issues that arise in applying these techniques to LPF, and into the extent of the modifications needed to be made to these techniques for LPF is presented. This provides key insights into providing mechanised proof support for LPF.

Recall that there has been a lack of direct proof support for LPF over the years. Investigating the impact of the fundamental basic techniques in LPF is thus the essential and obvious starting point for investigating proof support for LPF. These fundamental basic proof techniques are the foundation on which many advanced proof techniques, (see Section 7.2.4) are built. This work provides the essential foundation to facilitate research into the modification of advanced proof techniques for LPF, and for providing tool support.

The $\mathcal{E}$ semantic function definition presented in Figure 3.8 is used in this chapter to aid in presenting concepts, issues that arise, as well as being the underlying basis on which proofs are conducted. A simplified version of the $\mathcal{E}$ semantic function definition is used in this chapter as the expressions $mk\_Arith(a, op, b)$ and $mk\_Equality(a, b)$ are removed from consideration. This is because the techniques used in this chapter are syntactic, so for instance, the equality predicate is just an arbitrary predicate. The set of expressions $Expr^{\Delta}$ considered in this chapter does not include any *Arith* and *Equality* expressions, and is referred to as *Expr* from here on.

The notions of validity, satisfiability, and related definitions for LPF are introduced first in this chapter. This is followed by introducing the clausal form notation for LPF. The topics of the resolution rule of inference and refutation procedures (proving the validity of a formula by refuting its negation [BA01]) are then considered separately, before they are addressed combined (essentially doing a proof by contradiction). Resolution is a refutation procedure [BA01]. Below when the term resolution proof procedure is used refutation is not of consideration. When concerned with refutation, the term resolution refutation procedure will be used. The main contributions of this chapter is the adaption of the resolution refutation procedure for LPF, presented in Section 6.5 onwards. The work prior to that in this chapter is necessary to support those contributions.

(Some initial collaborative work with my supervisors on this work are published in [JLS12a][1] which has been extended in this chapter. Key definitions, methods, and results, such as the inclusion of unification constraints and discharging definedness obligations using resolution for instance, are mine. Illustrative proofs done in respect to $\mathcal{E}$ and Σ, and the resolution proofs are my own as well.)

6.1 Validity and Satisfiability

Key definitions from two-valued classical logic are re-stated before they are formally defined for LPF.

6.1.1 Two-Valued Classical Logic Recap

An interpretation is a map that assigns a meaning to the variables, as well as to the function and the predicate symbols that appear in a given formula. Given a formula e:

[1]A shorter version of this paper has been accepted for publication elsewhere. It has been peer-reviewed, but as of July 2013 it has not been published yet.

- e is satisfiable iff there exists an interpretation where e evaluates to true (such a satisfying interpretation is known as a model for e);
- e is unsatisfiable iff it is not satisfiable, that is, there exists no interpretation where e evaluates to true;
- e is valid ($\models e$) iff e evaluates to true in every interpretation (and is thus also satisfiable); and
- e is not valid ($\not\models e$) iff there exists an interpretation where e does not evaluate to true.

Two formulae e_1 and e_2 are logically equivalent iff e_1 and e_2 have the same truth value in every interpretation. Two formulae are equi-satisfiable when e_1 is satisfiable iff e_2 is satisfiable (they may not be logically equivalent or even share the same model).

Let $\Gamma = \{e_1, \ldots, e_n\}$ be a set of formulae, where the commas are to be interpreted as conjunctions. The set of formulae Γ is satisfiable iff there exists an interpretation where each e_i evaluates to true (such a satisfying interpretation is known as a model of Γ). The set of formulae Γ is unsatisfiable iff there exists no interpretation where Γ evaluates to true (that is, in every interpretation an e_i must not evaluate to true).

Furthermore, e is a logical consequence of Γ ($\Gamma \models e$), if e evaluates to true, in every interpretation where Γ evaluates to true. If $\Gamma = \{\}$ then logical consequence is the same as validity. The notation $\Gamma \not\models e$ is used when e is not a logical consequence of Γ.

6.1.2 LPF

The above concepts will now be defined for LPF through the use of the $\mathcal{E}$ (including the Σ definition) semantic function definition.

In terms of the $\mathcal{E}$ semantic function definition, an interpretation is a $\sigma \in \Sigma$. Given a formula e, e^{σ} represents $(\sigma, \mathbf{true}) \in \mathcal{E}(e)$ and $e^{\overline{\sigma}}$ represents $(\sigma, \mathbf{false}) \in \mathcal{E}(e)$. Additionally, $e^{\Sigma} = \{\sigma \mid \sigma\colon \Sigma \wedge e^{\sigma}\}$. The concepts of validity and satisfiability etc. are defined as:

$$\begin{gathered}
\mathit{satisfiable}(e) \text{ iff } \exists \sigma\colon \Sigma \cdot (\sigma, \mathbf{true}) \in \mathcal{E}(e) \text{ (that is, } e^{\Sigma} \neq \{\}) \\
\mathit{unsatisfiable}(e) \text{ iff } \neg\, \exists \sigma\colon \Sigma \cdot (\sigma, \mathbf{true}) \in \mathcal{E}(e) \text{ (that is, } e^{\Sigma} = \{\}) \\
\mathit{valid}(e) \text{ iff } \forall \sigma\colon \Sigma \cdot (\sigma, \mathbf{true}) \in \mathcal{E}(e) \text{ (that is, } e^{\Sigma} = \Sigma)
\end{gathered}$$

Notice that if unsatisfiable were defined as: $\forall \sigma : \Sigma \cdot (\sigma, \textbf{false}) \in \mathcal{E}(e)$, then the set of unsatisfiable expressions would be smaller since a formula e not evaluating to true in LPF, is not the same as it evaluating to false. In two-valued classical logic, the only possible outcomes are true and false but in LPF it is necessary to take a position on the "gaps", that is, $\sigma \notin \textbf{dom}\, \mathcal{E}(e)$.

In terms of the $\mathcal{E}$ semantics, e_1 and e_2 are logically equivalent iff for all $\sigma \in \Sigma$ it is the case that $(\sigma, v) \in \mathcal{E}(e_1)$ iff $(\sigma, v) \in \mathcal{E}(e_2)$. In the $\mathcal{E}$ semantics it thus follows that e_1 and e_2 are logically equivalent iff $\mathcal{E}(e_1) = \mathcal{E}(e_2)$.

Two formulae are equi-satisfiable when there exists a model for e_1 iff there is a model for e_2.

Furthermore, Γ^σ is taken to represent $e_1^\sigma \wedge \ldots \wedge e_n^\sigma$, and $\Gamma^\Sigma = \{\sigma \mid \sigma : \Sigma \wedge \Gamma^\sigma\}$.

Logical consequence can be defined using the more concise notation instead of the longer set definition as was presented in Section 4.2, (the set definitions though are semantically the same), as $\Gamma^\Sigma \subseteq e^\Sigma$. When Γ is empty $\models e$ is written and every $\sigma \in \Sigma$ must make e true.

6.2 The Method of Truth Tables

For propositional logic, validity etc. can be checked by using *truth tables.* To decide if two formulae are logically equivalent a truth table could be constructed for each formula, and the two formulae are logically equivalent if the two corresponding truth tables are identical.

A truth table is a two dimensional array that, for a formula e, has columns to represent each atom of e and one column to represent the results for e, with each row of the truth table corresponding to an interpretation (an assignment of values to the atoms) as well as the result of applying the interpretation to e. Since only a finite number of atoms can exist within a given formula it is possible to check all possibilities.

The performance of the truth table method is exponential because it requires checking 2^n interpretations, where n is the number of distinct atoms contained within the formula. Thus for n atoms there will be 2^n rows in the corresponding truth table.

The number of rows required in a truth table for LPF increases. In terms of the $\mathcal{E}$ semantics presented earlier, every assignment of values is a $\sigma \in \Sigma$ (σ's where a propositional variable under consideration is not included in the domain –as a maplet– must be considered to allow for undefined propositional variables to occur). For propositional LPF, the truth table method would require checking 3^n σ instances to check a formula for validity, that is, that there will be 3^n rows in the corresponding truth table. It might not be immediately obvious why it is necessary to check the result when propositional variables fail to denote but consider an example like $\Delta\neg\, p \vdash \Delta p$:

p	$\neg\, p$	$\Delta\neg\, p$	Δp
true	**false**	**true**	**true**
$\perp_{\mathbb{B}}$	$\perp_{\mathbb{B}}$	**false**	**false**
false	**true**	**true**	**true**

Of the 3^n rows in a truth table in LPF, 2^n rows correspond to those rows where all propositional variables denote (the same rows that would be present in a two-valued classical logic truth table), and $(3^n) - (2^n)$ rows correspond to a case where at least the one propositional variable does not denote. Thus, in LPF there are a significant number of "extra" cases to consider.

Thus, if there exists a large number of atoms in a formula then this method is inefficient. For predicate calculus, formulae can include the use of quantifiers, so the method of finite truth tables is not adequate, since an infinite number of distinct interpretations may need to be considered. It is not possible to exhaustively search an infinite number of cases, so truth tables are not adequate.

6.3 Clausal Form

6.3.1 Two-Valued Classical Logic Recap

Formulae are commonly reduced to a *normal form*, which allows for the form of formulae to be standardised. This may be done for instance, to allow proof procedures to be used. The normal form that is of interest here is *Clausal Form* [BA01], where a set-based representation is used. A formula in clausal form is represented as a set of clauses (an implicit conjunction of clauses), where each clause is an implicit disjunction of literals. Thus in clausal form a formula is represented as a set of sets of literals. Clausal form is frequently used in (automated) theorem proving systems, for instance by the *resolution* proof procedure which is considered later in this chapter.

In two-valued classical logic both propositional and predicate formulae can be converted into clausal form. The conversion of a propositional formula into clausal form proceeds by first converting the formula under question into *Conjunctive Normal Form* (*CNF*) [BA01, Har09, Bun10] so that the formula is a conjunction of disjunctions of literals, where a literal is an atom (a positive literal) or the negation of an atom (a negative literal), that is, a formula in CNF is represented as: $C_1 \wedge \ldots \wedge C_n$, where each C_i is of the form: $l_{i_1} \vee \ldots \vee l_{i_{m_j}}$. Any two-valued classical propositional formula can be converted into a logically equivalent formula in CNF.

A propositional formula in CNF can be represented in the logically equivalent clausal form set-based notation, that is, $\{C_1, \ldots, C_n\}$, where each C_i is a set of the form: $\{l_{i_1}, \ldots, l_{i_{m_j}}\}$.

The conversion of a predicate formula into clausal form first requires converting the formula into *Prenex Normal Form* (*PNF*) [BA01], where any quantifiers are to occur on the left to the rest of the formula (known as the matrix). Any predicate formula can be converted into a logically equivalent formula in PNF, where it will be of the form: $Q_1 x_1 \cdot \ldots \cdot Q_n x_n \cdot M$, where each Q_i is a universal or an existential quantifier, and M is the quantifier free matrix.

The conversion of a closed (no free variables) predicate formula into clausal form first needs converting into PNF and then it needs *Skolemising* [BA01, Har09]. Skolemising a formula removes any existential quantifiers replacing them with either a Skolem constant or a Skolem function (over any universally quantified variables that preceded the existential quantifier). A Skolemised formula is equi-satisfiable to the original corresponding closed PNF formula.

The CNF conversions are to be used on the matrix in the same way as for propositional logic. The CNF formula can then be represented in clausal form. The universal quantifiers can then be dropped since a clausal form formula is closed; that is, that the variables in each clause are implicitly universally quantified.

6.3.2 LPF

This section outlines how to convert LPF formulae into *clausal form.* Propositional logic is considered first, followed by predicate logic. The standard two-valued classical logic conversions needed to be able to convert a formula into clausal form carry over to LPF, but the non-monotone Δ operator requires additional conversions to be able to convert an LPF formula into clausal form. The use of Δ can lead to large resulting clausal form formulae.

Fortunately, the use of Δ can be limited; this is discussed in Section 6.6. Here the logically equivalent conversions for the Δ logical operator are presented.

These standard conversions presented in this section provide the foundation on which research into more advanced (optimised) conversion techniques can be conducted, that avoid such a rapid expansion of formulae.

Propositional Logic

The process of converting a propositional formula into CNF is extended from the two-valued classical case since any Δ that occurs needs pushing inwards so that any Δ in a formula that is in CNF/clausal form will only surround a literal. Thus in LPF, what is meant by a literal is extended to also include Δl and $\neg \Delta l$, where l is an literal in the standard sense. The process of converting a formula into CNF basically follows the standard approach [BA01], but supplemented with the application of Δ conversion rules:

- eliminate any propositional operators other than conjunction, disjunction and negation by applying the standard syntactic definitions, e.g. replace any $p \Rightarrow q$ with $\neg p \vee q$;
- push in the Δ operator, (see the discussion below);
- use *de Morgan's Laws* to force negations inwards, (see Lemma 10);
- eliminate all double negations, (see Lemma 11); and
- use the *distributive laws* to remove conjunctions from within disjunctions, (see Lemma 12).

The second step where a Δ operator is to be pushed inwards takes place by using the following equivalences:

- $\Delta(p \vee q)$ is logically equivalent to:

  $$\neg((\neg p \wedge \neg \Delta q) \vee (\neg \Delta p \wedge \neg q) \vee (\neg \Delta p \wedge \neg \Delta q))$$

  (i.e. the negation of the three cases that make $\Delta(p \vee q)$ denote false), which converted into CNF is:

  $$(p \vee \Delta q) \wedge (\Delta p \vee q) \wedge (\Delta p \vee \Delta q)$$

- $\Delta(p \wedge q)$ is logically equivalent to the CNF formula:

  $$(\neg p \vee \Delta q) \wedge (\Delta p \vee \neg q) \wedge (\Delta p \vee \Delta q)$$

For illustration of this tranformations consider the formula:

$$\neg \Delta(p \vee q)$$

the Δ should be pushed inwards first and then the negation can be pushed inwards. For instance, the formula $\neg \Delta(p \vee q)$ is first to be converted to:

$$\neg((p \vee \Delta q) \wedge (\Delta p \vee q) \wedge (\Delta p \vee \Delta q))$$

which is then converted into the CNF formula:

$$(\neg\, p \vee \neg\, \Delta p) \wedge (\neg\, q \vee \neg\, \Delta q) \wedge (\neg\, \Delta p \vee \neg\, \Delta q)$$

The formula:

$$\neg\, \Delta(p \wedge q)$$

converted into CNF becomes:

$$(p \vee \neg\, \Delta p) \wedge (q \vee \neg\, \Delta q) \wedge (\neg\, \Delta p \vee \neg\, \Delta q)$$

Furthermore, given:

$$\neg\, (\Delta(p \vee q \vee r))$$

then the corresponding logically equivalent CNF formula is:

$$(\neg\, p \vee \neg\, \Delta p) \wedge (\neg\, q \vee \neg\, \Delta q) \wedge (\neg\, r \vee \neg\, \Delta r) \wedge (\neg\, \Delta p \vee \neg\, \Delta q \vee \neg\, \Delta r)$$

It is also the case that $\Delta \neg\, l$ can be simplified to Δl, (see Lemma 13).

Transformations like these for Δ are needed in the well-definedness approach for the well-definedness operator $\mathcal{D}$ in [Meh08]. For the work in this chapter the introduction of Δ can be minimised/avoided, and the size of the transformations in places reduced; this is discussed later in this chapter when resolution and refutation are being considered. An aim of this work is to investigate the extent of the extra work needed for LPF when applying resolution and refutation, and key to this was investigating how to reduce the introduction of the expensive Δ operator in resolution proofs.

Illustrative proofs that show that these propositional conversions carry over to LPF follow. All such proofs presented in this chapter are done with respect to the $\mathcal{E}$ semantic function definition that was presented in Section 3.4, which formally defines the semantics of LPF.

Lemma 10. Any formula $\neg\, (p \vee q)$ is logically equivalent to $(\neg\, p) \wedge (\neg\, q)$.
Proof. By the definition of $\mathcal{E}$, $\mathcal{E}(\neg\, (p \vee q))$ expands to:
$\{(\sigma, \mathbf{true}) \mid (\sigma, \mathbf{false}) \in \mathcal{E}(p \vee q)\} \cup \{(\sigma, \mathbf{false}) \mid (\sigma, \mathbf{true}) \in \mathcal{E}(p \vee q)\}$.

By the definition of $\mathcal{E}$ this further expands to:
$\{(\sigma, \mathbf{true}) \mid (\sigma, \mathbf{false}) \in \mathcal{E}(p) \wedge (\sigma, \mathbf{false}) \in \mathcal{E}(q)\} \cup$
$\{(\sigma, \mathbf{false}) \mid (\sigma, \mathbf{true}) \in \mathcal{E}(p)\} \cup \{(\sigma, \mathbf{false}) \mid (\sigma, \mathbf{true}) \in \mathcal{E}(q)\}$.

By the definition of $\mathcal{E}$, $\mathcal{E}((\neg\, p) \wedge (\neg\, q))$ expands to:
$\{(\sigma, \mathbf{true}) \mid (\sigma, \mathbf{true}) \in \mathcal{E}(\neg\, p) \wedge (\sigma, \mathbf{true}) \in \mathcal{E}(\neg\, q)\} \cup$
$\{(\sigma, \mathbf{false}) \mid (\sigma, \mathbf{false}) \in \mathcal{E}(\neg\, p)\} \cup \{(\sigma, \mathbf{false}) \mid (\sigma, \mathbf{false}) \in \mathcal{E}(\neg\, q)\}$.

By the definition of $\mathcal{E}$ this further expands to:
$\{(\sigma, \mathbf{true}) \mid (\sigma, \mathbf{false}) \in \mathcal{E}(p) \wedge (\sigma, \mathbf{false}) \in \mathcal{E}(q)\} \cup$
$\{(\sigma, \mathbf{false}) \mid (\sigma, \mathbf{true}) \in \mathcal{E}(p)\} \cup \{(\sigma, \mathbf{false}) \mid (\sigma, \mathbf{true}) \in \mathcal{E}(q)\}$.

Thus $\mathcal{E}(\neg\, (p \vee q)) = \mathcal{E}((\neg\, p) \wedge (\neg\, q))$ as required. $\square$

Lemma 11. Any formula $\neg\,\neg\, p$ is logically equivalent to p.
Proof. By the definition of $\mathcal{E}$, $\mathcal{E}(\neg\,\neg\, p)$ expands to:
$\{(\sigma, \mathbf{true}) \mid (\sigma, \mathbf{false}) \in \mathcal{E}(\neg\, p)\} \cup \{(\sigma, \mathbf{false}) \mid (\sigma, \mathbf{true}) \in \mathcal{E}(\neg\, p)\}$.

By the definition of $\mathcal{E}$ this further expands to:
$\{(\sigma, \mathbf{true}) \mid (\sigma, \mathbf{true}) \in \mathcal{E}(p)\} \cup \{(\sigma, \mathbf{false}) \mid (\sigma, \mathbf{false}) \in \mathcal{E}(p)\}$.

This immediately reduces to $\mathcal{E}(p)$ as required ($\mathcal{E}(\neg\,\neg\, p) = \mathcal{E}(p)$). $\square$

Lemma 12. Any formula $p \vee (q \wedge r)$ is logically equivalent to $(p \vee q) \wedge (p \vee r)$
Proof. By the definition of $\mathcal{E}$, $\mathcal{E}(p \vee (q \wedge r))$ expands to:
$\{(\sigma, \mathbf{true}) \mid (\sigma, \mathbf{true}) \in \mathcal{E}(p)\} \cup$
$\{(\sigma, \mathbf{true}) \mid (\sigma, \mathbf{true}) \in \mathcal{E}(q \wedge r)\} \cup$
$\{(\sigma, \mathbf{false}) \mid (\sigma, \mathbf{false}) \in \mathcal{E}(p) \wedge (\sigma, \mathbf{false}) \in \mathcal{E}(q \wedge r)\}$.

By the definition of $\mathcal{E}$, this further expands to:
$\{(\sigma, \mathbf{true}) \mid (\sigma, \mathbf{true}) \in \mathcal{E}(p)\} \cup$
$\{(\sigma, \mathbf{true}) \mid (\sigma, \mathbf{true}) \in \mathcal{E}(q) \wedge (\sigma, \mathbf{true}) \in \mathcal{E}(r)\} \cup$
$\{(\sigma, \mathbf{false}) \mid (\sigma, \mathbf{false}) \in \mathcal{E}(p) \wedge (\sigma, \mathbf{false}) \in \mathcal{E}(q)\} \cup$
$\{(\sigma, \mathbf{false}) \mid (\sigma, \mathbf{false}) \in \mathcal{E}(p) \wedge (\sigma, \mathbf{false}) \in \mathcal{E}(r)\}$.

By the definition of $\mathcal{E}$, $\mathcal{E}((p \vee q) \wedge (p \vee r))$ expands to:
$\{(\sigma, \mathbf{true}) \mid (\sigma, \mathbf{true}) \in \mathcal{E}(p \vee q) \wedge (\sigma, \mathbf{true}) \in \mathcal{E}(p \vee r)\} \cup$
$\{(\sigma, \mathbf{true}) \mid (\sigma, \mathbf{false}) \in \mathcal{E}(p \vee q)\} \cup$
$\{(\sigma, \mathbf{false}) \mid (\sigma, \mathbf{false}) \in \mathcal{E}(p \vee r)\}$.

By the definition of $\mathcal{E}$, this further expands and simplifies to the set definition of $\mathcal{E}(p \vee (q \wedge r))$ presented above. $\square$

Lemma 13. Any formula $\Delta \neg\, l$ is logically equivalent to Δl.
Proof. By the definition of $\mathcal{E}$, $\mathcal{E}(\Delta \neg\, l)$ expands to:
$\{(\sigma, \mathbf{true}) \mid \sigma \in \mathbf{dom}\, \mathcal{E}(\neg\, l)\} \cup \{(\sigma, \mathbf{false}) \mid \sigma \in (\Sigma \setminus \mathbf{dom}\, \mathcal{E}(\neg\, l))\}$.

By the definition of $\mathcal{E}$ this further expands to:
$\{(\sigma, \mathbf{true}) \mid \sigma \in \mathbf{dom}\, \mathcal{E}(l)\} \cup \{(\sigma, \mathbf{false}) \mid \sigma \in (\Sigma \setminus \mathbf{dom}\, \mathcal{E}(l))\}$, which is equiv-

alent according to the definition of $\mathcal{E}$, to $\mathcal{E}(\Delta l)$ as required. $\square$

Also note that $p \vee \neg\, p \vee \neg\, \Delta p$ is equivalent to the truth value true, (the law of the excluded fourth, $p^{\Sigma} \cup (\neg\, p)^{\Sigma} \cup (\neg\, \Delta p)^{\Sigma} = \Sigma$). The formula $\Delta(\Delta p)$ and the formula $(\Delta p \vee \neg\, \Delta p)$ are also equivalent to the truth value true. Furthermore, the formula $(\Delta p \wedge \neg\, \Delta p)$ is equivalent to the truth value false. Remember that in LPF the simplification of the formula $p \vee \neg\, p$ (the law of the excluded middle) to the truth value true and the simplification of the formula $p \wedge \neg\, p$ to the truth value false cannot be made, because of the case when p is undefined.

All of these equivalences used when converting a formula into CNF hold in LPF. Additionally, every propositional formula in LPF can be converted into an equivalent formula that is in CNF.

Theorem 14. Every LPF propositional formula $e \in \mathit{Expr}$, can be converted into an equivalent formula that is in CNF.
Proof. This theorem follows immediately from the fact that all of the required conversions hold in LPF (see Lemmas 10, 11, 12 and 13). The proofs of other laws which are not presented here follow in a similar way. $\square$

The conversion of a CNF formula into *clausal form* relies on the idempotence properties and the commutativity of conjunctions and disjunctions. These properties all hold in LPF and the proof of one of these properties follows.

Lemma 15. Any formula $p \vee p$ is logically equivalent to p.
Proof. By the definition of $\mathcal{E}$, $\mathcal{E}(p \vee p)$ expands to:
$\{(\sigma, \mathbf{true}) \mid (\sigma, \mathbf{true}) \in \mathcal{E}(p)\} \cup \{(\sigma, \mathbf{true}) \mid (\sigma, \mathbf{true}) \in \mathcal{E}(p)\} \cup$
$\quad \{(\sigma, \mathbf{false}) \mid (\sigma, \mathbf{false}) \in \mathcal{E}(p) \wedge (\sigma, \mathbf{false}) \in \mathcal{E}(p)\}$.

By the definition of a set, the first two sets from the set union definition presented above are equivalent ($A \cup A = A$) and the third set additionally can be simplified. The resulting set union is equivalent to $\mathcal{E}(p)$ as required. $\square$

Theorem 16. Every LPF propositional formula $e \in \mathit{Expr}$, can be converted into an equivalent clausal form.
Proof. This immediately follows from Theorem 14, Lemma 15, the other idempotent property ($\mathcal{E}(p \wedge p) = \mathcal{E}(p)$), as well as the fact that the commutativity of conjunctions and disjunctions all hold in LPF which all follow by the

definition of $\mathcal{E}$. □

In order to try to reduce the size of a resulting clausal form formula, the *absorption properties* can be used, whereby both $p \wedge (p \vee q)$ and $p \vee (p \wedge q)$ can be simplified to p. This is illustrated in the following proof.

Lemma 17. Any formula $p \wedge (p \vee q)$ and any formula $p \vee (p \wedge q)$ are both logically equivalent to p.

Proof. First consider the case of $p \wedge (p \vee q)$ being logically equivalent to p.

By the definition of $\mathcal{E}$, $\mathcal{E}(p \wedge (p \vee q))$ expands to:

$\{(\sigma, \mathbf{true}) \mid (\sigma, \mathbf{true}) \in \mathcal{E}(p) \wedge (\sigma, \mathbf{true}) \in \mathcal{E}(p \vee q)\} \cup$
$\{(\sigma, \mathbf{false}) \mid (\sigma, \mathbf{false}) \in \mathcal{E}(p)\} \cup$
$\{(\sigma, \mathbf{false}) \mid (\sigma, \mathbf{false}) \in \mathcal{E}(p \vee q)\}$.

By the definition of $\mathcal{E}$ this further expands to:

$\{(\sigma, \mathbf{true}) \mid (\sigma, \mathbf{true}) \in \mathcal{E}(p) \wedge (\sigma, \mathbf{true}) \in \mathcal{E}(p)\} \cup$
$\{(\sigma, \mathbf{true}) \mid (\sigma, \mathbf{true}) \in \mathcal{E}(p) \wedge (\sigma, \mathbf{true}) \in \mathcal{E}(q)\} \cup$
$\{(\sigma, \mathbf{false}) \mid (\sigma, \mathbf{false}) \in \mathcal{E}(p)\} \cup$
$\{(\sigma, \mathbf{false}) \mid (\sigma, \mathbf{false}) \in \mathcal{E}(p) \wedge (\sigma, \mathbf{false}) \in \mathcal{E}(q)\}$.

The second set from the set union is a subset of the first set; similarly, the fourth set from the set union is a subset of the third set. The first set (after the trivial simplification) and the third set immediately match the expansion of $\mathcal{E}(p)$, which is:

$\{(\sigma, \mathbf{true}) \mid (\sigma, \mathbf{true}) \in \mathcal{E}(p)\} \cup \{(\sigma, \mathbf{false}) \mid (\sigma, \mathbf{false}) \in \mathcal{E}(p)\}$, and this concludes the first case.

The proof of the other case, that is, $p \vee (p \wedge q)$ being logically equivalent to p ($\mathcal{E}(p \vee (p \wedge q)) = \mathcal{E}(p)$) is similar to the proof of the case presented above.

□

Predicate Logic

The process of converting a formula into PNF is as follows [BA01]:

- *standardise the variables apart*, i.e. rename variables, where necessary, so that no two quantifiers bind the same variable name;
- push any negation operators inwards so that they only apply to atomic formulae, e.g. through the use of de Morgan's Laws and through conversions such as $\neg \exists x \cdot p$ to $\forall x \cdot \neg p$; and
- move any quantifiers out of the matrix, e.g. through conversions such as $p \vee \exists x \cdot q$ to $\exists x \cdot (p \vee q)$, and $p \vee \forall x \cdot q$ to $\forall x \cdot p \vee q$.

but since predicate LPF is being considered, the standard process outlined above for converting a two-valued classical logic formula into PNF needs extending, since any Δ needs pushing into the matrix, before the CNF conversions can be used as normal on the matrix.

Given Δe, where e is a quantified formula, then e should first be put into PNF and then the Δ can be pushed inwards, followed by any further use of the PNF conversions as required (after pushing in any Δ any negation that was to the left of the Δ can be pushed in). The following equivalences are needed:

- $\Delta(\forall i \cdot p(i))$ is logically equivalent to:

  $$\neg\,(\exists i \cdot \neg\,\Delta p(i) \wedge \forall i \cdot (p(i) \vee \neg\,\Delta p(i)))$$

  (i.e. the negation of the cases that make $\Delta(\forall i \cdot p(i))$ false, which is when $p(i)$ is always undefined, or when $p(i)$ is true at least once, undefined at least once and is always true or undefined, so $p(i)$ is never false), which gives rise to:

  $$\forall i \cdot \Delta p(i) \vee \exists i \cdot (\neg\,p(i) \wedge \Delta p(i))$$

- $\Delta(\exists i \cdot p(i))$ is logically equivalent to:

  $$\neg\,(\exists i \cdot \neg\,\Delta p(i) \wedge \forall i \cdot (\neg\,p(i) \vee \neg\,\Delta p(i)))$$

Again, note that $\Delta(\forall i \cdot p(i))$ and $\Delta(\exists i \cdot p(i))$ are two-valued, and thus the formulations above that maintain the presence of Δ are needed since logical equivalence is being sought. Unfortunately, they give rise to much larger formulae. The formula $\Delta(\forall i \cdot p(i))$ is represented in clausal form as:

$$\{\{\neg\,p(c), \Delta p(x)\}, \{\Delta p(c), \Delta p(x)\}\}$$

while the formula $\neg\,\Delta(\forall i \cdot p(i))$ is represented in clausal form as:

$$\{\{p(x), \neg\,\Delta p(x)\}, \{\neg\,\Delta p(c)\}\}$$

where c is a Skolem constant.

Illustrative proofs to show that the above holds in LPF follow. Again such proofs are done with respect to the $\mathcal{E}$ semantic function definition presented in Section 3.4.

Lemma 18. Any formula $\neg\,\exists x \cdot p$ is logically equivalent to $\forall x \cdot \neg\, p$.
Proof. By the definition of $\mathcal{E}$, $\mathcal{E}(\neg\,\exists x \cdot p)$ expands to:
$\{(\sigma, \mathbf{true}) \mid (\sigma, \mathbf{false}) \in \mathcal{E}(\exists x \cdot p)\} \cup \{(\sigma, \mathbf{false}) \mid (\sigma, \mathbf{true}) \in \mathcal{E}(\exists x \cdot p)\}$.

By the definition of $\mathcal{E}$ this further expands to:
$\{(\sigma, \mathbf{true}) \mid \sigma \in \Sigma \wedge \mathbf{rng}\,(\{\sigma \dagger \{x \mapsto i\} \mid i{:}\,\mathbb{Z}\}) = \{\mathbf{false}\}\} \cup$
$\quad\{(\sigma, \mathbf{false}) \mid \sigma \in \Sigma \wedge \mathbf{true} \in \mathbf{rng}\,(\{\sigma \dagger \{x \mapsto i\} \mid i{:}\,\mathbb{Z}\} \lhd \mathcal{E}(p))\}$.

By the definition of $\mathcal{E}$, $\mathcal{E}(\forall x \cdot \neg\, p)$ expands to:
$\{(\sigma, \mathbf{true}) \mid \sigma \in \Sigma \wedge \mathbf{rng}\,(\{\sigma \dagger \{x \mapsto i\} \mid i{:}\,\mathbb{Z}\} \lhd \mathcal{E}(\neg\, p)) = \{\mathbf{true}\}\} \cup$
$\quad\{(\sigma, \mathbf{false}) \mid \sigma \in \Sigma \wedge \mathbf{false} \in \mathbf{rng}\,(\{\sigma \dagger \{x \mapsto i\} \mid i{:}\,\mathbb{Z}\} \lhd \mathcal{E}(\neg\, p))\}$.

By the definition of $\mathcal{E}$ this further expands to:
$\{(\sigma, \mathbf{true}) \mid \sigma \in \Sigma \wedge \mathbf{rng}\,(\{\sigma \dagger \{x \mapsto i\} \mid i{:}\,\mathbb{Z}\} \lhd \mathcal{E}(p)) = \{\mathbf{false}\}\} \cup$
$\quad\{(\sigma, \mathbf{false}) \mid \sigma \in \Sigma \wedge \mathbf{true} \in \mathbf{rng}\,(\{\sigma \dagger \{x \mapsto i\} \mid i{:}\,\mathbb{Z}\} \lhd \mathcal{E}(p))\}$.

The two sets formed are equivalent which concludes the result ($\mathcal{E}(\neg\,\exists x {\cdot} p) = \mathcal{E}(\forall x \cdot \neg\, p)$) as required. □

The following proof assumes that all variables are standardised apart. Also remember that, in the $\mathcal{E}$ semantic definition all quantification is performed only over the set of integers ($\mathbb{Z}$), so x in the following proof is always defined.

Lemma 19. Let p be a formula that contains no free occurrences of the variable x. Then any formula $p \vee \exists x \cdot q$ is logically equivalent to $\exists x \cdot (p \vee q)$.
Proof. By the definition of $\mathcal{E}$, $\mathcal{E}(p \vee \exists x \cdot q)$ expands to:
$\{(\sigma, \mathbf{true}) \mid (\sigma, \mathbf{true}) \in \mathcal{E}(p)\} \cup$
$\quad\{(\sigma, \mathbf{true}) \mid (\sigma, \mathbf{true}) \in \mathcal{E}(\exists x \cdot q)\} \cup$
$\quad\{(\sigma, \mathbf{false}) \mid (\sigma, \mathbf{false}) \in \mathcal{E}(p) \wedge (\sigma, \mathbf{false}) \in \mathcal{E}(\exists x \cdot q)\}$.

By the definition of $\mathcal{E}$ this further expands to:
$\{(\sigma, \mathbf{true}) \mid (\sigma, \mathbf{true}) \in \mathcal{E}(p)\} \cup$
$\quad\{(\sigma, \mathbf{true}) \mid \sigma \in \Sigma \wedge \mathbf{true} \in \mathbf{rng}\,(\{\sigma \dagger \{x \mapsto i\} \mid i{:}\,\mathbb{Z}\} \lhd \mathcal{E}(q))\} \cup$
$\quad\{(\sigma, \mathbf{false}) \mid \sigma \in \Sigma \wedge (\sigma, \mathbf{false}) \in \mathcal{E}(p) \wedge$
$\qquad\mathbf{rng}\,\{\sigma \dagger \{x \mapsto i\} \mid i{:}\,\mathbb{Z}\} \lhd \mathcal{E}(q) = \{\mathbf{false}\}\}$.

By the definition of $\mathcal{E}$, $\mathcal{E}(\exists x \cdot (p \vee q))$ expands to:
$\{(\sigma, \mathbf{true}) \mid \sigma \in \Sigma \wedge \mathbf{true} \in \mathbf{rng}\,(\{\sigma \dagger \{x \mapsto i\} \mid i{:}\,\mathbb{Z}\} \lhd \mathcal{E}(p \vee q))\} \cup$
$\quad\{(\sigma, \mathbf{false}) \mid \sigma \in \Sigma \wedge \mathbf{rng}\,\{\sigma \dagger \{x \mapsto i\} \mid i{:}\,\mathbb{Z}\} \lhd \mathcal{E}(p \vee q) = \{\mathbf{false}\}\}$.

By the definition of $\mathcal{E}$ this further expands to:
$\{(\sigma, \mathbf{true}) \mid \sigma \in \Sigma \wedge \mathbf{true} \in \mathbf{rng}\,(\{\sigma \dagger \{x \mapsto i\} \mid i{:}\,\mathbb{Z}\} \lhd \mathcal{E}(p))\} \cup$
$\quad\{(\sigma, \mathbf{true}) \mid \sigma \in \Sigma \wedge \mathbf{true} \in \mathbf{rng}\,(\{\sigma \dagger \{x \mapsto i\} \mid i{:}\,\mathbb{Z}\} \lhd \mathcal{E}(q))\} \cup$

$$\{(\sigma, \mathbf{false}) \mid \sigma \in \Sigma \wedge \mathbf{rng}\,\{\sigma \dagger \{x \mapsto i\} \mid i{:}\,\mathbb{Z}\} \lhd \mathcal{E}(p) = \{\mathbf{false}\} \wedge$$
$$\mathbf{rng}\,\{\sigma \dagger \{x \mapsto i\} \mid i{:}\,\mathbb{Z}\} \lhd \mathcal{E}(q) = \{\mathbf{false}\}\}.$$

Since the variables have first been standardised apart and by the assumption that x is not free in p, if p denotes true or false when p contains no reference to x, then quantifying over x causes no change in the result. Thus the two sets formed are equivalent as required. $\square$

Every first-order LPF formula can be converted into an equivalent formula that is in PNF.

Theorem 20. Every LPF formula $e \in \mathit{Expr}$, can be converted into an equivalent formula that is in PNF.
Proof. This follows since the conversions required for converting a two-valued classical logic formula into PNF all hold in LPF. The proofs of these conversions follow in a similar way to the proofs of the conversions presented in Lemmas 18 and 19, and because the renaming of variables (through the standardising apart process) has no effect on logical equivalence. $\square$

Skolemisation also carries over to LPF and furthermore, because satisfiability is being sought all of the Skolem constants (0-ary functions)/Skolem functions introduced are total. This result is key to results that are provided later.

Theorem 21. Let S' be the formula formed by Skolemising the formula S, where it is assumed that every Skolem function introduced is a distinct function symbol not present in S. It must then follow that S and S' are equi-satisfiable.
Proof. The proof follows like a two-valued classical proof, e.g. [BA01], but is presented here with respect to $\mathcal{E}$ and Σ. If S contains no existential quantifier then no change results from performing Skolemisation and the result follows immediately. In the case that S contains at least the one existential quantifier then there are two cases to consider:

1. If S is satisfied then S' must be satisfied: Suppose $\forall x \cdot \exists y \cdot P(x, y)^{\sigma}$, where $\sigma \in \Sigma$, then it needs to be shown that an interpretation $\sigma' \in \Sigma$ exists such that $\forall x \cdot P(x, f(x))^{\sigma'}$. Thus for every possible value for x there exists a value for y that causes $P(x, y)$ to evaluate to true. A function f such that $f(x) = y$ exists and the interpretation σ' can then be defined as $\sigma' = \sigma \dagger \{f \mapsto \beta\}$, where β is a *Function* from each possible value for x to a corresponding result y. There may be many witness values

for the existential quantifier, but for a function it requires restricting to the one such witness value, that is, one specific value for y for each value of x, cf. the *Axiom of Choice* [Har09, §3.6]. It then follows that if S is satisfied (with the existential quantifier) then S' is satisfied since the $mk\_FuncCall(f, al)$ case of the $\mathcal{E}$ semantic function definition can return a value that would otherwise have been produced by the existential quantifier case of the $\mathcal{E}$ semantic function definition.

2. If S' is satisfied then S must be satisfied: In the example from case 1 if it is the case that $\forall x \cdot P(x, f(x))^{\sigma'}$ then σ' must have an interpretation for the Skolem function f. Therefore σ $\forall x \cdot \exists y \cdot P(x, y)$ through taking for every x, $y = f(x)$. $\square$

As usual the matrix can now be put into CNF to arrive at the clausal form representation. The universal quantifiers can be omitted from the clausal form representation, as still in LPF the clauses are considered to be universally quantified.

Theorem 22. Every closed LPF formula $e \in Expr$, can be converted into a formula that is in clausal form, such that the original formula and the formula in clausal form are equi-satisfiable.
Proof. This is an immediate consequence of Theorems 20 and 21 and, for the conversion of the matrix, Theorems 14 and 16. $\square$

6.4 Unification

Before the resolution proof procedure is discussed, the concepts of a *substitution* and *unification* are introduced. When considering the predicate calculus, unification [Rob65, BA01, Har09, Bun10] is an integral part of the resolution proof procedure. Here unification is only used within a resolution step.

A *substitution* as standard is a map of variables to terms of the form:

$$\phi = \{\psi_1 \mapsto \beta_1, \ldots, \psi_n \mapsto \beta_n\}$$

where each ψ_i is a distinct variable and each β_i is a term.

The application of a substitution ϕ to a term α, denoted $\phi[\alpha]$, is the simultaneous replacement of each $\psi_i \in \mathbf{dom}\,\phi$ in α with the respective $\phi(\psi_i)$.

Unification is the process of finding a substitution that makes terms identical, that is, finding whether there exists a substitution ϕ for the variables in two terms α and β, such that:

$$\phi[\alpha] = \phi[\beta]$$

If such a substitution exists then it is known as a *unifier* for α and β.

A set of terms that can be unified has what is known as a *most general unifier* (*mgu*), which is unique up to variable renaming. A mgu for α and for β is a unifier ϕ such that any other unifier ϕ' for α and for β can be derived by composing ϕ with a further substitution ϕ''.

Not all terms can be unified. There is no unifier for the terms $f(x)$ and $g(y)$ where f and g are different function symbols. There is also no unifier for $f(x)$ and $f(g(x))$, since x "occurs" within the larger term $g(x)$, cf. the *occurs check* [BA01, §7.7].

Unification here has been performed on *uninterpreted function symbols*, that is, that only the name and the arity of the function symbols has been taken into account.

A *unification algorithm* [Rob65, Vad88, BA01] takes as input a set of terms and if there exists a unifier for the input yields an mgu. If the input terms are not unifiable, then the unification algorithm will terminate with the result that there is no unifier for the input.

The issue of treating partial terms in substitutions is discussed in Section 6.6. This is a key issue that must be resolved in order to carry the resolution refutation procedure over to LPF.

6.5 Resolution

Resolution and refutation can now be considered in the rest of this chapter. It is this work on resolution and refutation for LPF that carries the main contribution of this chapter. The concepts are introduced first in the context of two-valued classical logic, and then the procedures are considered for LPF.

6.5.1 Two-Valued Classical Logic Recap

Over the years, numerous proof procedures have been developed that can be used to show whether a formula is (un)satisfiable in two-valued classical logic. These include the *semantic tableaux* proof procedure [BA01] and the *resolution* proof procedure [Rob65, BA01, Har09, Bun10]; it is the latter that is focused on here. *Resolution* is used in numerous automated theorem provers and works for both the propositional and the predicate calculus, but its real payoff is for the latter.

The resolution rule works by taking two clauses that contain contradictory literals (l and $\neg\, l$) and using this knowledge to infer a new clause. This relies on the fact that l and $\neg\, l$ cannot both be true in the same interpretation. For propositional logic this works as follows: given two clauses C_1 and C_2 which both include the literal l where it is positive in one clause $\{l\} \subseteq C_1$ and negative in the other clause $\{\neg\, l\} \subseteq C_2$, a resolvent can be inferred from these two clauses which is the union of the two clauses without the complementary ("clashing") literal, i.e. $(C_1 \setminus \{l\}) \cup (C_2 \setminus \{\neg\, l\})$. The reasoning is that if l is true then another literal must cause C_2 to be satisfied, and if l is false then another literal must cause C_1 to be satisfied.

An approach that can be used for predicate logic is *binary resolution* [BA01, Har09] which utilises *unification* to generate "clashing clauses" that can then be resolved. Since the clauses can contain variables, the aim is to resolve on the most general forms of clauses. So for instance, if two literals l_1 and $\neg\, l_2$ can be unified by a mgu ϕ, where $\{l_1\} \subseteq C_1$ and $\{\neg\, l_2\} \subseteq C_2$, then C_1 and C_2 can be resolved, inferring a new clause: $(\phi[C_1] \setminus \phi[\{l_1\}]) \cup (\phi[C_2] \setminus \phi[\{\neg\, l_2\}])$.

The resolution proof procedure takes as input a set of clauses and repeatedly applies the resolution rule to infer new clause(s). The process is iterative so new clauses inferred are added to the original set of clauses, so that they can be used to infer further resolvents. Resolution maintains satisfiability, if the set of clauses are satisfiable then it follows that the set of clauses after resolving are satisfiable.

If the empty clause ($\Box$) (which is unsatisfiable) is inferred, then the set of clauses is unsatisfiable. If the empty clause cannot be inferred, and no more *new* resolvents (clauses) can be inferred, then the set of clauses must be satisfiable. There is also the possibility that the resolution proof procedure continues deriving new clauses forever.

Resolution is in fact a generalisation of the *modus ponens* rule:

$$\boxed{\textit{modus-ponens}}\ \frac{p;\ p \;\Rightarrow\; q}{q}$$

A technique called *factoring* [BA01, Rob65, Bun10] is used along with resolution. Factoring is the merging of unifiable literals in a single clause. Given a clause C where $\{l_1\} \subseteq C$ and $\{l_2\} \subseteq C$ then the clause $\phi[C \setminus \{l_2\}]$ can be inferred, where ϕ is an mgu of l_1 and l_2. Resolution with factoring is refutationally complete [Rob65].

The Davis Putnam procedure [Har09] is a method which can decide the satisfiability of a propositional formula in CNF, where there are three rules used. One of which is resolution, and the other two are known as the *affirmative-negative rule* and the *one-literal rule* which can reduce the number of literals that need to be considered. These two rules are considered in the next subsection. Preferential use of these other two rules can be useful for efficiency [Har09].

6.5.2 LPF

The proofs in this section assume no use of Δ; the Δ operator is a meta-level operator and it is not written in normal assertions. The Δ operator is introduced in a restricted circumstance when considering refutation; this is considered in Section 6.6.

The key property underlying resolution is the cancellation of contradictory information (literals) from clauses. In LPF (as in two-valued classical logic) an assertion p and its negation $\neg p$ cannot both be true in an interpretation.

Lemma 23. The set of clauses $\{\{p\}, \{\neg p\}\}$ cannot be true in an interpretation, i.e. there exists no $\sigma \in \Sigma$ such that $(p \wedge \neg p)^\sigma$.

Proof. By the definition of $\mathcal{E}$, $\mathcal{E}(p \wedge \neg p)$ expands to:

$\{(\sigma, \mathbf{true}) \mid (\sigma, \mathbf{true}) \in \mathcal{E}(p) \wedge (\sigma, \mathbf{true}) \in \mathcal{E}(\neg p)\} \cup$
$\quad \{(\sigma, \mathbf{false}) \mid (\sigma, \mathbf{false}) \in \mathcal{E}(p)\} \cup$
$\quad \{(\sigma, \mathbf{false}) \mid (\sigma, \mathbf{false}) \in \mathcal{E}(\neg p)\}$.

By the definition of $\mathcal{E}$ this further expands to:

$\{(\sigma, \mathbf{true}) \mid (\sigma, \mathbf{true}) \in \mathcal{E}(p) \wedge (\sigma, \mathbf{false}) \in \mathcal{E}(p)\} \cup$
$\quad \{(\sigma, \mathbf{false}) \mid (\sigma, \mathbf{false}) \in \mathcal{E}(p)\} \cup$
$\quad \{(\sigma, \mathbf{false}) \mid (\sigma, \mathbf{true}) \in \mathcal{E}(p)\}$.

By Lemma 1 it follows that p cannot be both true and false in any σ and therefore the first set above is equivalent to $\{\}$, leaving no set where $p \wedge \neg p$ evaluates to true in any $\sigma \in \Sigma$, that is, $p^\Sigma \cap (\neg p)^\Sigma = \{\}$. □

If two clauses C_1 and C_2 are true in an interpretation σ, then a resolvent of C_1 and C_2 is true in the interpretation σ. This also applies to the LPF case. First consider the proof for the propositional case in LPF.

Theorem 24. Given two propositional clauses C_1 and C_2 which are true in some σ, where $\{l\} \subseteq C_1$ and $\{\neg l\} \subseteq C_2$ and l is a literal, then the resolvent $C_3 = (C_1 \setminus \{l\}) \cup (C_2 \setminus \{\neg l\})$, is true in the same σ.

Proof. By assumption, C_1^σ and C_2^σ hold for some $\sigma \in \Sigma$. For an arbitrary satisfying interpretation σ there are three cases to consider:

1. l^σ: By the definition of $\mathcal{E}$, it follows that both C_1^σ and $\neg l^{\overline{\sigma}}$. Since by assumption it is known that C_2^σ there must exist another disjunct (literal, $\neg l \neq l'$) $\{l'\} \subseteq C_2$ that ensures that C_2 is satisfied, i.e. $(C_2 \setminus \{\neg l\})^\sigma$. Thus C_3 is satisfied (C_3^σ) by the definition of $\mathcal{E}$ since $\{l'\} \subseteq C_3$ and l'^σ.

2. $l^{\overline{\sigma}}$: This follows by a similar argument to case 1, as there must exist another disjunct ($l \neq l'$) $\{l'\} \subseteq C_1$ that ensures that C_1 is satisfied,

i.e. $(C_1 \setminus \{l\})^\sigma$ and C_2^σ holds because $\neg\, l^\sigma$. Thus C_3 is satisfied (C_3^σ) by the definition of $\mathcal{E}$ since $\{l'\} \subseteq C_3$ and l'^σ.

3. $\sigma \notin \mathbf{dom}\,\mathcal{E}(l)$: By the definition of $\mathcal{E}$, it also follows that $\sigma \notin \mathbf{dom}\,\mathcal{E}(\neg\, l)$. Thus another disjunct $(\{l'\} \subseteq C_1)$ must ensure that C_1 is satisfied, i.e. $(C_1 \setminus \{l\})^\sigma$ and another disjunct $(\{l''\} \subseteq C_2)$ must ensure that C_2 is satisfied, i.e. $(C_2 \setminus \{\neg\, l\})^\sigma$, where $l \neq l'$ and $\neg\, l \neq l''$. Thus C_3 is satisfied (C_3^σ) by the definition of $\mathcal{E}$ since $\{l', l''\} \subseteq C_3$ and l'^σ and l''^σ.

Thus, in all cases for an arbitrary σ where both C_1^σ and C_2^σ hold, it is the case that C_3^σ holds. □

Now the predicate case is considered for LPF which, as mentioned earlier, makes use of unification.

Corollary 25. Given two clauses C_1 and C_2 which are true in some σ, where $\{l_1\} \subseteq C_1$ and $\{\neg\, l_2\} \subseteq C_2$ and both l_1 and $\neg\, l_2$ are literals which can be unified by an mgu ϕ, then a resolvent $C_3 = (\phi[C_1] \setminus \phi[\{l_1\}]) \cup (\phi[C_2] \setminus \phi[\{\neg\, l_2\}])$, is true in the same σ.

Proof. By assumption, C_1^σ and C_2^σ hold for some $\sigma \in \Sigma$. Since ϕ makes the two literals l_1 and l_2 identical (l'), i.e. $l' = \phi[l_1] = \phi[l_2]$, it cannot be the case that both l' and $\neg\, l'$ are true in any $\sigma \in \Sigma$ by Lemma 23. The result then follows in a similar way to Theorem 24. □

When using the resolution refutation procedure, the use of unification does need restricting since validity is being sought, this is achieved by the inclusion of unification constraints. Refutation procedures in LPF are considered in Section 6.6. The use of factoring in a refutation proof also needs protecting through the inclusion of unification constraints.

From satisfiable clauses, only satisfiable clauses can be inferred. Thus if a resolvent is ever the empty clause then the set of clauses must have been unsatisfiable, i.e. there must be a contradiction.

Theorem 26. If the empty clause is ever inferred by resolution on the set of clauses S, then S must be unsatisfiable.

Proof. The fewest number of clauses that can be used to infer the empty clause is two where the only literal in each clause is identical (same propositional variable or they unify), only the literal is positive in the one clause and negative in the other clause. By Lemma 23, it follows that both of these clauses cannot be true (the set of clauses is unsatisfiable) and thus the empty clause (which is unsatisfiable) is inferred. □

The other two rules mentioned earlier in the Davis-Putnam procedure also hold in LPF, and the proofs needed for LPF follow similarly to [Har09]; they are presented in terms of the $\mathcal{E}$ and Σ definitions.

Lemma 27. Suppose there is a set of clauses S and $C_1 \subseteq S$, for which there is a literal (positive or negative) $\{l\} \subseteq C_1$ and $\neg\, l$ does not occur in any $C_i \in S$. Let S' be the set of clauses formed from S by removing every clause containing l. It follows that S and S' are equi-satisfiable.

Proof. There are two cases to consider:

- If S is satisfiable then S' is satisfiable since $S' \subset S$.
- If S' is satisfiable then S is satisfiable since every $C_i \in S'$ is true (C_i^{σ}) for at least one σ, where $\sigma \in \Sigma$. If σ is extended to σ' so that $\sigma' = \sigma \dagger \{l \mapsto$ **true**$\}$ then for each $C_j \in S$ it follows that $C_j^{\sigma'}$.

□

Lemma 28. Suppose that there is a set of clauses S and $C_1 \subseteq S$, for which there is a literal (positive or negative) $\{l\} \subseteq C_1$ and l is the only literal contained within C_1. Let S' be the set of clauses formed from S by removing $\neg\, l$ from every other clause in S and removing every clause that contains l. It follows that S and S' are equi-satisfiable.
Proof. There are two cases to consider:

- If S is satisfiable then S' is satisfiable since if l is the only literal contained within a clause then this literal must be true (l^{σ}) and thus C_1 must be true (C_1^{σ}), if the set of clauses is satisfiable (S^{σ}), where $\sigma \in \Sigma$. Thus another literal in every other $C_i \subseteq S$, must be true, where $\{\neg\, l\} \subseteq C_i$. Since l must be true, then every clause $C_i \subseteq S$ where $\{l\} \subseteq C_i$ must be true.
- If S' is satisfiable then S is satisfiable in a given σ as σ can be extended to σ' so that $\sigma' = \sigma \dagger \{l \mapsto$ **true**$\}$.

□

6.6 Refutation Procedures

6.6.1 Two-Valued Classical Logic Recap

In two-valued classical logic, a formula e is valid iff $\neg e$ is unsatisfiable [BA01]. This well known duality between validity and satisfiability follows from the fact that there are only two possible values (true and false) that a formula in two-valued classical logic can take. So, if a formula e is valid then every interpretation must make e true, and thus every interpretation must make $\neg e$ false, that is, $\neg e$ must be unsatisfiable. Conversely, if $\neg e$ is satisfiable, then there must exist at least one interpretation that makes $\neg e$ true and thus e false, thus as a result e cannot be valid, that is, e is satisfiable iff $\neg e$ is not valid.

The above result is important since it means that a proof procedure for satisfiability can be used for validity (a *refutation procedure* [BA01]) in two-valued classical logic. Hence the validity of a formula is proved by refuting its negation [BA01]. Refutation procedures *can* be more efficient since there is only a requirement to find the one counter example as opposed to having to check that a formula is always true for validity.

The above approach can be extended to reason about logical consequence in two-valued classical logic [BA01, BM99]. To show that a logical consequence statement:

$$\Gamma \models e$$

holds, where $\Gamma = \{e_1, \ldots, e_n\}$ and the commas are to be interpreted as conjunctions, the expression e is negated and the satisfiability of the conjunction:

$$e_1 \wedge \ldots \wedge e_n \wedge \neg e \tag{6.1}$$

is considered, where the assumptions Γ are generally assumed to be true. For instance, Γ could be a consistent set of axioms that are assumed to be true independent of the theorem/goal e that is to be proved.

If formula 6.1 is unsatisfiable then $\Gamma \models e$ must hold, as every interpretation where Γ is true, must make $\neg e$ false and thus e true. However, if formula 6.1 is satisfiable then there must exist at least one interpretation where Γ is true, that also makes $\neg e$ true, and thus e false, and as a result $\Gamma \not\models e$.

Resolution is a refutation procedure which can be used to show that a formula is unsatisfiable [BA01]. The aim is to show that the *goal* e is derivable from the set of expressions Γ (the *assumptions*). The goal e is negated followed by converting all of the formulae within Γ and the formula $\neg e$ into clausal form.

Resolution is then performed on the clauses within the set $\Gamma \cup \{\neg\, e\}$, that is, the combination of the clauses produced during the clausal form conversions into a single set.

Resolution is *refutationally complete* so, if $\Gamma \models e$, then the empty clause will (eventually) be able to be derived from the set of clauses $\Gamma \cup \{\neg\, e\}$; it may never terminate given a satisfiable set of clauses [BA01].

6.6.2 LPF

The application of a refutation procedure in LPF is complicated by the presence of "gaps" in denotations which affect the duality between validity and satisfiability. In LPF, if $\neg\, e$ is satisfiable then e cannot be valid, but if $\neg\, e$ is unsatisfiable then it is not possible to infer that e is valid. The following results clarify the relationship between satisfiability and validity in LPF.

Lemma 29. In LPF, if e is valid then $\neg\, e$ is unsatisfiable.
Proof. By the definition of validity, it is known that e is valid in LPF iff $e^{\Sigma} = \Sigma$ and by the definition of unsatisfiability, that e is unsatisfiable iff $e^{\Sigma} = \{\,\}$. By assumption it is the case that e^{σ} for each $\sigma \in \Sigma$. By the definition of $\mathcal{E}$, if e^{σ} then $\neg\, e^{\overline{\sigma}}$ and since the truth value false is an unsatisfiable value the result is concluded as required. □

Lemma 30. In LPF, if $\neg\, e$ is unsatisfiable then e may be not valid.
Proof. This result is due to the presence of "gaps" in LPF and can be shown using a simple counter example. Consider the Boolean formula $p \vee \neg\, p$ and its negation $\neg\,(p \vee \neg\, p)$ which is unsatisfiable, i.e. $\neg\,(p \vee \neg\, p)^{\Sigma} = \{\}$. However, $p \vee \neg\, p$ is not valid in LPF since any interpretation $\sigma \in \Sigma$ which has a "gap" for p, that is $\sigma \notin \mathbf{dom}\,\mathcal{E}(p)$, results in a "gap" for $p \vee \neg\, p$ ($\sigma \notin \mathbf{dom}\,\mathcal{E}(p \vee \neg\, p)$) and so $(p \vee \neg\, p)^{\Sigma} \subset \Sigma$. □

Lemma 31. In LPF, if e is not valid then $\neg\, e$ may not be satisfiable.
Proof. By the definition of validity $e^{\Sigma} = \Sigma$, thus there must exist an interpretation $\sigma \in \Sigma$ such that e^{σ} does not hold. The proof follows by a simple counter example. Suppose that e is undefined for every $\sigma \in \Sigma$, that is, $\sigma \notin \mathbf{dom}\,\mathcal{E}(e)$ for every $\sigma \in \Sigma$. Then it follows that e is not valid. By the definition of $\mathcal{E}$, if $\sigma \notin \mathbf{dom}\,\mathcal{E}(e)$ then it follows that $\sigma \notin \mathbf{dom}\,\mathcal{E}(\neg\, e)$. Thus it cannot be the case that for any $\sigma \in \Sigma$ that $\neg\, e^{\sigma}$ holds, that is, that $\neg\, e$ is not true for any $\sigma \in \Sigma$. □

Lemma 32. In LPF, if $\neg\, e$ is satisfiable then e is not valid.

Proof. By the definition of satisfiability if $\neg\, e^{\sigma}$ for some $\sigma \in \Sigma$, then by the definition of $\mathcal{E}$ it follows that $e^{\overline{\sigma}}$ holds. As a result it must follow that e cannot be valid $e^{\Sigma} = \Sigma$ since there exists a $\sigma \in \Sigma$ such that $e^{\overline{\sigma}}$. □

Logical Consequence

Applying a refutation procedure to a logical consequence statement in LPF is now considered. Resolution is the satisfiability proof procedure that will be utilised. Consider the logical consequence $\Gamma \models e$ in LPF:

- If there is a satisfying interpretation for $e_1 \wedge \ldots \wedge e_n \wedge \neg\, e$, then at least one interpretation makes all of the expressions $e_1, \ldots, e_n, \neg\, e$ true and so $\Gamma \models e$ cannot hold, ($\Gamma \not\models e$); or
- If there is no satisfying interpretation for $e_1 \wedge \ldots \wedge e_n \wedge \neg\, e$, then there does not exist an interpretation that makes all the expressions $e_1, \ldots, e_n, \neg\, e$ true. Further information is now needed to provide assurance that e is a logical consequence of Γ, that is, to be able to conclude $\Gamma \models e$.

While for two-valued classical logic only refuting the set of clauses $\Gamma \cup \{\neg\, e\}$ is required, in LPF the case that the goal e denotes a "gap" also needs refuting (the definedness of the goal needs to be established). In other words, if $\Gamma \cup \{\neg\, e\}$ is unsatisfiable then if it can be shown that e is defined in every interpretation that makes Γ true, that is, $\Gamma \models \Delta e$, then it can be inferred that $\Gamma \models e$ holds. Thus to ensure that e is true and defined when entailed by Γ, it is necessary to prove that $\Gamma \models e$ and $\Gamma \models \Delta e$.

Recall that in LPF one reasons from truth to truth, and so if Γ is true then e must be true in the same interpretation ($\Gamma^{\Sigma} \subseteq e^{\Sigma}$). Also recall that the negation of $\perp_{\mathbb{B}}$ is $\perp_{\mathbb{B}}$ in LPF.

An appropriately extended refutation procedure can be used to check the logical consequent $\Gamma \models \Delta e$ (recall the *law of the excluded fourth*). Notice that no circularity is introduced because Δe is guaranteed to always return either true or false.

The use of the meta-level operator Δ will generally not appear in any formula in Γ and in the formula e; it is generally not written in normal assertions. The logical operator Δ is intended to only be used when it is introduced around e in a refutation procedure to refute the "gap" case. If a Δ logical operator does occur in Γ then the following procedure will not break down, but an extra resolution possibility of allowing Δp to be resolved with $\neg\, \Delta p$ will be needed.

The following results formalise the above discussion.

Theorem 33. If $\Gamma \cup \{\neg\, e\}$ is true then $\Gamma \not\models e$.

Proof. By assumption $e_1 \wedge \ldots \wedge e_n \wedge \neg\, e$ can be satisfied and so it must be the case that $(e_1 \wedge \ldots \wedge e_n \wedge \neg\, e)^\sigma$, for some interpretation $\sigma \in \Sigma$. Therefore it follows by the definition of $\mathcal{E}$ that e_1^σ, …, e_n^σ and $\neg\, e^\sigma$ and thus $e^{\overline{\sigma}}$. Thus, there is an interpretation $\sigma \in \Gamma^\Sigma$ that makes the expression e evaluate to false and therefore $\Gamma^\Sigma \nsubseteq e^\Sigma$. By the definition of logical consequence it follows that $\Gamma \not\models e$. □

Lemma 34. If $\Gamma \cup \{\neg\, e\}$ is unsatisfiable then $\Gamma \models e$ may not hold, and thus $\Gamma^\Sigma \not\subset e^\Sigma$.

Proof. Consider a counter example that illustrates that if $\Gamma \cup \{\neg\, e\}$ is unsatisfiable then $\Gamma \models e$ does not hold. Given the logical consequence $\models p \vee \neg\, p$, from which by Lemma 30 it follows that $(p \vee \neg\, p)^\Sigma \subset \Sigma$; but notice that $\Gamma = \{\}$ and so by the definition of logical consequence $\Gamma^\Sigma = \Sigma$. □

Theorem 35. If $\Gamma \cup \{\neg\, e\}$ is unsatisfiable and $\Gamma \models \Delta e$, then $\Gamma \models e$.

Proof. By assumption $\Gamma \cup \{\neg\, e\}$ is unsatisfiable and so it follows that $(\Gamma \cup \{\neg\, e\})^\Sigma = \{\}$. This means that for any interpretation $\sigma \in \Gamma^\Sigma$ either: 1. $\neg\, e^{\overline{\sigma}}$; or 2. $\sigma \notin \mathbf{dom}\,\mathcal{E}(\neg\, e)$. Now, by the assumption $\Gamma \models \Delta e$, it follows that $\Gamma^\Sigma \subseteq (\Delta e)^\Sigma$. Therefore, $(\Delta e)^\sigma$ holds for any interpretation $\sigma \in \Gamma^\Sigma$. Thus by the definition of $\mathcal{E}$ it follows that $\sigma \in \mathbf{dom}\,\mathcal{E}(\neg\, e)$ holds and therefore $\sigma \in \mathbf{dom}\,\mathcal{E}(e)$ holds. Thus only possibility 1 from above can hold for any $\sigma \in \Gamma^\Sigma$, and so by the definition of $\mathcal{E}$ it follows that e^σ. Therefore, $\Gamma^\Sigma \subseteq e^\Sigma$ and so by the definition of logical consequence it follows that $\Gamma \models e$. □

It is clear from Lemma 34 that as well as refuting the false case as in the two-valued classical logic case, in LPF the undefined ("gap") case also needs refuting. If unsatisfiable is returned by applying resolution procedure on $\Gamma \cup \{\neg\, e\}$, then the undefined "gap" case needs refuting (the definedness of the goal needs to be established).

In order to show that $\Gamma \models \Delta e$ holds, one approach is to apply resolution on the set of clauses $\Gamma \cup \{\neg\, \Delta e\}$. If unsatisfiable is returned from this proof when refuting that e is undefined then validity ($\Gamma \models e$) can be concluded according to Theorem 35. If satisfiable is returned from this proof when refuting that e is undefined, then $\Gamma \not\models \Delta e$ and thus $\Gamma \not\models e$ must be concluded by Theorem 33.

The resolution rule can be extended to cope with Δ. The following discussion considers using resolution to refute the possibility of a "gap" for LPF.

An optimisation to the PNF conversion process when considering pushing in a Δ operator is considered first, which is applicable when Δ is introduced around the goal formula of a logical consequent statement mentioned above.

This fact can be used to reduce the size of a predicate formula in clausal form.

Clausal Form Size Reduction

So far, a Δ surrounding a quantifier is replaced with two quantifiers whereby logical equivalence is maintained, and any occurrence of the Δ operator is now inside the quantifiers. When considering only universal quantifiers the following reduces the resulting clausal form size, (what follows does not apply to existential quantifiers).

When trying to show $\Gamma \models e$, the aim has been extended in LPF to show that e is true and defined. If unsatisfiable is returned by resolution on the set of clauses $\Gamma \cup \{\neg\, e\}$, then it can only be the case that e is either true if defined or undefined. Resolution on the set of clauses $\Gamma \cup \{\neg\, \Delta e\}$ would then follow and in this proof it is known that e cannot be false, otherwise satisfiable would have been returned by resolution on the set of clauses $\Gamma \cup \{\neg\, e\}$. This second proof is to show definedness by refuting that it is undefined, and recall that Δ can only return true or false.

Consider that the goal e is $\forall i \cdot p(i)$. While $\Delta(\forall i \cdot p(i))$ is not logically equivalent to $\forall i \cdot \Delta p(i)$ (consider the case that $p(i)$ is true at least once, and undefined at least once and is always true or undefined and thus never false) they are logically equivalent in the restricted case when $\forall i \cdot p(i)$ is not false. The clausal form of $\neg\, \Delta(\forall i \cdot p(i))$ would now be $\{\{\neg\, \Delta p(c)\}\}$ ($\neg\, \Delta(\forall i \cdot p(i))$ is converted to $\neg\, \forall i \cdot \Delta p(i)$, and then to $\exists i \cdot \neg\, \Delta p(i)$, which is then Skolemised). The clauses that arise from the conversion of $\neg\, \Delta e$, where e is a universally quantified formula, are a proper subset of the corresponding set of clauses that arise after applying the logically equivalent conversion rules to $\neg\, \Delta e$.

This is obviously not complete, but such a technique can help to reduce the search space. It is a heuristic. Use as a first attempt, and only use a complete second attempt if the first attempt with the reduced search space is unsuccessful.

Refuting the Possibility of a "Gap"

If unsatisfiable is returned from applying resolution on the set of clauses $\Gamma \cup \{\neg\, e\}$, then $\Gamma \models \Delta e$ needs to be shown to hold in order to conclude $\Gamma \models e$. To show that $\Gamma \models \Delta e$ holds, the first part of the approach taken here is to refute the undefined ("gap") case by performing resolution on the set of clauses $\Gamma \cup \{\neg\, \Delta e\}$. This leads to the need for extra "resolvent" possibilities in LPF. These include allowing resolving on the following two pairs of contradictory literals: p and $\neg\, \Delta p$, and $\neg\, p$ and $\neg\, \Delta p$. The following result shows that the Δ "resolvent" possibilities are sound.

Lemma 36. The literal pairs p and $\neg\,\Delta p$, and $\neg\,p$ and $\neg\,\Delta p$ are contradictory and their simultaneous satisfaction is impossible.
Proof. The goal is to show that $p^{\Sigma}\cap(\neg\,\Delta p)^{\Sigma} = \{\}$ and $(\neg\,p)^{\Sigma}\cap(\neg\,\Delta p)^{\Sigma} = \{\}$. Consider an arbitrary $\sigma \in \Sigma$:

- if $\sigma \in \mathbf{dom}\,\mathcal{E}(p)$ holds (then also $\sigma \in \mathbf{dom}\,\mathcal{E}(\neg\,p)$ holds) then by the definition of $\mathcal{E}$ it follows that $(\Delta p)^{\sigma}$ (and $(\Delta\neg\,p)^{\sigma}$ which is equivalent to $(\Delta p)^{\sigma}$ by Lemma 13); and
- if $\sigma \notin \mathbf{dom}\,\mathcal{E}(p)$ holds then by the definition of $\mathcal{E}$ it follows that $(\Delta p)^{\overline{\sigma}}$ and thus $(\neg\,\Delta p)^{\sigma}$.

Thus these literal pairs are contradictory and therefore no $\sigma \in \Sigma$ can simultaneously satisfy both p and $\neg\,\Delta p$, nor both $\neg\,p$ and $\neg\,\Delta p$. $\square$

The use of these "extra" resolvent possibilities provides a way of refuting the set of clauses $\Gamma \cup \{\neg\,\Delta e\}$. Reducing the number of circumstances to where Δ needs introducing around the goal e is discussed later.

Theorem 35 establishes the need to show that the goal is defined. When attempting to refute the possibility of a "gap" in the goal, it can be shown that just refuting the clausal form of $\Gamma \cup \{\neg\,\Delta e\}$ is not enough on its own to establish the definedness of a goal, when considering predicate clauses and the resolution refutation procedure. Undefinedness can still arise due to the presence of partial terms in substitutions, so here in unifiers. Thus further action must be taken to establish the definedness of the goal, and this is considered next; unification constraints need including in a resolvent clause because of the possibility of undefined terms.

Reconsidering the Unrestricted Use of Unification

Goal based terms can be unified with assumption based terms in a resolution step. Unifying can lead to substituting variables which range only over defined values for terms that can be undefined in particular interpretations. The occurrence of possible partial terms from goal clauses arising in unifiers needs guarding against. This is done by carefully guarding the unification used in an application of the resolution rule, to ensure that a defined *Var* from the left hand assumption side Γ is not unified without restriction with a possible partial term from the right hand goal side e. The following example illustrates this, (recall that in $\mathcal{E}$ all functions and predicates are strict):

$$\forall x \cdot x = x \models 5/0 = 5/0$$

Performing a standard resolution refutation proof with unification on the clausal form:

$$\{\{x = x\}, \{\neg\,(5/0 = 5/0)\}\}$$

of this logical consequence statement leads to the empty clause, since x unifies with the function $5/0$.

Surrounding the goal with Δ and performing a new resolution refutation proof to refute the presence of a "gap" this time on the clausal form:

$$\{\{x = x\}, \{\neg\,\Delta(5/0 = 5/0)\}\}$$

again leads to the empty clause being inferred.

But clearly this formula is not valid in LPF. As a counter example consider $=$ being interpreted as weak equality, and the standard partial division operator. A defined term x from the assumption side has been unified with a term $5/0$ (a function application) that is undefined from the goal side of the logical consequent. Recall that in LPF one only reasons from truth to truth, and that the term x must be defined because it is a quantified variable and quantification can only be over a set of proper (i.e. defined) values in LPF. It thus must follow that the application of unification within a resolution step in LPF, (in a resolution refutation proof), due to the presence of "gaps" needs guarding in certain circumstances.

The approach taken is for constraint(s) to be included as literal(s) (disjuncts) in an inferred resolvent within a resolution refutation proof. These constraints effectively take the form of further definedness obligations, but this time using the δ definedness operator. A resolvent inferred by resolving on the clauses C_1 and C_2 where $\{l_1\} \subseteq C_1$ and $\{\neg\, l_2\} \subseteq C_2$ is defined to be:

$$(\phi[C_1] \setminus \phi[\{l_1\}]) \cup (\phi[C_2] \setminus \phi[\{\neg\, l_2\}]) \cup \theta$$

where l_1 and l_2 unify with an mgu ϕ, and where θ is a set of unification constraint(s), where each $\psi_i \in \theta$ is a literal. This form of the resolvent is needed whenever a clause from the right (*goal*) side (containing the potentially partial term) is resolved (and thus unified) with a clause from the left (*assumption*) side of the logical consequent, and whenever a goal clause is resolved with a goal clause. Any resolvent that is inferred when at least one of the two clauses resolved on is a goal clause is deemed to be a goal clause for the purposes of introducing δ unification constraints.

The unification constraints θ can be built up by considering an mgu ϕ, where given $\phi = \{x_1 \mapsto \alpha_1, \ldots, x_n \mapsto \alpha_n\}$, then $\theta = \{\neg\,\delta\alpha_1, \ldots, \neg\,\delta\alpha_n\}$, where each $\psi_i \in \theta$ is a literal (a disjunct added to the resolvent). For instance, if $\phi = \{x \mapsto f(\ldots)\}$, where $x \in \mathit{Var}$ and $f \in \mathit{Fn}$, then $\theta = \{\neg\,\delta f(\ldots)\}$. The unification constraints need introducing to ensure that only valid formulae can actually be proven valid.

In $\mathcal{E}$, the δ logical operator was only provided for formulae. Thus in $\mathcal{E}$, now the δ logical operator needs overloading so that it can also be applied to terms. If the term α is defined then $\delta\alpha$ is to return true:

$$\{(\sigma, \mathbf{true}) \mid \sigma \in \mathbf{dom}\,\mathcal{E}(\alpha)\}$$

This is the same definition for the δ logical operator with a term operand, as was given to the δ logical operator when it was defined with a formulae operand, in the $\mathcal{E}$ semantic function definition.

Recall that the δ operator is monotone, and is true if the (integer here) operand α is defined, otherwise it is undefined. This treatment of adding unification constraints into the resolvent for every maplet in ϕ can, however, be improved upon.

First consider that the function identifiers *Fn* can be seen as a shorthand for $\mathit{Fn} = \mathit{SkolemFun} \mid \mathit{Fun}$, where the identifier names in *SkolemFun* and *Fun* are disjoint. This split is illustrated explicitly for the purposes of including the unification constraints because a Skolem function is total, but a *Function* mapped to by any *Fun* can be a partial function, and thus an application of any $f \in \mathit{Fun}$ can be undefined (a term that applies a partial function can fail to denote a proper defined value).

Therefore in certain circumstances the use of unification requires no additional constraints to be included into a resolvent, for instance, when unifying x with y when $x \in \mathit{Var}$ and $y \in \mathit{Var}$, and when unifying x with $f(\ldots)$ when $x \in \mathit{Var}$ and $f \in \mathit{SkolemFun}$. However, when unifying x with $f(\ldots)$ when $x \in \mathit{Var}$ and $f \in \mathit{Fun}$ then a unification constraint (a definedness obligation) must be introduced into any inferred resolvent arising from such a resolution step. Notice that a predicate cannot be unified with any variable, as only integer and propositional variables are present, in the $\mathcal{E}$ semantic function definition.

The reason behind including the unification constraints in a resolvent is that in $\mathcal{E}$, for any $f \in \mathit{Fun}$, it can only be known that $f(\ldots) \in \mathbb{Z}_\perp$ (either it is defined and a member of $\mathbb{Z}$, or it is undefined). If this term is unified with any $x \in \mathit{Var}$, when it is known that all integer variables (Var) are defined in $\mathcal{E}$ (all quantification is over defined values in LPF), then unification within a

$$\begin{array}{r|l|l}
1 & p \vee \neg\, p & \mathit{goal} \\
2 & \{\neg\, p\} & \multirow{2}{*}{$\}\mathit{deny}(\mathit{clausal\_form}(1))$} \\
3 & \{p\} & \\
4 & \square & \mathit{resolve}(2,3) \\
5 & \Delta(p \vee \neg\, p) & \mathit{goal} \\
6 & \{\neg\, \Delta p\} & \mathit{deny}(\mathit{clausal\_form}(5)) \\
7 & - &
\end{array}$$

Figure 6.1: An illustrative resolution refutation attempt

resolution step that allows for something that is guaranteed to be defined (in an assumption clause) to be unified and thus resolved with something that *can* be undefined from the goal, violates a condition that needs to hold in order for the result indicated in Theorem 35 to follow.

A unification constraint can be removed (only if it is known to be defined) by a further resolution step. In order to do this a literal of the form $\delta\alpha$ can be included as a positive literal on the assumption side Γ of a logical consequent statement, to state that a function (term) α is defined. This will all be illustrated in the examples that follow in this chapter.

The use of factoring in a refutation proof also needs protecting through the inclusion of unification constraints.

Illustrative Examples

Consider again the earlier counter example of $\models p \vee \neg\, p$ where the empty clause (unsatisfiability) is infered. Therefore in LPF $\models \Delta(p \vee \neg\, p)$ needs to be shown to hold to be able to infer that $\models p \vee \neg\, p$ holds. In the modified LPF clausal form the negation $\neg\, \Delta(p \vee \neg\, p)$ is represented as $\{\{\neg\, \Delta p\}\}$ after simplification, which cannot be refuted ($\{\{\neg\, \Delta p\}\}$ is satisfiable). This is presented in Figure 6.1, where the $-$ on line number 7 denotes that no more rule applications apply.

As a further example reconsider Property 1.5:

$$\forall i\colon \mathbb{Z} \cdot (i/i = 1) \vee ((i-1)/(i-1) = 1)$$

In order to prove this property several assumptions (properties) of division and subtraction are introduced, because the resolution is syntactic, and thus the semantics of the functions $-$ and $/$ cannot be used:

$$\forall i\colon \mathbb{Z} \cdot i = 0 \;\Rightarrow\; \neg\,((i-1) = 0);\;\; \forall i\colon \mathbb{Z} \cdot \neg\,(i = 0) \;\Rightarrow\; i/i = 1 \;\vdash \\ \forall i\colon \mathbb{Z} \cdot (i/i = 1) \vee ((i-1)/(i-1) = 1)$$

The two function symbols $-$ and $/$, as well as the predicate symbol $=$ used are just to be interpreted as arbitrary functions and an arbitrary predicate respectively. Their meaning needs to be constrained by including assumptions.

1	$\forall i{:}\mathbb{Z} \cdot i = 0 \;\Rightarrow\; \neg\,((i-1) = 0)$	*assumption*
2	$\forall i{:}\mathbb{Z} \cdot \neg\,(i = 0) \;\Rightarrow\; (i/i = 1)$	*assumption*
3	$\forall i{:}\mathbb{Z} \cdot (i \div i = 1) \vee ((i-1)/(i-1) = 1)$	*goal*
4	$\{\neg\,(i = 0), \neg\,((i-1) = 0)\}$	$\mathit{clausal\_form}(1)$
5	$\{i = 0, (i/i = 1)\}$	$\mathit{clausal\_form}(2)$
6	$\{\neg\,(c/c = 1)\}$	$\}\mathit{deny}(\mathit{clausal\_form}(3))$
7	$\{\neg\,((c-1)/(c-1) = 1)\}$	
8	$\{c = 0\}$	$\mathit{resolve}(5,6)$
9	$\{(c-1) = 0\}$	$\mathit{resolve}(5,7)$
10	$\{\neg\,((c-1) = 0)\}$	$\mathit{resolve}(4,8)$
11	$\square$	$\mathit{resolve}(9,10)$

Figure 6.2: An illustrative resolution refutation subproof (part 1)

1	$\forall i{:}\mathbb{Z} \cdot i = 0 \;\Rightarrow\; \neg\,((i-1) = 0)$	*assumption*
2	$\forall i{:}\mathbb{Z} \cdot \neg\,(i = 0) \;\Rightarrow\; (i/i = 1)$	*assumption*
3	$\forall i{:}\mathbb{Z} \cdot \delta(i-1)$	*assumption*
4	$\Delta(\forall i{:}\mathbb{Z} \cdot (i/i = 1) \vee ((i-1)/(i-1) = 1))$	*goal*
5	$\{\neg\,(i = 0), \neg\,((i-1) = 0)\}$	$\mathit{clausal\_form}(1)$
6	$\{i = 0, (i/i = 1)\}$	$\mathit{clausal\_form}(2)$
7	$\{\delta(i-1)\}$	$\mathit{clausal\_form}(3)$
8	$\{\neg\,(c/c = 1), \neg\,\Delta(c/c = 1)\}$	
9	$\{\neg\,((c-1)/(c-1) = 1), \neg\,\Delta((c-1)/(c-1) = 1)\}$	$\}\mathit{deny}(\mathit{clausal\_form}(4))$
10	$\{\neg\,\Delta(c/c = 1), \neg\,\Delta((c-1)/(c-1) = 1)\}$	
11	$\{c = 0, \neg\,\Delta(c/c = 1)\}$	$\mathit{resolve}(6,8)$
12	$\{c = 0\}$	$\mathit{resolve}(6,11)$
13	$\{(c-1) = 0, \neg\,\Delta((c-1)/(c-1) = 1), \neg\,\delta(c-1)\}$	$\mathit{resolve}(6,9)$
14	$\{(c-1) = 0, \neg\,\delta(c-1)\}$	$\mathit{resolve}(6,13)$
15	$\{\neg\,((c-1) = 0)\}$	$\mathit{resolve}(5,12)$
16	$\{\neg\,\delta(c-1)\}$	$\mathit{resolve}(14,15)$
17	$\square$	$\mathit{resolve}(7,16)$

Figure 6.3: An illustrative resolution refutation proof establishing definedness (part 2)

An example resolution refutation subproof of Property 1.5 is presented in Figure 6.2, where c in this proof is a Skolem constant. Figure 6.3 presents the same proof but which also establishes the definedness of the goal. An additional assumption is provided in the latter proof. This is because a unification constraint/a definedness obligation is introduced, and the additional assumption is needed to be able to discharge the unification constraint that is introduced into the proof. The unification constraint is introduced to ensure that the $-$ function is total. The additional assumption states only that $-$ is total, which ensures that only those interpretations where $-$ is total are considered.

As can be seen from the two proofs the second proof with Δ has a larger clausal form. Additionally, a greater number of resolvents are inferred, and the size of the search space as expected also increases.

For the purposes of illustrating the issues surrounding the mechanisation of the resolution refutation procedure in LPF the proofs have been presented separately up until now. They can be combined by taking the clausal form of $(\neg\, e)^{CNF} \vee (\neg\, \Delta e)^{CNF}$. The goal e is negated and put into CNF. The goal with the occurrence of the Δ logical operator, $\neg\, \Delta e$ is put into CNF. By CNF in these two instances it is meant that a propositional formula is put into CNF, and for a predicate formula that the matrix is put into CNF, after going through the PNF conversions, and then going through the Skolem form conversions, and dropping any remaining universal quantifiers. The disjunction of these two formulae that are already in CNF are then taken, and should be put into clausal form. This ensures that if necessary that a distributivity rule application occurs when putting this disjunction into clausal form. The unification constraints/definedness obligations will need considering also in this proof.

An optimisation to what has so far been presented is considered next, which is concerned with limiting the introduction of the expensive Δ logical operator into proofs.

Optimisation

This optimisation considers reducing the number of cases in which Δ needs to be introduced around the goal. This is important because any use of a Δ logical operator leads to a large clausal form representation of the goal, and if cases can be identified whereby a Δ does not need wrapping around the goal then the size of the search space can be reduced. This optimisation does not concern reducing the number of occasions when a unification constraint δ needs introducing. Unification constraints still need to be introduced whenever one of the circumstances mentioned earlier in this section arises.

By the definition of logical consequence, it follows that $\Gamma^{\Sigma} \subseteq e^{\Sigma}$ (recall the LPF SS-sequent interpretation), and thus resolving anything from the *goal* side in resolution, with anything from the *assumption* side of the logical consequent, is safe. This follows from the fact that only those $\sigma \in \Gamma^{\Sigma}$ are of interest in LPF (in LPF one is only concerned with reasoning from truth to truth). If the assumptions of a logical consequence statement are false or undefined then there is no constraint on the goal, that is, that the goal can be true, false or undefined.

Thus the extent to which Δ needs to be introduced around the goal can be limited. If a goal clause is resolved with a goal clause then Δ needs introducing

1	$\forall i{:}\,\mathbb{Z} \cdot i = 0 \;\Rightarrow\; \neg\,((i-1) = 0)$	*assumption*
2	$\forall i{:}\,\mathbb{Z} \cdot \neg\,(i = 0) \;\Rightarrow\; (i/i = 1)$	*assumption*
3	$\forall i{:}\,\mathbb{Z} \cdot \delta(i-1)$	*assumption*
4	$\forall i{:}\,\mathbb{Z} \cdot (i/i = 1) \vee ((i-1)/(i-1) = 1)$	*goal*
5	$\{\neg\,(i = 0), \neg\,((i-1) = 0)\}$	$\mathit{clausal\_form}(1)$
6	$\{i = 0, (i/i = 1)\}$	$\mathit{clausal\_form}(2)$
7	$\{\delta(i-1)\}$	$\mathit{clausal\_form}(3)$
8	$\{\neg\,(c/c = 1)\}$	$\}\mathit{deny}(\mathit{clausal\_form}(4))$
9	$\{\neg\,((c-1)/(c-1) = 1)\}$	
10	$\{c = 0\}$	$\mathit{resolve}(6, 8)$
11	$\{(c-1) = 0, \neg\,\delta(c-1)\}$	$\mathit{resolve}(6, 9)$
12	$\{\neg\,((c-1) = 0)\}$	$\mathit{resolve}(5, 10)$
13	$\{\neg\,\delta(c-1)\}$	$\mathit{resolve}(11, 12)$
14	$\Box$	$\mathit{resolve}(7, 13)$

Figure 6.4: An illustrative (optimised) resolution refutation proof establishing definedness

around the goal, to ensure that a "gap" is not inferred (resolving a goal side clause with a goal side clause of the logical consequent, as already illustrated causes a problem in LPF, and requires the necessary introduction of Δ around the goal). Definedness obligations (the unification constraints) arising from the use of unification are needed regardless. If a goal clause is always resolved with an assumption clause then only the unification constraints are needed, so a Δ will not need introducing around the goal. A resolvent formed by resolving a goal clause with an assumption clause is deemed to be an assumption clause for the purposes of introducing the Δ logical operator around the goal. Since the clausal form for Δ can be expensive then this is a significant improvement, since the size of the search space gets reduced.

If the goal does not need surrounding with Δ then the unification constraints (which are still needed) could be taken into consideration in the first proof on the set of clauses $\Gamma \cup \{\neg\, e\}$, as illustrated in the proof in Figure 6.4. As can be seen in the proof in Figure 6.4 fewer resolvents are needed in comparison to the proof of the same formula that was presented in Figure 6.3.

Starting the proof without surrounding the goal with Δ may be the best starting point (to reduce the size of the goal formula and thus to reduce the search space), and thus restricting goal clause on goal clause resolving. Therefore limiting oneself to attempting to prove the goal from the assumptions that are generally assumed to be true.

6.7 Conclusions

This chapter has presented an investigation into the applicability of mechanised proof support for LPF, by focusing on the basic but fundamental two-valued classical logic proof procedure of resolution and the associated technique of proof by contradiction. The work has highlighted the issues that arise when applying these techniques to LPF, as well as having proposed modifications to these techniques to cover LPF.

Clearly, when mechanising these techniques in LPF more work was going to be needed than in a total framework of two-valued classical logic due to the possible occurrence of partial terms. The extra work takes the form of having to show the definedness of terms and formulae. In respect to the amount of work needed briefly re-consider the method of truth tables where for a propositional formula in LPF 3^n rows in the truth table will be needed as opposed to 2^n rows in the truth table that are needed in two-valued classical logic. Determining the extent of when definedness obligations need introducing to these techniques for LPF has been undertaken. In the small examples considered not too many more extra resolvents needed to be made in the LPF proofs with partial terms present, in comparison to comparable proofs in two-valued classical logic. Further comparisons would of course be beneficial this is discussed in Section 7.2.2. This work has provided key insights into providing mechanised proof support for a non-classical logic like LPF.

Advanced proof techniques have been developed that are built on the fundamental basic proof techniques considered here, see Section 7.2.4. An investigation into these advanced proof techniques has not been pursued here, as the essential and obvious starting point was to investigate the fundamental basic proof techniques. This investigation which has given rise to modifications to the basic techniques to cover LPF provides the essential foundation to facilitate research into the modification of advanced proof techniques and tool support for LPF.

The $\mathcal{E}$ semantic function definition that was presented in Figure 3.8 that formally captures LPF has been used throughout this chapter, to be able to precisely highlight the issues that arise when applying the basic selected proof techniques in LPF. Furthermore, proofs about resolution in LPF and of associated techniques such as CNF and clausal form conversions etc. have been based upon this $\mathcal{E}$ semantic function definition. The $\mathcal{E}$ semantic function definition precisely and succinctly captured the core of LPF, ensuring that proofs could be conducted and concepts and issues illustrated with respect to only a small but core definition that can be easily understood. Illustrative proofs of key examples using the proposed modifications to the techniques considered here have also been presented, which in certain cases capture the benefit of the use of the optimisations that have been proposed in building up the modifications to the techniques to cover LPF.

Due to the use of resolution, a topic that needed considering was the conversion of formulae into clausal form. Properties such as the commutativity, and the distributivity of disjunctions and conjunctions are retained in LPF, as well as other well known two-valued classical logic clausal form conversions, such as Skolemisation. Because of the Δ definedness logical operator in LPF though, extra conversions are needed to be able to convert an LPF formula into clausal form. The conversions that are required for Δ to maintain logical equivalence are expensive in regards to the size of the resulting clausal form. This is because a Δ needs pushing inwards so that it only surround literals. It has been shown how to limit the introduction of Δ's however.

A key topic that was addressed in the investigation was to consider basic definitions like validity and unsatisfiability in LPF. The definition of unsatisfiability in LPF must take into account undefinedness ("gaps"). This impacts the duality between validity and unsatisfiability that is key to certain results in two-valued classical logic; this impacts refutation procedures. In LPF when considering the resolution refutation procedure the definedness of the goal is forced to be shown, leading to the introduction of δ and Δ definedness logical operators, to ensure that only valid formulae can actually be proven valid. It has been shown how definedness obligations can be refuted using resolution.

The resolution rule of inferences carries over to LPF. It is the use of the resolution refutation procedure, that forces the definedness of the goal to be established. It is key to the results proposed here that the definedness of the assumptions does not need to be established; recall the LPF *SS* sequent interpretation.

There exists two key areas when definedness obligations must be introduced into resolution refutation proofs to ensure that the definedness of the goal is established: when goal clause on goal clause resolving can take place, and to guard against possible partial terms in substitutions (here unifiers).

Due to the expense of using the Δ definedness operator, it was vital to show how to limit the introduction of Δ into proofs. Only when resolving a goal clause with a goal clause does a Δ need introducing around the goal, to ensure that only valid formulae can actually be proven valid. If such goal clause on goal clause resolving is forbidden in LPF then the introduction of the Δ logical operator around the goal is not necessary.

One of the biggest issues that needs resolving in using the resolution refutation procedure in LPF is the possible occurrence of partial terms in unifiers. The use of unification needs to be carefully guarded, to ensure the definedness of the goal being proved. Definedness obligations need introducing based upon the occurrence of functions in a unifier. When unifying an integer variable with an integer variable no such definedness obligation needs introducing. Because an integer variable is a quantified variable, and in LPF quantification only ever occurs over defined sets of values. Definedness obligations, however, do need introducing in certain cases when unifying an integer variable with a non-Skolem function (it has been identified that Skolem functions are total). Because in LPF it is the case that partial functions can arise, and terms that apply partial functions can fail to denote a proper value. To cope with such partial terms in unifiers definedness obligations need introducing into a resolvent. Here the δ definedness logical operator can be used, which here will only need to surround terms; the clausal form size is the same.

An alternative technique to using LPF is that of using the Well-Definedness (WD) approach. Such a technique also introduces extra work, that being a WD proof. Expensive WD conditions (called $\mathcal{D}$ in some literature, e.g. [Meh08]) need discharging to remove the concern of undefined expressions from validity proofs. The use of Δ in LPF is closely related and is also expensive. In this work a goal formula needs surrounding with Δ, only if a goal clause is allowed to be resolved with a goal clause. Here Δ is not needed around any assumption formula. A function in the WD approach to coping with partial terms requires showing every argument is well-defined, and that the domain restriction predicates hold.

In [KK94] a mechanisation of Kleene logic for partial functions is presented. Kleene's logic is formalised in an order-sorted three-valued logic and a resolution calculus is presented. Their work leads to a more expensive clausal form, and thus an increased search space than what is required for the method that has been proposed here. A thorough investigation of where undefinedness arises has been presented here, and this has led to a reduction in the number of definedness obligations that need introducing, and thus have to be discharged.

The example proof that is used in [KK94] is:

$$\forall i{:}\,\mathbb{R} \cdot \forall j{:}\,\mathbb{R} \cdot \neg\,(i = j) \;\Rightarrow\; (((1/(i-j))^2) > 0)$$

and a proof of this property using the proposed method from this chapter is presented in Figure 6.5. The example has been changed slightly to avoid using both the $\mathbb{R}$ datatype and the $\mathbb{R}_1$ datatype, that is, the set of real numbers without the number 0. This has forced the introduction of a disjunct $\neg\,(i = 0)$ into assumption at line reference number 3, and into the assumption at line reference number 4 in Figure 6.5. This proof is completed with fewer resolvents needing to be inferred than in [KK94]. A smaller number of resolvents on such a small example is encouraging for tackling larger examples in the future, see

1	$\forall i\colon \mathbb{R} \cdot \forall j\colon \mathbb{R} \cdot \neg\,(i = j) \;\Rightarrow\; \neg\,(1/(i-j) = 0)$	*assumption*
2	$\forall i\colon \mathbb{R} \cdot \forall j\colon \mathbb{R} \cdot \delta(i-j)$	*assumption*
3	$\forall i\colon \mathbb{R} \cdot \neg\,(i = 0) \;\Rightarrow\; \delta(1/i)$	*assumption*
4	$\forall i\colon \mathbb{R} \cdot \neg\,(i = 0) \;\Rightarrow\; i^2 > 0$	*assumption*
5	$\forall i\colon \mathbb{R} \cdot \forall j\colon \mathbb{R} \cdot \neg\,(i = j) \;\Rightarrow\; \neg\,(i-j = 0)$	*assumption*
6	$\forall i\colon \mathbb{R} \cdot \forall j\colon \mathbb{R} \cdot \neg\,(i = j) \;\Rightarrow\; (((1/(i-j))^2) > 0)$	*goal*
7	$\{i = j, \neg\,(1/(i-j) = 0)\}$	$\mathit{clausal\_form}(1)$
8	$\{\delta(i-j)\}$	$\mathit{clausal\_form}(2)$
9	$\{i = 0, \delta(1/i)\}$	$\mathit{clausal\_form}(3)$
10	$\{i = 0, i^2 > 0\}$	$\mathit{clausal\_form}(4)$
11	$\{i = j, \neg\,(i-j = 0)\}$	$\mathit{clausal\_form}(5)$
12	$\{\neg\,(c = d)\}$	$\}\mathit{deny}(\mathit{clausal\_form}(6))$
13	$\{\neg\,(((1/(c-d))^2) > 0)\}$	
14	$\{(1/(c-d)) = 0, \neg\,\delta(1/(c-d))\}$	$\mathit{resolve}(10, 13)$
15	$\{(1/(c-d)) = 0, c-d = 0, \neg\,\delta(c-d)\}$	$\mathit{resolve}(9, 14)$
16	$\{(1/(c-d)) = 0, c-d = 0\}$	$\mathit{resolve}(8, 15)$
17	$\{c = d, c-d = 0\}$	$\mathit{resolve}(7, 16)$
18	$\{c = d\}$	$\mathit{resolve}(11, 17)$
19	$\square$	$\mathit{resolve}(12, 18)$

Figure 6.5: An illustrative resolution refutation proof establishing definedness

Section 7.2.2.

Furthermore, consider the proof of Property 1.3, that is presented in Figure 6.6. Like in the LPF natural deduction style version of this proof that was presented in Figure 2.13, definedness obligations do not play a part in this resolution refutation proof using the method proposed here, and the proof proceeds as it would do in two-valued classical logic.

1	$\forall i\colon \mathbb{Z} \cdot i \geq 0 \;\Rightarrow\; \mathit{zero}(i) = 0$	*assumption*
2	$\forall i\colon \mathbb{Z} \cdot i \geq 0 \vee -i \geq 0$	*assumption*
3	$\forall i\colon \mathbb{Z} \cdot \mathit{zero}(i) = 0 \vee \mathit{zero}(-i) = 0$	*goal*
4	$\{\neg\,(i \geq 0), \mathit{zero}(i) = 0\}$	$\mathit{clausal\_form}(1)$
5	$\{i \geq 0, -i \geq 0\}$	$\mathit{clausal\_form}(2)$
6	$\{\neg\,(\mathit{zero}(c) = 0)\}$	$\}\mathit{deny}(\mathit{clausal\_form}(3))$
7	$\{\neg\,(\mathit{zero}(-c) = 0)\}$	
8	$\{\neg\,(c \geq 0)\}$	$\mathit{resolve}(4, 6)$
9	$\{-c \geq 0\}$	$\mathit{resolve}(5, 8)$
10	$\{\mathit{zero}(-c) = 0\}$	$\mathit{resolve}(4, 9)$
11	$\square$	$\mathit{resolve}(7, 10)$

Figure 6.6: An illustrative resolution refutation proof where definedness obligations do not need introducing

Chapter 7

Concluding Remarks

Contents

7.1 Summary and Conclusions 220

7.2 Future Work 224

7.2.1 Further Comparison Results 225

7.2.2 Further Performance Results 226

7.2.3 Implementing the Proof Techniques Considered . . . 226

7.2.4 Investigating the Applicability of Further Proof Techniques for LPF 227

First a summary of this thesis is provided, highlighting the contributions that have been made. Key points of how the work that is presented in this thesis can be extended in the future are then discussed.

7.1 Summary and Conclusions

Partial functions arise frequently when reasoning about programs, and a term that applies a partial function can fail to denote a proper value (a partial term). Partial terms can occur in logical formulae, and reasoning about such logical formulae that can contain references to partial terms is problematic in two-valued classical logic. Undefinedness from terms can propagate up leading to formulae that can fail to denote, which makes no sense in two-valued classical logic, since the truth tables only define the logical operators for proper Boolean values. In this work as opposed to using concrete undefined values, the term “gap” is used, that is, the absence of a defined value, for example, a truth value “gap”.

Numerous approaches have been proposed over the years to cope with partial terms. Some of these attempt to stay within the realm of two-valued classical logic, by ensuring that undefinedness cannot be propagated out to the logical operators, so that the use of the two-valued classical logic logical operators can be maintained. Other approaches are based on non-classical logics. LPF is a non-classical (three-valued) logic, based upon Strong Kleene logic, where the interpretations of the logical operators are extended to cope with undefinedness, truth value "gaps".

An obstacle to the use of a non-classical logic like LPF for reasoning about logical formulae that can contain references to partial terms is that a large body of research and engineering has gone into two-valued classical logic over the years. This has led to a wide range of mechanised (interactive and automated) proof based tool support, and proof procedures for two-valued classical logic, which cannot be re-used without change for LPF due to the presence of partial functions, leading to the necessary establishment of the definedness of terms and formulae. There is a lack of direct proof support available for LPF. An aim of this work was to investigate the applicability of mechanised proof support for reasoning in LPF. How this aim was addressed is summarised below.

Before this investigation could be tackled it was key to develop a semantic foundation of LPF, to facilitate the investigation. This foundation also gave rise to a method by which to formally compare and to investigate different approaches to coping with partial terms. The semantic foundation has been at the core of the rest of the work presented.

Two semantic definitions have been presented which formally capture LPF. Both Structural Operational Semantic (SOS) definitions (a big-step definition and a small-step definition), and denotational semantic (DS) definitions were defined (the SOS definitions preceded the DS definitions). The SOS definitions focus on how expressions are evaluated not just what the final results are, while the DS definitions provide a more concise definition of the values that are denoted by expressions [NN92]. Related proofs of the semantic definitions have been presented which show relationships between the semantic definitions, for example how the DS definitions coincide, and how the small-step SOS definition and a DS definition coincide. The definitions provide clear and precise descriptions of LPF, allowing one to be clear about the semantics of LPF before starting with a mechanisation of LPF. Furthermore, they provide a means to precisely describe concepts and illustrate issues, they provide a basis on which to conduct proofs of modifications to proof procedures for LPF, and they provide a basis on which to conduct formal comparisons between approaches to coping with partial terms. The semantic definitions for LPF are the foundation which underlies the rest of the work presented.

An LPF DS definition has been modified to formally define the semantics for other approaches to coping with logical formulae that can contain references to partial terms. The differences between the approaches can be seen clearly by noting the changes that are made between the definitions, since in most cases only small changes needed making between the definitions. Such definitions provided a way of formally comparing the different approaches to coping with partial terms, and they have also been used to illustrate how theorems can be moved between the different approaches to coping with partial terms. Being able to move theorems between different proof tools/formal methods relies on identifying the differences between the approaches so that they can be used together, and this work has focused on overcoming any mismatches in respect to the different treatments of coping with partial terms. The DS definitions have been proposed as a way of formally comparing the different approaches to coping with partial terms. The use of DS definitions in effect precisely and succinctly capture the crucial points and the differences between the approaches, facilitating a formal comparison between them.

The SOS definitions have been used as a basis of developing mechanisations of LPF in tool support environments. One of the mechanisations utilises the Maude term-rewriting system, allowing for expressions to be evaluated by a tool according to the semantics of LPF. Both the big-step SOS definition, and the small-step SOS definition were coded into Maude. This also allowed for precise comparisons to be made between the two definitions.

Another mechanisation provided the foundation of some interactive proof support for LPF in the Isabelle proof assistant. This was the first attempt at providing proof support for LPF in this work. The work on coding the big-step SOS definition into Isabelle (defining the disjunction logical operator, and allowing for function applications etc.), form the key foundation on which to facilitate further development of interactive proof support for LPF. Only a small set of expression constructs were defined in Isabelle, but enough to ensure that fundamental properties involving applications of partial functions could be proved. It would be useful to extend this mechanisation further to include support for additional expression constructs and datatypes etc.

An investigation into the applicability of mechanised proof support for LPF, focused on the basic but fundamental two-valued classical logic proof procedure: resolution and the associated technique of proof by contradiction. Advanced proof techniques (see Section 7.2.4), are built on the foundation provided by these basic but fundamental proof techniques. Thus an investigation into the basic techniques was the essential and obvious starting point for addressing the issue of mechanised proof support for LPF. The work provides key insights into the provision of mechanised proof support for a non-classical logic like LPF, and provides the essential foundation on which to facilitate research into the modification of advanced proof techniques for LPF, and for providing tool support in the future. The work was to argue that when supplemented with modifications that the key fundamental basic proof techniques can be re-used to conduct reasoning within LPF, and furthermore that they can be modified efficiently for LPF.

The investigation highlighted the issues that arise when applying the resolution proof procedure and the associated technique of proof by contradiction in LPF, and determined the extent of the modifications needed to adapt them for LPF. Outcomes of this work were thus an insight into the amount of extra work that is brought into these techniques when they are applied in LPF, as well as the modification of the techniques for LPF.

An LPF DS definition was used as a basis to highlight the issues that arise, define concepts, and it was also used as the basis on which to prove the modifications made to the resolution proof procedure to cover LPF. The use of such a DS definition here aided greatly in being able to illustrate the issues precisely, as well as ensuring that the proofs could be performed by relying only on a relatively small but core underlying basis.

The issue of undefinedness is present even in the definition of basic concepts. For instance, the definition of unsatisfiability must take into account undefinedness, due to the presence of partial functions and thus partial terms. This impacts the well-known duality between validity and unsatisfiability. Since undefinedness issues present themselves at such a low level, addressing the core of the classical fundamental basic proof procedure resolution (with refutation) was necessitated, which obviously needed doing before looking at any advanced proof techniques built around them over the years.

The resolution rule of inference carries over to LPF. But, the resolution refutation procedure does not carry over to LPF. This is an impact of the loss of duality between validity and unsatisfiability. The definedness of the goal must be shown in addition in LPF. The resolution refutation procedure needs adapting with definedness obligations which need discharging to ensure that only valid formulae can actually be proven valid. Assessing the extent of the required introduction of definedness obligations for a resolution refutation proof has been investigated. This has already been summarised in more detail in the conclusion section of Chapter 6, so such detail will not be reiterated here.

Pleasingly, the two-valued classical logic clausal form conversion techniques considered carry over to LPF, but need supplementing with conversions for a definedness logical operator.

The definedness obligations introduced into resolution proofs in LPF use the Δ and δ definedness logical operators. Due to the expense of using the non-monotonic Δ definedness logical operator it has been investigated how to limit its introduction into resolution proofs in LPF. Its use can be constrained to only being needed when a goal clause can be resolved with a goal clause. The occurrence of partial terms in substitutions through the use of unification in resolution requires the introduction of definedness obligations, but these can be introduced using the δ definedness logical operator; this does not carry the expense of the Δ definedness logical operator in the conversion to clausal form, since it is only applied to terms.

Pleasingly it has been shown that the basis of the considered proof techniques in two-valued classical logic can be re-used when supplemented with vital modifications for LPF. The modifications relate to showing the definedness of terms and formulae. This ensures that an implementation of them can be built up from modifying existing code bases, and existing tool support can be adapted instead of having to start from scratch, for instance. Definedness obligations can be proved using resolution itself.

It is pleasing that, on the small examples tested, the LPF resolution refutation procedure (with partial terms being able to arise) does not need many more resolvents to be inferred in the proof, in comparison to the comparable proofs conducted in two-valued classical logic, but coping with partial terms in two-valued classical logic is problematic. In a specific example of Property 1.3 considered the proof was exactly the same as a comparable proof in two-valued classical logic. However, what is missing is results on larger examples to get further performance results to be able to evaluate the performance of the proof procedures considered for LPF in more depth. This issue is discussed further in Section 7.2.2.

The mechanisation work for LPF has started to address a major criticism that was put towards LPF, which was a lack of proof support for LPF. This mechanisation work has also greatly aided in the case of justifying the use of LPF for reasoning about logical formulae that can contain reference to partial terms.

There are several areas where this work can be extended; these are outlined further in the next section.

7.2 Future Work

As usual there is more work that could be done. The key areas identified for the extension of this work are:

- a further comparison of different non-classical logic approaches to coping with partial terms based upon implementing proof procedures in them as has been done for LPF (see Section 7.2.1);

- gaining further results on the performance/efficiency of the proof techniques considered in LPF by undertaking case studies (see Section 7.2.2);

- providing a concrete implementation of the proof techniques considered for LPF (see Section 7.2.3); and
- extending the work on LPF to cope with equality, and considering other advanced proof techniques that have been built around the basic fundamental proof techniques considered (see Section 7.2.4).

Each of these key tasks is outlined in more detail in the subsections below, explaining why they will make significant contributions, as well as insights into how such tasks could be completed. An indication of the effort that these tasks are believed to involve is also made.

7.2.1 Further Comparison Results

In Chapter 4 a DS definition for LPF was modified to formally define the semantics of other approaches to coping with logical formulae that can contain reference to partial terms. This enabled some comparisons to be made between the different approaches. First on the basis of the meaning of different expressions between the different approaches, and secondly on properties that hold in the non-classical logic approaches. The comparisons in Chapter 2 and Chapter 4 justified the choice of LPF for coping with partial terms.

It would also be beneficial to extend this work to compare the extra work that arises when carrying over two-valued classical logic proof procedures to the other non-classical logic approaches considered. Proof procedures for LPF were investigated in Chapter 6.

This extension to the work presented in this thesis would comprise of an investigation into the extent of the modifications needed for the resolution refutation procedure for weak Kleene logic and for McCarthy's conditional logic etc. against that needed for LPF. For instance, how does the sequential interpretation of McCarthy's conditional logic hold up against that of the parallel interpretation nature of LPF which provides the strongest monotonic extension of the familiar two-valued classical logic logical operators in the resolution refutation procedure. Specifically, in terms of the size of the resulting clausal form, the number of definedness obligations that need introducing into proofs, and thus the amount of work that the other logics introduce compared to LPF.

The corresponding $\mathcal{E}^i$ semantic functions definitions from Chapter 4 could be used to aid in defining the concepts, and used for proving the modifications made hold. Similar to how the $\mathcal{E}$ semantic function definition for LPF has been used in Chapter 6.

7.2.2 Further Performance Results

A resolution proof procedure alongside the associated technique of proof by contradiction has been proposed that is sound in LPF. However, the only results on the efficiency of this for LPF have come about from applying it to *small* examples. Thus one obvious extension of this work is to apply it to large case studies in order to be able to document more extensive results on its efficiency.

A suggested case study is the Mondex Electronic Purse system [SCW00]. An attempt at mechanising the Mondex Electronic Purse system in the Z/Eves theorem prover is discussed in [FW08]. To ensure expressions that apply partial functions are defined Z/Eves generates domain checks; examples of which are noted in [FW08].

Undertaking case studies will allow for a *reasonable* number of example proofs to be conducted. This will ensure that a more conclusive idea of the efficiency of resolution in LPF can be gained.

The time to complete such a task is linked to two key factors:

- the actual size of the case study involved; and
- whether implemented support of the procedures is available.

Thus to aid with completing such a task first undertaking all or a part of the task that is discussed next would be beneficial.

7.2.3 Implementing the Proof Techniques Considered

In order to aid in the task of undertaking case studies/conducting proofs a concrete implementation of resolution (with refutation) for LPF will be beneficial. The overall aim should be to mechanise the resolution procedure in a theorem prover, for instance in Prover 9 [McC10]. The reason behind the choice of Prover 9 is to have a small code base with which to modify, even though it is not a state of the art theorem prover like E [Sch02], or Vampire [RV02].

For earlier prototypes, taking a different approach may offer considerable benefits. In [Har09] John Harrison presents numerous fragments of *OCaml* code for methods including resolution that he describes in his book. Additionally, in [BA01] Mordechai Ben-Ari presents fragments of *Prolog* code for different methods including resolution that he describes in his book.

Implementing prototypes using either the aforementioned OCaml code or the aforementioned Prolog code could be a worthwhile first step as it should allow for a *quicker* implementation. This is because a smaller code base will need to be understood and to be extended, and the code is well documented in the two books cited above. This can lead to earlier case study results being able to be obtained, to get a further understanding of additional work that results from the proof procedures in LPF before moving the ideas across to a theorem proving system.

7.2.4 Investigating the Applicability of Further Proof Techniques for LPF

Some of the work presented in this thesis has all been focused on applying the basic proof procedure resolution and its associated technique of proof by contradiction in LPF. There has been little direct proof support for LPF over the years, so investigating the classical basic fundamental proof techniques in LPF was the essential and obvious starting point, to be able to gain insights into the issues that arise in applying these techniques to LPF, and to determine the extent of the modifications needed to cover LPF.

Key insights have been gained from this work on the topic of providing mechanised proof support for LPF. This work also provides the essential foundation with which to facilitate research into advanced proof techniques for LPF. A question that arises is: how many of the underlying ideas proposed for the adaption of the basic proof techniques for LPF considered in the main body of this thesis can be replayed to cover advanced proof techniques that are built around the basic proof techniques considered? Furthermore, can the ideas be replayed in another fundamental proof procedure of semantic tableauxs [BA01]? Some proof techniques will be considered in more depth below. In particular modifications to cover the demodulation proof technique for LPF are proposed using the ideas put forth in the main body of this thesis.

In Chapter 6 the equality symbol has been used in examples, but its treatment was just as a binary predicate that could be interpreted arbitrarily. So far, when considering validity, interpretations where the equality symbol is interpreted as something other than what equality normally means have had to be taken into account. However, the notion of equality plays a central role in formal methods/mathematical reasoning [Har09]. There is thus a need to consider those interpretations where equality is constrained to its normal meaning.

Constraining equality is a topic of future work, but key ideas to doing so are outlined in this section. One approach when using resolution to constrain equality is to add axioms such as reflexivity and symmetry etc. More efficient ways of coping with equality can be to use the demodulation or the paramodulation rule in addition to resolution and factoring.

One can constrain equality and deal with the equality predicate in resolution by adding the following equivalence and congruence axioms $Eq(\Gamma)$ to the set of clauses, that is, considering the set of clauses $\Gamma \cup Eq(\Gamma)$. However, resolution and factoring with these clauses can cause the generation of many unnecessary clauses.

Since equality is an equivalence relation it must be *reflexive*:

$$\forall x \cdot x = x$$

as well as *symmetric*:

$$\forall x \cdot \forall y \cdot x = y \;\Rightarrow\; y = x$$

and *transitive*:

$$\forall x \cdot \forall y \cdot \forall z \cdot x = y \wedge y = z \;\Rightarrow\; x = z$$

and *congruent*:

$$\forall x \cdot \forall y \cdot x = y \;\Rightarrow\; f(x) = f(y)$$

for each n-ary function, and *congruent*:

$$\forall x \cdot \forall y \cdot x = y \wedge P(x) \;\Rightarrow\; P(y)$$

for each n-ary predicate [Har09].

In LPF the notion of equality is weak/strict, that is, undefined if either operand is undefined, so while in two-valued classical logic reflexivity is a tautology it may not be in LPF as $x = x$ can be undefined. However, because of the use of quantification in these formulae, this notion of reflexivity can still be used in LPF. All variables are assumed to be universally quantified in the clausal form notation being used, and since quantification in LPF is only over sets of proper (i.e. defined) values then this reflexivity axiom is still holds in LPF. In effect the use of the universal quantifier is masking a *typing hypothesis* $x\colon T$ that needs to be present in LPF for reflexivity to hold. By the same reasoning, the symmetry and the transitivity axioms as they are presented above can be carried over to LPF.

However, the function congruence axiom and the predicate congruence axiom do not hold in LPF since it could be the case that x and y are equal to each other but when given as arguments to a function or to a predicate in the consequent a "gap" may arise, which causes a "gap" to arise in the whole formula. The function congruence rules need changing to:

$$\forall x \cdot \forall y \cdot (x = y \wedge \Delta(f(x))) \;\Rightarrow\; (f(x) = f(y))$$

for each n-ary function. The predicate congruence rules need changing to:

$$\forall x \cdot \forall y \cdot (x = y \wedge \Delta(P(x)) \wedge P(x)) \;\Rightarrow\; (P(y))$$

for each n-ary predicate.

The ideas from Chapter 6 form the essential foundation on which to research into the modification of advanced proof techniques for LPF. Dealing with demodulation and paramodulation appears to just be a case of applying similar techniques to the definedness of terms as to what has already been presented. The following discussion on demodulation and paramodulation introduces the proof techniques, and hints at an issue that may arise, as well as how such an issue could be solved to carry these techniques over to LPF. Further investigation and proofs of the following ideas are left as a topic of future work.

Demodulation [WRCS67] is a form of rewriting used for simplification. Given a clause $\{a = b\}$ and a clause $\{P(a')\}$, then $P(b)$ can be inferred, where the terms a and a' can be unified. If a, b, and a' are all *Var*s then this rule should carry over to LPF, as all *Var*s are defined. However, consider $a = b$ and $P(f(x))$, where a unifies with $f(x)$, but $f(x)$ could be undefined, while the *Var* a is guaranteed to be defined. The predicate P no longer has a possibly undefined function as an argument, but has a defined *Var* (from a quantifier) as an argument. Thus to guard against this possibility a unification constraint/a definedness obligation using the δ definedness logical operator on $f(x)$ will need to be introduced in the same way as has been discussed for resolution in Chapter 6.

Paramodulation [RW69, Har09, Bun10] is a technique that is generally used alongside resolution as a way of handling equality. Given a clause C_1, where $\{a = b\} \subseteq C_1$, where a and b are terms, and given a clause C_2, where $\{P[a']\} \subseteq C_2$, where $P[a']$ is a literal (possibly negative) that contains a subterm a', then a paramodulant of $\phi[C_1] \cup \phi[C_2] \cup \phi[P[b]]$ can be inferred, where ϕ is the mgu of a and a'. Note that $a = b$ can be interpreted as either $a = b$ or $b = a$, since the equality under consideration is not oriented.

Dealing with paramodulation in LPF should be the same as for demodulation by adding in unification constraints/definedness obligations using the δ definedness logical operator.

Additionally, a concrete implementation of these proof techniques for LPF will be of considerable benefit, as will applying such proof techniques in case studies to gain further extra results on the extra work that LPF carries.

It is worthwhile mentioning some similar work. In [Sch11] term rewriting is considered in the presence of partial functions. A partial order is used, that is, $x \sqsubseteq y$ iff $x \equiv \perp \vee x \equiv y$, where every operator is monotone. The SW sequent interpretation semantics are used but recall that an interpretation where $\mathbf{true} \vdash \perp_{\mathbb{B}}$ occurs is valid with the SW sequent interpretation semantics, but with such an interpretation the sequent will be invalid with the LPF SS sequent interpretation semantics. The author of [Sch11] uses directed rewriting, and so x can be replaced with y, if $x \sqsubseteq y$. This does not though carry over to LPF, since it is unsound to apply such term rewriting on a goal; definedness of the goal needs establishing in LPF. Recall that in LPF one reasons only from truth to truth.

Further work should also look at whether any further proof techniques can be carried over to LPF, and how efficiently. This work focused on investigating whether selected basic but fundamental proof techniques, could be carried over to LPF, identifying the issues that arise when applying them in LPF, and determining the extent of the modifications that needed making to carry them over to LPF. Extensions around these techniques have been proposed over the years, and ensuring that this work can be made use of in LPF is a vital task that needs undertaking, for instance, researching the application of *superposition* [BG94] in LPF.

The resolution refutation procedure that has been presented in Chapter 6 is not complete. Research into the refutation completeness of it for LPF is a valuable task. As it stands this cannot be complete due to the introduction of definedness obligations. Further cases for resolving need identifying and covering, for example, through the formulation of additional rule(s).

Bibliography

[AF97] Sten Agerholm and Jacob Frost. An Isabelle-based Theorem Prover for VDM-SL. In *In Proceedings of the 10th International Conference on Theorem Proving in Higher Order Logics (TPHOLs'97), LNCS*, pages 1–16. Springer-Verlag, 1997.

[Age94] Sten Agerholm. Domain Theory in HOL. In Jeffery Joyce and Carl-Johan Seger, editors, *Higher Order Logic Theorem Proving and Its Applications*, volume 780 of *Lecture Notes in Computer Science*, pages 295–309. Springer Berlin Heidelberg, 1994.

[AGM92] S. Abramsky, Dov M. Gabbay, and S. E. Maibaum, editors. *Handbook of logic in computer science (vol. 2): background: computational structures.* Oxford University Press, Inc., New York, NY, USA, 1992.

[Art96] R.D. Arthan. Undefinedness in Z: Issues for specification and proof. In *CADE-13 Workshop on Mechanization of Partial Functions*, pages 3–12. Springer-Verlag, 1996.

[BA01] Mordechai Ben-Ari. *Mathematical Logic for Computer Science.* Springer-Verlag, 2 edition, 2001.

[BBS$^+$05] Sergey Berezin, Clark Barrett, Igor Shikanian, Marsha Chechik, Arie Gurfinkel, and David L. Dill. A practical approach to partial functions in CVC Lite. *Electronic Notes in Theoretical Computer Science*, 125:13–23, July 2005.

[BCJ84] H. Barringer, J.H. Cheng, and C.B. Jones. A logic covering undefinedness in program proofs. *Acta Informatica*, 21:251–269, 1984.

[BFL$^+$94] J. C. Bicarregui, J. S. Fitzgerald, P. A. Lindsay, R. Moore, and B. Ritchie. *Proof in VDM: A Practitioner's Guide.* FACIT. Springer-Verlag, 1994.

[BG94] Leo Bachmair and Harald Ganzinger. Rewrite-based equational theorem proving with selection and simplification. *Journal of Logic and Computation*, 4(3):217–247, 1994.

[Bic98] J. C. Bicarregui, editor. *Proof in VDM: Case Studies.* FACIT. Springer-Verlag, 1998.

[Bla80] S. R. Blamey. *Partial Valued Logic.* PhD thesis, Oxford University, 1980.

[BM99] Juan C. Bicarregui and Brian M. Matthews. Proof and refutation in formal software development. In *3rd Irish Workshop on Formal Software Development*, 1999.

[Boc81] D.A. Bochvar. On a three-valued logical calculus and its application to the analysis of the paradoxes of the classical extended functional calculus. *History and Philosophy of Logic*, 2:87–112, 1981. Translated by Bergmann, Merrie.

[BT07] Clark Barrett and Cesare Tinelli. CVC3. In Werner Damm and Holger Hermanns, editors, *Proceedings of the* 19*th* *International Conference on Computer Aided Verification (CAV '07)*, volume 4590 of *Lecture Notes in Computer Science*, pages 298–302. Springer-Verlag, July 2007.

[Bun10] Alan Bundy. *The Computer Modelling of Mathematical Reasoning.* University of Edinburgh, digital edition, 2010.

[But55] Ronald J. Butler. Aristotle's sea fight and three-valued logic. *The Philosophical Review*, 64(2):pp. 264–274, 1955.

[CDE$^+$07] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Carolyn Talcott. *All about Maude — a high-performance logical framework: how to specify, program and verify systems in rewriting logic.* Springer-Verlag, 2007.

[Che86] J. H. Cheng. *A Logic for Partial Functions.* PhD thesis, University of Manchester, 1986.

[CJ91] J. H. Cheng and C. B. Jones. On the usability of logics which handle partial functions. In C. Morgan and J. C. P. Woodcock, editors, *3rd Refinement Workshop*, pages 51–69. Springer-Verlag, 1991.

[COR$^+$95] Judy Crow, Sam Owre, John Rushby, Natarajan Shankar, and Ayam Srivas. A tutorial introduction to PVS. In *Computer Science Laboratory, SRI International*, 1995.

[Cor10] SCSK Corporation. VDMTools: The VDM-SL Language Manual, 2010.

[CW96] Edmund M. Clarke and Jeannette M. Wing. Formal methods: state of the art and future directions. *ACM Computer Survey*, 28(4):626–643, December 1996.

[DCN$^+$00] Louise A. Dennis, Graham Collins, Michael Norrish, Richard J. Boulton, Konrad Slind, Graham Robinson, Michael J. C. Gordon, and Thomas F. Melham. The PROSPER Toolkit. In Susanne Graf and Michael I. Schwartzbach, editors, *Tools and Algorithms for Construction and Analysis of Systems, 6th International Conference, TACAS 2000, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS 2000, Berlin, Germany, March 25 - April 2, 2000*, volume 1785 of *Lecture Notes in Computer Science*, pages 78–92. Springer-Verlag, 2000.

[DMR08] Ádám Darvas, Farhad Mehta, and Arsenii Rudich. Efficient well-definedness checking. In *Proceedings of the 4th international joint conference on Automated Reasoning*, IJCAR '08, pages 100–115. Springer-Verlag, 2008.

[ELL94] R. Elmstrom, P. G. Larsen, and P. B. Lassen. The IFAD VDM-SL toolbox: A practical approach to formal specification. *SIGPLAN Not.*, 29:77–80, September 1994.

[Far90] William M. Farmer. A partial functions version of Church's simple theory of types. *Journal of Symbolic Logic*, 55(3):1269–1291, 1990.

[Far96] William M. Farmer. Mechanizing the traditional approach to partial functions. In M. Kohlhase W. Farmer, M. Kerber, editor, *Proceedings of the Workshop on the Mechanization of Partial Functions*, pages 27–32, 1996.

[FFL97] S. Finn, M. P. Fourman, and J. Longley. Partial functions in a total setting. *Journal of Automated Reasoning*, 18:85–104, 1997.

[Fit07] J. S. Fitzgerald. The typed Logic of Partial Functions and the Vienna Development Method. In D. Bjørner and M. C. Henson, editors, *Logics of Specification Languages*, EATCS Texts in Theoretical Computer Science, pages 427–461. Springer-Verlag, 2007.

[FJ08] J. S. Fitzgerald and C. B. Jones. The connection between two ways of reasoning about partial functions. *Information Processing Letters*, 107(3–4):128–132, 2008.

[FL09] J. S. Fitzgerald and P. G. Larsen. *Modelling Systems — Practical Tools and Techniques in Software Development.* Cambridge University Press, second edition, 2009.

[FLM$^+$05] J. S. Fitzgerald, P. G. Larsen, P. Mukherjee, N. Plat, and M. Verhoef. *Validated Designs for Object-oriented Systems.* Springer-Verlag, 2005.

[FLS08] J. S. Fitzgerald, P. G. Larsen, and S. Sahara. VDMTools: Advances in support for formal modelling in VDM. *SIGPLAN Not.*, 43:3–11, February 2008.

[FW08] Leo Freitas and Jim Woodcock. Mechanising Mondex with Z/Eves. *Formal Aspects of Computing*, 20(1):117–139, 2008.

[GHH$^+$92] Chris George, Peter Haff, Klaus Havelund, Anne E. Haxthausen, Robert Milne, Clause Bendix Nielson, Soren Prehn, and Kim Ritter Wagner. *The RAISE Specification Language.* BCS Practitioner Series. Prentice Hall, 1992.

[GHH$^+$95] C. George, A. E. Haxthausen, S. Hughes, R. Milne, S. Prehn, and J. S. Pedersen. *The RAISE Development Method.* BCS Practitioner Series. Prentice Hall, 1995.

[GM93] M. J. C. Gordon and T. F. Melham, editors. *Introduction to HOL: A theorem proving environment for higher order logic.* Cambridge University Press, 1993.

[GMW79] Michael J. C. Gordon, Robin Milner, and Christopher P. Wadsworth. *Edinburgh LCF*, volume 78 of *Lecture Notes in Computer Science.* Springer-Verlag, 1979.

[Got05] Siegfeld Gottwald. Many-valued logics. Technical report, Institute of Logic and Philosophy of Science, Leipzig University, May 2005.

[GS95] David Gries and Fred B. Schneider. Avoiding the undefined by underspecification. In *Computer Science Today: Recent Trends and Developments, number 1000 in Lecture Notes in Computer Science*, pages 366–373. Springer-Verlag, 1995.

[Häh05] Reiner Hähnle. Many-valued logic, partiality, and abstraction in formal specification languages. *Logic Journal of the IGPL*, 13(4):415–433, 2005.

[Hal90] Anthony Hall. Seven myths of formal methods. *IEEE Software*, 7(5):11–19, September 1990.

[Har09] John Harrison. *Handbook of Practical Logic and Automated Reasoning*. Cambridge University Press, 2009.

[Hol97] M. C. Holloway. Why engineers should consider formal methods. Technical report, NASA Langley Research Center, 1997.

[JJLM91] C. B. Jones, K. D. Jones, P. A. Lindsay, and R. Moore. *mural: A Formal Development Support System*. Springer-Verlag, 1991.

[JL11] C. B. Jones and M. J. Lovert. Semantic models for a logic of partial functions. *International Journal of Software and Informatics*, 5:55–76, 2011.

[JLS12a] C. B. Jones, M. J. Lovert, and L. J. Steggles. Towards a mechanisation of a logic of partial terms. Technical Report CS-TR-1314, Newcastle University, February 2012.

[JLS12b] Cliff B. Jones, Matthew J. Lovert, and L. Jason Steggles. A semantic analysis of logics that cope with partial terms. In J. Derrick et al., editors, *ABZ 2012*, volume 7316 of *Lecture Notes in Computer Science*, pages 252–265. Springer-Verlag, June 2012.

[JM94] C. B. Jones and C. A. Middelburg. A typed logic of partial functions reconstructed classically. *Acta Informatica*, 31(5):399–430, 1994.

[Jon90] C. B. Jones. *Systematic Software Development using VDM*. Prentice Hall International, second edition, 1990.

[Jon95] C. B. Jones. Partial functions and logics: A warning. *Information Processing Letters*, 54(2):65–67, 1995.

[Jon06] Cliff B. Jones. Reasoning about partial functions in the formal development of programs. In *Proceedings of AVoCS'05*, volume 145, pages 3–25. Elsevier, Electronic Notes in Theoretical Computer Science, 2006.

[Kah87] Gilles Kahn. Natural semantics. In *STACS '87: Proc. Fourth Annual Symposium on Theoretical Aspects of Computer Science*, pages 22–39. Springer-Verlag, 1987.

[KK94] Manfred Kerber and Michael Kohlhase. A mechanization of strong kleene logic for partial functions. In Alan Bundy, editor, *Proceedings of the 12th International Conference on Automated Deduction*, CADE-12, pages 371–385. Springer-Verlag, 1994.

[Kle38] S. C. Kleene. On notation for ordinal numbers. *The Journal of Symbolic Logic*, 3(4):pp. 150–155, 1938.

[Kle52] S. C. Kleene. *Introduction to Metamathematics*. Van Nostrad, 1952.

[Kol76] G. Koletsos. Sequent calculus and partial logic. Master's thesis, Manchester University, 1976.

[LBF$^+$10] P. G. Larsen, N. Battle, M. Ferreira, J. S. Fitzgerald, K. Lausdhal, and M. Verhoef. The Overture initiative integrating tools for VDM. *SIGSOFT Software Engineering Notes*, 35:1–6, January 2010.

[Lov10] M. J. Lovert. A semantic model for a logic of partial functions. In K. Pierce, N. Plat, and S. Wolff, editors, *Proceedings of the 8th Overture Workshop*, number CS-TR-1224 in School of Computing Science Technical Report, pages 33–45. Newcastle University, 2010.

[Łuk20] J. Łukasiewicz. O logice trojwartosciowej. *Ruch Filozoficzny*, 5:170–171, 1920. Translated as On three-valued logic, in L. Borkowski (ed.), Selected works by Jan Łukasiewicz, 1970.

[McC62] J. McCarthy. Towards a mathematical science of computation. In *In IFIP Congress*, pages 21–28. North-Holland, 1962.

[McC67] J. McCarthy. A basis for a mathematical theory for computation. In P. Braffort and D. Hirschberg, editors, *Computer Programming and Formal Systems*, pages 33–70. North-Holland Publishing Company, 1967.

[McC10] W. McCune. Prover9 and mace4. `http://www.cs.unm.edu/~mccune/prover9/`, 2005–2010.

[Meh08] Farhad Mehta. A practical approach to partiality — a proof based approach. In *ICFEM'08*, pages 238–257, 2008.

[MJ12] Aleksandar Milicevic and Daniel Jackson. Preventing arithmetic overflows in alloy. In J. Derrick et al., editors, *ABZ 2012*, volume 7316 of *Lecture Notes in Computer Science*, pages 108–121. Springer-Verlag, June 2012.

[MS97] O. Müller and K. Slind. Treating partiality in a logic of total functions. *The Computer Journal*, 40(10):640–652, 1997.

[NN92] Hanne Riis Nielson and Flemming Nielson. *Semantics with Applications A Formal Introduction.* Wiley Professional Computing, 1992.

[NWP02] Tobias Nipkow, Markus Wenzel, and Lawrence C. Paulson. *Isabelle/HOL: a proof assistant for higher-order logic.* Springer-Verlag, 2002.

[ORS92] S. Owre, J. M. Rushby, , and N. Shankar. PVS: A prototype verification system. In Deepak Kapur, editor, *11th International Conference on Automated Deduction (CADE)*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752. Springer-Verlag, June 1992.

[Owe97] Olaf Owe. Partial logics reconsidered: A conservative approach. *Formal Aspects of Computing*, 5:208–223, 1997.

[Plo81] G. D. Plotkin. A structural approach to operational semantics. Technical report, Aarhus University, 1981.

[Plo04] Gordon D. Plotkin. The origins of structural operational semantics. *Journal of Logic and Algebraic Programming*, 6061(0):3 – 15, 2004.

[Pri53] A. N. Prior. Three-valued logic and future contingents. *The Philosophical Quarterly*, 3(13):pp. 317–326, 1953.

[Rob65] J. A. Robinson. A machine-oriented logic based on the resolution principle. *J. ACM*, 12:23–41, 1965.

[Rus05] Bertrand Russell. On denoting. *Mind*, 14(56):pp. 479–493, 1905.

[RV02] Alexandre Riazanov and Andrei Voronkov. The design and implementation of vampire. *AI Communications*, 15(2,3):91–110, 2002.

[RW69] G. Robinson and L. Wos. Paramodulation and theorem proving in first order theories with equality. In B. Meltzer and D. Michie, editors, *Machine Intelligence, volume IV*, pages 135–150. American Elsevier, 1969.

[SB99] Birgit Schieder and Manfred Broy. Adapting calculational logic to the undefined. *The Computer Journal*, 42, 1999.

[Sch02] Stephan Schulz. E - a brainiac theorem prover. *AI Communications*, 15(2,3):111–126, 2002.

[Sch11] Matthias Schmalz. Term rewriting in logics of partial functions. In Shengchao Qin and Zongyan Qiu, editors, *Formal Methods and Software Engineering*, volume 6991 of *Lecture Notes in Computer Science*, pages 633–650. Springer-Verlag, 2011.

[SCW00] Susan Stepney, David Cooper, and Jim Woodcock. An Electronic Purse: Specification, Refinement, and Proof. Technical Report PRG-126, Oxford University Computing Laboratory Programming Research Group, July 2000.

[SDG99] Bill Stoddart, Steve Dunne, and Andy Galloway. Undefined Expressions and Logic in Z and B. *Form. Methods Syst. Des.*, 15(3):201–215, November 1999.

[Sid10] Theodore Sider. *Logic for Philosophy*. OUP Oxford, 2010.

[Spi92] J. M. Spivey. *The Z Notation: A Reference Manual*. Prentice Hall, second edition, 1992.

[Vad88] Sunil Vadera. A theory of unification. *Softw. Eng. J.*, 3:149–160, September 1988.

[Ver07] Sander D. Vermolen. Automatically discharging VDM proof obligations using HOL. Master's thesis, Radboud University Nijmegen Computing Science Department, 2007.

[VHL10] Sander D. Vermolen, Jozef Hooman, and Peter Gorm Larsen. Proving consistency of VDM models using HOL. In *Proceedings of the 2010 ACM Symposium on Applied Computing*, SAC '10, pages 2503–2510. ACM, 2010.

[VMH01] S. Vadera, F. Meziane, and M.-L.L. Huang. Experience with mural in formalising dust-expert. *Information and Software Technology*, 43(4):231 – 240, 2001.

[VMO06] Alberto Verdejo and Narciso Mart-Oliet. Executable structural operational semantics in Maude. *Journal of Logic and Algebraic Programming*, 67(12):226 – 293, 2006.

[Wal97] Michal Walicki. The history of mathematical logic (vastly abbreviated and horribly simplified), 1997.

[WF08] J. Woodcock and L. Freitas. Linking VDM and Z. In *13th IEEE International Conference on Engineering of Complex Computer Systems*, pages 143 –152, April 2008.

[Win90] Jeannette M. Wing. A specifier's introduction to formal methods. *IEEE Computer*, 23(9):8–23, September 1990.

[WLBF09] J. Woodcock, P. G. Larsen, J. Bicarregui, and J. S. Fitzgerald. Formal methods: Practice and experience. *ACM Computing Surveys*, 41:1–36, October 2009.

[WRCS67] Lawrence Wos, George A. Robinson, Daniel F. Carson, and Leon Shalla. The concept of demodulation in theorem proving. *Journal of the ACM*, 14(4):698–709, October 1967.

Appendix A

Full LPF Semantic Definitions

The full abstract syntax (Appendix A.1), context conditions (Appendix A.2), Structural Operational Semantic definitions, both big-step and small-step (Appendix A.3) and Denotational Semantic definitions (Appendix A.4) that define the semantics of LPF for evaluating different expression constructs are presented in this appendix.

A.1 Abstract Syntax

A.1.1 Expression Constructs

The selected expression constructs are all presented using abstract syntax. The available expressions are:

$$\begin{aligned} \mathit{Expr} = {} & \mathit{Value} \mid \mathit{Id} \mid \mathit{Arith} \mid \mathit{Equality} \mid \mathit{Cond} \mid \mathit{Not} \mid \mathit{delta} \mid \\ & \mathit{Or} \mid \mathit{Exists} \mid \mathit{FuncCall} \mid \mathit{PredCall} \end{aligned}$$

where *Value* represents the two data types available:

$$\mathit{Value} = \mathbb{B} \mid \mathbb{Z}$$

and where *Id* represents propositional variable identifiers (*Prop*), integer variable identifiers (*Var*), function identifiers (symbols) (*Fn*) and predicate identifiers (symbols) (*Pr*):

$$\mathit{Id} = \mathit{Prop} \mid \mathit{Var} \mid \mathit{Fn} \mid \mathit{Pr}$$

The four identifier sets are assumed to be disjoint.

The rest of the expression constructs are presented as records:

$$\begin{aligned} \mathit{Arith} \;::\; & a : \mathit{Expr} \\ & \mathit{op} : + \mid - \mid \times \mid \div \\ & b : \mathit{Expr} \end{aligned}$$

$$\begin{aligned} \mathit{Equality} \;::\; & a : \mathit{Expr} \\ & b : \mathit{Expr} \end{aligned}$$

$$\begin{aligned} \mathit{Cond} \;::\; & p : \mathit{Expr} \\ & a : \mathit{Expr} \\ & b : \mathit{Expr} \end{aligned}$$

$$\mathit{Not} \;::\; p : \mathit{Expr}$$

$$\mathit{delta} \;::\; p : \mathit{Expr}$$

$$\begin{array}{l} \mathit{Or} :: p : \mathit{Expr} \\ \qquad\; q : \mathit{Expr} \end{array}$$

$$\begin{array}{l} \mathit{Exists} :: x : \mathit{Var} \\ \qquad\qquad p : \mathit{Expr} \end{array}$$

$$\begin{array}{rl} \mathit{FuncCall} :: \mathit{function} & : \mathit{Fn} \\ \mathit{args} & : \mathit{Expr}^* \end{array}$$

$$\begin{array}{rl} \mathit{PredCall} :: \mathit{predicate} & : \mathit{Pr} \\ \mathit{args} & : \mathit{Expr}^* \end{array}$$

A.1.2 Syntactic Definitions

As usual, assuming the extra syntax is defined as above for the other expression constructs, the following syntactic definitions hold in LPF:

- The formula $mk\_And(p, q)$ is equivalent to the formula $mk\_Not(mk\_Or(mk\_Not(p), mk\_Not(q)))$;
- The formula $mk\_Implies(p, q)$ is equivalent to the formula $mk\_Or(mk\_Not(p), q)$;
- The formula $mk\_Iff(p, q)$ is equivalent to the formula $mk\_And(mk\_Implies(p, q), mk\_Implies(q, p))$; and
- The formula $mk\_Forall(x, p)$ is equivalent to the formula $mk\_Not(mk\_Exists(x, mk\_Not(p)))$.

These other operators are not defined in the semantic definitions that follow, since they can be defined in terms of other operators that are defined in the following semantic definitions. Defining these extra operators would be trivial, but would expand the size of the presentation of the semantics without adding clarity.

A.1.3 Function Definitions and Predicate Definitions

Function definitions and predicate definitions are also represented as records:

$$\begin{array}{rl} \mathit{Func} :: \mathit{params} & : \mathit{Var}^* \\ \mathit{result} & : \mathit{Expr} \end{array}$$

$$\begin{array}{rl} \mathit{Pred} :: \mathit{params} & : \mathit{Var}^* \\ \mathit{result} & : \mathit{Expr} \end{array}$$

A.2 Context Conditions

A.2.1 Type Map

To be able to perform type checks in the language a *Types* map is used that maps identifiers to the corresponding type $\textsc{Bool}$ or $\textsc{Int}$:

$$\begin{aligned} &\mathit{Types} = \mathit{Prop} \xrightarrow{m} \textsc{Bool} \mid \\ &\qquad \mathit{Var} \xrightarrow{m} \textsc{Int} \end{aligned}$$

$$\mathit{Type} = \textsc{Bool} \mid \textsc{Int}$$

In addition to the *Types* map a map called *Defs* is used that maps function identifiers and predicate identifiers to the corresponding function definitions and predicate definitions respectively:

$$\begin{aligned} &\mathit{Defs} = \mathit{Fn} \xrightarrow{m} \mathit{Func} \mid \\ &\qquad \mathit{Pr} \xrightarrow{m} \mathit{Pred} \end{aligned}$$

The *Defs* map is needed to be able to make some type checks.

A.2.2 Expressions

The context condition for expressions (*Expr*) is defined as:

$$\mathit{wf\text{-}Expr} : \mathit{Expr} \times \mathit{Types} \times \mathit{Defs} \to (\mathit{Type} \mid \textsc{Error})$$

```
$\mathit{wf\text{-}Expr}(e, \mathit{vars}, \mathit{defs}) \quad \triangleq$
    **cases** $e$ **of**
    The cases are defined below.
    **others** $\textsc{Error}$
    **end**
```

where the cases for each $e \in \mathit{Expr}$ are defined as:

$$e \in \mathbb{B} \to \textsc{Bool}$$

$$e \in \mathbb{Z} \to \textsc{Int}$$

```
$e \in \mathit{Prop} \to$ **if** $e \in$ **dom** $\mathit{vars}$
              **then** $\textsc{Bool}$
              **else** $\textsc{Error}$

$e \in \mathit{Var} \to$ **if** $e \in$ **dom** $\mathit{vars}$
             **then** $\textsc{Int}$
             **else** $\textsc{Error}$
```

```
$mk\_Arith(a, op, b) \to$ let $l = wf\text{-}Expr(a, vars, defs)$ in
                          let $r = wf\text{-}Expr(b, vars, defs)$ in
                          if $l = \textsc{Int} \wedge l = r \wedge op \in \{+, -, \times, \div\}$
                          then $\textsc{Int}$
                          else $\textsc{Error}$

$mk\_Equality(a, b) \to$ let $l = wf\text{-}Expr(a, vars, defs)$ in
                          let $r = wf\text{-}Expr(b, vars, defs)$ in
                          if $l = \textsc{Int} \wedge l = r$
                          then $\textsc{Bool}$
                          else $\textsc{Error}$

$mk\_Cond(p, a, b) \to$ let $l = wf\text{-}Expr(p, vars, defs)$ in
                          let $r = wf\text{-}Expr(a, vars, defs)$ in
                          let $s = wf\text{-}Expr(b, vars, defs)$ in
                          if $l = \textsc{Bool} \wedge r = \textsc{Int} \wedge r = s$
                          then $\textsc{Int}$
                          else $\textsc{Error}$

$mk\_Not(p) \to$ if $wf\text{-}Expr(p, vars, defs) = \textsc{Bool}$
                 then $\textsc{Bool}$
                 else $\textsc{Error}$

$mk\_delta(p) \to$ if $wf\text{-}Expr(p, vars, defs) = \textsc{Bool}$
                   then $\textsc{Bool}$
                   else $\textsc{Error}$

$mk\_Or(p, q) \to$ let $l = wf\text{-}Expr(p, vars, defs)$ in
                   let $r = wf\text{-}Expr(q, vars, defs)$ in
                   if $l = \textsc{Bool} \wedge l = r$
                   then $\textsc{Bool}$
                   else $\textsc{Error}$

$mk\_Exists(x, p) \to$ if $wf\text{-}Expr(p, vars \dagger \{x \mapsto \textsc{Int}\}) = \textsc{Bool}$
                       then $\textsc{Bool}$
                       else $\textsc{Error}$
```

$$\begin{aligned} \mathit{mk\_FuncCall}(id, args) \to\ & \textbf{if } (\forall i\colon \textbf{inds } args \cdot \\ & \qquad \mathit{wf\text{-}Expr}(args(i), vars, defs) = \textsc{Int}) \wedge \\ & \qquad\quad id \in \textbf{dom } defs \wedge \\ & \qquad\quad \textbf{len } args = \textbf{len } defs(id).params \\ & \textbf{then } \textsc{Int} \\ & \textbf{else } \textsc{Error} \end{aligned}$$

$$\begin{aligned} \mathit{mk\_PredCall}(id, args) \to\ & \textbf{if } (\forall i\colon \textbf{inds } args \cdot \\ & \qquad \mathit{wf\text{-}Expr}(args(i), vars, defs) = \textsc{Int}) \wedge \\ & \qquad\quad id \in \textbf{dom } defs \wedge \\ & \qquad\quad \textbf{len } args = \textbf{len } defs(id).params \\ & \textbf{then } \textsc{Bool} \\ & \textbf{else } \textsc{Error} \end{aligned}$$

A.2.3 Function Definitions and Predicate Definitions

The context conditions that check function definitions and predicate definitions are defined as:

$$\mathit{wf\text{-}Func} : \mathit{Func} \times \mathit{Types} \times \mathit{Defs} \to \mathbb{B}$$

$$\begin{aligned} & \mathit{wf\text{-}Func}(\mathit{mk\_Func}(p, r), vars, defs) \quad \triangleq \\ & \qquad \mathit{wf\text{-}Expr}(r, \{p(i) \mapsto \textsc{Int} \mid i\colon \textbf{inds } p\}, defs) = \textsc{Int} \end{aligned}$$

$$\mathit{wf\text{-}Pred} : \mathit{Pred} \times \mathit{Types} \times \mathit{Defs} \to \mathbb{B}$$

$$\begin{aligned} & \mathit{wf\text{-}Pred}(\mathit{mk\_Pred}(p, r), vars, defs) \quad \triangleq \\ & \qquad \mathit{wf\text{-}Expr}(r, \{p(i) \mapsto \textsc{Int} \mid i\colon \textbf{inds } p\}, defs) = \textsc{Bool} \end{aligned}$$

A.3 Structural Operational Semantics

A big-step SOS definition and a small-step SOS definition for LPF is presented in full in this appendix. The two SOS definitions are used to describe/model the process of expression evaluation according to the semantics of LPF.

A.3.1 Semantic Object

A map Σ from identifiers to values, function definitions and predicate definitions is defined as:

$$\begin{aligned} \Sigma = \ & \mathit{Prop} \xrightarrow{m} \mathbb{B} \mid \\ & \mathit{Var} \xrightarrow{m} \mathbb{Z} \mid \\ & \mathit{Fn} \xrightarrow{m} \mathit{Func} \mid \\ & \mathit{Pr} \xrightarrow{m} \mathit{Pred} \end{aligned}$$

where Σ is the set of all memory stores and a $\sigma \in \Sigma$ represents a specific mapping.

A *Prop* map can be partial to allow for undefined propositional identifiers, that is, a propositional identifier can be absent from the domain of a specific σ to represent an undefined propositional identifier. The other three maps are all assumed to be total. Function definitions (*Func*) and predicate definitions (*Pred*) can themselves be partial.

A.3.2 Big-Step Structural Operational Semantics Definition

The semantic (transition) relation is:

$\xrightarrow{e} : \mathcal{P}((Expr \times \Sigma) \times Value)$

The semantic (inference) rules follow.

Values

$$\boxed{Value\_E}\ \frac{v \in Value}{(v, \sigma) \xrightarrow{e} v}$$

Identifiers

$$\boxed{Prop\_E}\ \frac{\begin{array}{c} id \in Prop; \\ id \in \mathbf{dom}\, \sigma \end{array}}{(id, \sigma) \xrightarrow{e} \sigma(id)}$$

$$\boxed{Var\_E}\ \frac{id \in Var}{(id, \sigma) \xrightarrow{e} \sigma(id)}$$

Arithmetic

$$\boxed{Arith\_E1}\ \frac{\begin{array}{c} (a, \sigma) \xrightarrow{e} a'; \\ (b, \sigma) \xrightarrow{e} b'; \\ a' \in \mathbb{Z}; \\ b' \in \mathbb{Z}; \end{array}}{(mk\_Arith(a, +, b), \sigma) \xrightarrow{e} [\![+]\!](a', b')}$$

$$\boxed{Arith\_E2}\;\frac{(a,\sigma) \xrightarrow{e} a';\quad (b,\sigma) \xrightarrow{e} b';\quad a' \in \mathbb{Z};\quad b' \in \mathbb{Z};}{(mk\_Arith(a,-,b),\sigma) \xrightarrow{e} [\![-]\!](a',b')}$$

$$\boxed{Arith\_E3}\;\frac{(a,\sigma) \xrightarrow{e} a';\quad (b,\sigma) \xrightarrow{e} b';\quad a' \in \mathbb{Z};\quad b' \in \mathbb{Z};}{(mk\_Arith(a,\times,b),\sigma) \xrightarrow{e} [\![\times]\!](a',b')}$$

$$\boxed{Arith\_E4}\;\frac{(a,\sigma) \xrightarrow{e} a';\quad (b,\sigma) \xrightarrow{e} b';\quad a' \in \mathbb{Z};\quad b' \in \mathbb{Z};\quad b' \neq 0}{(mk\_Arith(a,\div,b),\sigma) \xrightarrow{e} [\![\div]\!](a',b')}$$

where $a \in T$ in all of the SOS rules checks whether a is a member of T, that is $0 \in \mathbb{Z}$ is true, but $mk\_Arith(1,+,1) \in \mathbb{Z}$ is false, since $mk\_Arith(1,+,1)$ has not yet been evaluated to a constant integer value. Furthermore, $[\![op]\!](a,b)$ is to be regarded as the standard mathematical result of the specified operator op applied to two given operands a and b.

Equality

$$\boxed{Equality\_E}\;\frac{(a,\sigma) \xrightarrow{e} a';\quad (b,\sigma) \xrightarrow{e} b';\quad a' \in \mathbb{Z};\quad b' \in \mathbb{Z}}{(mk\_Equality(a,b),\sigma) \xrightarrow{e} [\![=]\!](a',b')}$$

The Conditional Expression

$$\boxed{\mathit{Cond\_E1}}\ \frac{(p,\sigma) \xrightarrow{e} \mathbf{true};\quad (a,\sigma) \xrightarrow{e} a';\quad a' \in \mathbb{Z}}{(\mathit{mk\_Cond}(p,a,b),\sigma) \xrightarrow{e} a'}$$

$$\boxed{\mathit{Cond\_E2}}\ \frac{(p,\sigma) \xrightarrow{e} \mathbf{false};\quad (b,\sigma) \xrightarrow{e} b';\quad b' \in \mathbb{Z}}{(\mathit{mk\_Cond}(p,a,b),\sigma) \xrightarrow{e} b'}$$

Negation

$$\boxed{\mathit{Not\_E1}}\ \frac{(p,\sigma) \xrightarrow{e} \mathbf{true}}{(\mathit{mk\_Not}(p),\sigma) \xrightarrow{e} \mathbf{false}}$$

$$\boxed{\mathit{Not\_E2}}\ \frac{(p,\sigma) \xrightarrow{e} \mathbf{false}}{(\mathit{mk\_Not}(p),\sigma) \xrightarrow{e} \mathbf{true}}$$

The δ Operator

$$\boxed{\mathit{delta\_E1}}\ \frac{(p,\sigma) \xrightarrow{e} \mathbf{true}}{(\mathit{mk\_delta}(p),\sigma) \xrightarrow{e} \mathbf{true}}$$

$$\boxed{\mathit{delta\_E2}}\ \frac{(p,\sigma) \xrightarrow{e} \mathbf{false}}{(\mathit{mk\_delta}(p),\sigma) \xrightarrow{e} \mathbf{true}}$$

Disjunction

$$\boxed{\mathit{Or\_E1}}\ \frac{(p,\sigma) \xrightarrow{e} \mathbf{true}}{(\mathit{mk\_Or}(p,q),\sigma) \xrightarrow{e} \mathbf{true}}$$

$$\boxed{\mathit{Or\_E2}}\ \frac{(q,\sigma) \xrightarrow{e} \mathbf{true}}{(\mathit{mk\_Or}(p,q),\sigma) \xrightarrow{e} \mathbf{true}}$$

$$\boxed{Or\_E3}\ \frac{(p,\sigma) \xrightarrow{e} \mathbf{false};\quad (q,\sigma) \xrightarrow{e} \mathbf{false}}{(mk\_Or(p,q),\sigma) \xrightarrow{e} \mathbf{false}}$$

Existenital Quantification

$$\boxed{Exists\_E1}\ \frac{\exists i{:}\, \mathbb{Z} \cdot (p, \sigma \dagger \{x \mapsto i\}) \xrightarrow{e} \mathbf{true}}{(mk\_Exists(x,p),\sigma) \xrightarrow{e} \mathbf{true}}$$

$$\boxed{Exists\_E2}\ \frac{\forall i{:}\, \mathbb{Z} \cdot (p, \sigma \dagger \{x \mapsto i\}) \xrightarrow{e} \mathbf{false}}{(mk\_Exists(x,p),\sigma) \xrightarrow{e} \mathbf{false}}$$

Consider the $\exists$ above the line as a disjunction, and the $\forall$ above the line as a conjunction. See the small-step SOS definition in Appendix A.3.3 for the alternative approach that does not define a quantifier with a quantifier.

Function Application

$$\boxed{FuncCall\_E}\ \frac{\begin{array}{c}\mathbf{let}\ a = [args'(i) \mid i{:}\,\mathbf{inds}\ args \wedge \\ (args(i),\sigma) \xrightarrow{e} args'(i) \wedge args'(i) \in \mathbb{Z}]\ \mathbf{in} \\ \mathbf{len}\ args = \mathbf{len}\ a; \\ (\sigma(id).result, \sigma \dagger \{\sigma(id).params(i) \mapsto a(i) \mid \\ i{:}\,\mathbf{inds}\ \sigma(id).params\}) \xrightarrow{e} res; \\ res \in \mathbb{Z}\end{array}}{(mk\_FuncCall(id, args),\sigma) \xrightarrow{e} res}$$

Predicate Application

$$\boxed{PredCall\_E}\ \frac{\begin{array}{c}\mathbf{let}\ a = [args'(i) \mid i{:}\,\mathbf{inds}\ args \wedge \\ (args(i),\sigma) \xrightarrow{e} args'(i) \wedge args'(i) \in \mathbb{B}]\ \mathbf{in} \\ \mathbf{len}\ args = \mathbf{len}\ a; \\ (\sigma(id).result, \sigma \dagger \{\sigma(id).params(i) \mapsto a(i) \mid \\ i{:}\,\mathbf{inds}\ \sigma(id).params\}) \xrightarrow{e} res; \\ res \in \mathbb{B}\end{array}}{(mk\_PredCall(id, args),\sigma) \xrightarrow{e} res}$$

A.3.3 Small-Step Structural Operational Semantics Definition

The semantic (transition) relation is:

$$\xrightarrow{e} : \mathcal{P}((Expr \times \Sigma) \times Expr)$$

where $\xrightarrow{E}$ is the reflexive, transitive closure of $\xrightarrow{e}$ such that:

$$(e, \sigma) \xrightarrow{E} v \iff e = v \vee \exists e' : Expr \cdot (e, \sigma) \xrightarrow{e} e' \wedge (e', \sigma) \xrightarrow{E} v$$

The semantic (inference) rules follow.

Values

$$\boxed{Value\_E}\ \frac{v \in Value}{(v, \sigma) \xrightarrow{e} v}$$

Identifiers

$$\boxed{Prop\_E}\ \frac{id \in Prop;\quad id \in \mathbf{dom}\,\sigma}{(id, \sigma) \xrightarrow{e} \sigma(id)}$$

$$\boxed{Var\_E}\ \frac{id \in Var}{(id, \sigma) \xrightarrow{e} \sigma(id)}$$

Arithmetic

$$\boxed{Arith\_L}\ \frac{(a, \sigma) \xrightarrow{e} a'}{(mk\_Arith(a, op, b), \sigma) \xrightarrow{e} mk\_Arith(a', op, b)}$$

$$\boxed{Arith\_R}\ \frac{(b, \sigma) \xrightarrow{e} b'}{(mk\_Arith(a, op, b), \sigma) \xrightarrow{e} mk\_Arith(a, op, b')}$$

$$\boxed{Arith\_E1}\ \frac{a \in \mathbb{Z};\quad b \in \mathbb{Z}}{(mk\_Arith(a, +, b), \sigma) \xrightarrow{e} [\![+]\!](a, b)}$$

$$\boxed{\mathit{Arith\_E2}}\ \frac{a \in \mathbb{Z};\quad b \in \mathbb{Z}}{(\mathit{mk\_Arith}(a, -, b), \sigma) \xrightarrow{e} [\![-]\!](a, b)}$$

$$\boxed{\mathit{Arith\_E3}}\ \frac{a \in \mathbb{Z};\quad b \in \mathbb{Z}}{(\mathit{mk\_Arith}(a, \times, b), \sigma) \xrightarrow{e} [\![\times]\!](a, b)}$$

$$\boxed{\mathit{Arith\_E4}}\ \frac{a \in \mathbb{Z};\quad b \in \mathbb{Z};\quad b \neq 0}{(\mathit{mk\_Arith}(a, \div, b), \sigma) \xrightarrow{e} [\![\div]\!](a, b)}$$

Equality

$$\boxed{\mathit{Equality\_L}}\ \frac{(a, \sigma) \xrightarrow{e} a'}{(\mathit{mk\_Equality}(a, b), \sigma) \xrightarrow{e} \mathit{mk\_Equality}(a', b)}$$

$$\boxed{\mathit{Equality\_R}}\ \frac{(b, \sigma) \xrightarrow{e} b'}{(\mathit{mk\_Equality}(a, b), \sigma) \xrightarrow{e} \mathit{mk\_Equality}(a, b')}$$

$$\boxed{\mathit{Equality\_E}}\ \frac{a \in \mathbb{Z};\quad b \in \mathbb{Z}}{(\mathit{mk\_Equality}(a, b), \sigma) \xrightarrow{e} [\![=]\!](a, b)}$$

The Conditional Expression

$$\boxed{\mathit{Cond\_A}}\ \frac{(p, \sigma) \xrightarrow{e} p'}{(\mathit{mk\_Cond}(p, a, b), \sigma) \xrightarrow{e} \mathit{mk\_Cond}(p', a, b)}$$

$$\boxed{\mathit{Cond\_E1}}\ \frac{}{(\mathit{mk\_Cond}(\mathbf{true}, a, b), \sigma) \xrightarrow{e} a}$$

$$\boxed{Cond\_E2}\ \frac{}{(mk\_Cond(\mathbf{false}, a, b), \sigma) \xrightarrow{e} b}$$

Negation

$$\boxed{Not\_A}\ \frac{(p, \sigma) \xrightarrow{e} p'}{(mk\_Not(p), \sigma) \xrightarrow{e} mk\_Not(p')}$$

$$\boxed{Not\_E1}\ \frac{}{(mk\_Not(\mathbf{true}), \sigma) \xrightarrow{e} \mathbf{false}}$$

$$\boxed{Not\_E2}\ \frac{}{(mk\_Not(\mathbf{false}), \sigma) \xrightarrow{e} \mathbf{true}}$$

The δ Operator

$$\boxed{delta\_A}\ \frac{(p, \sigma) \xrightarrow{e} p'}{(mk\_delta(p), \sigma) \xrightarrow{e} mk\_delta(p')}$$

$$\boxed{delta\_E1}\ \frac{}{(mk\_delta(\mathbf{true}), \sigma) \xrightarrow{e} \mathbf{true}}$$

$$\boxed{delta\_E2}\ \frac{}{(mk\_delta(\mathbf{false}), \sigma) \xrightarrow{e} \mathbf{true}}$$

Disjunction

$$\boxed{Or\_L}\ \frac{(p, \sigma) \xrightarrow{e} p'}{(mk\_Or(p, q), \sigma) \xrightarrow{e} mk\_Or(p', q)}$$

$$\boxed{Or\_R}\ \frac{(q, \sigma) \xrightarrow{e} q'}{(mk\_Or(p, q), \sigma) \xrightarrow{e} mk\_Or(p, q')}$$

$$\boxed{Or\_E1}\ \frac{}{(mk\text{-}Or(\mathbf{true}, q), \sigma) \xrightarrow{e} \mathbf{true}}$$

$$\boxed{Or\_E2}\ \frac{}{(mk\text{-}Or(p, \mathbf{true}), \sigma) \xrightarrow{e} \mathbf{true}}$$

$$\boxed{Or\_E3}\ \frac{}{(mk\text{-}Or(\mathbf{false}, \mathbf{false}), \sigma) \xrightarrow{e} \mathbf{false}}$$

Existenital Quantification

$$\boxed{Exists\_E1}\ \frac{\exists i{:}\, \mathbb{Z} \cdot (p, \sigma \dagger \{x \mapsto i\}) \xrightarrow{E} \mathbf{true}}{(mk\text{-}Exists(x, p), \sigma) \xrightarrow{e} \mathbf{true}}$$

$$\boxed{Exists\_E2}\ \frac{\forall i{:}\, \mathbb{Z} \cdot (p, \sigma \dagger \{x \mapsto i\}) \xrightarrow{E} \mathbf{false}}{(mk\text{-}Exists(x, p), \sigma) \xrightarrow{e} \mathbf{false}}$$

or:

$$Expr = \ldots \mid ExistsInter$$

$$\begin{array}{lrcl} ExistsInter :: & x & : & Id \\ & pairs & : & ExistsPair^* \end{array}$$

where:

$$\begin{array}{lrcl} ExistsPair :: & i & : & \mathbb{Z} \\ & p & : & Expr \end{array}$$

$$\boxed{Exists\_E}\ \frac{}{\begin{array}{c}(mk\text{-}Exists(x, p), \sigma) \xrightarrow{e} \\ mk\text{-}ExistsInter(x, [mk\text{-}ExistsPair(i, p) \mid i{:}\, \mathbb{Z}])\end{array}}$$

$$\boxed{ExistsInter\_A}\ \frac{\begin{array}{c}\mathbf{let}\ j \in \mathbf{inds}\ pairs\ \mathbf{in} \\ (pairs(j).p, \sigma \dagger \{x \mapsto pairs(j).i\}) \xrightarrow{e} pairs'(j).p\end{array}}{(mk\text{-}ExistsInter(x, pairs), \sigma) \xrightarrow{e} mk\text{-}ExistsInter(x, pairs')}$$

where *pairs′* is *pairs* but incorporating the change made to the jth element:

$pairs'(j).p$. Also the let statement is to make an arbitrary choice.

$$\boxed{\mathit{ExistsInter\_E1}}\ \frac{\mathbf{true} \in \{pairs(i).p \mid i\colon \mathbf{inds}\, pairs\}}{(mk\_ExistsInter(x, pairs), \sigma) \xrightarrow{e} \mathbf{true}}$$

$$\boxed{\mathit{ExistsInter\_E2}}\ \frac{\{pairs(i).p \mid i\colon \mathbf{inds}\, pairs\} = \{\mathbf{false}\}}{(mk\_ExistsInter(x, pairs), \sigma) \xrightarrow{e} \mathbf{false}}$$

Function Application

$$Expr = \ldots \mid FuncInter$$

$$\begin{aligned} FuncInter \ :: \quad result &: Expr \\ paramid &: Id^* \\ args &: Expr^* \end{aligned}$$

$$\boxed{\mathit{FuncCall\_A}}\ \frac{\mathbf{let}\ i \in \mathbf{inds}\, args\ \mathbf{in}\ (args(i), \sigma) \xrightarrow{e} args'(i)}{(mk\_FuncCall(id, args), \sigma) \xrightarrow{e} mk\_FuncCall(id, args')}$$

$$\boxed{\mathit{FuncCall\_E}}\ \frac{[args(i) \mid i\colon \mathbf{inds}\, args \wedge args(i) \in \mathbb{Z}] = args}{\begin{array}{c}(mk\_FuncCall(id, args), \sigma) \xrightarrow{e} \\ mk\_FuncInter(\sigma(id).result, \sigma(id).params, args)\end{array}}$$

$$\boxed{\mathit{FuncInter\_A}}\ \frac{(res, \sigma \dagger \{paramids(i) \mapsto args(i) \mid i\colon \mathbf{inds}\, paramids\}) \xrightarrow{e} res'}{\begin{array}{c}(mk\_FuncInter(res, paramids, args), \sigma) \xrightarrow{e} \\ mk\_FuncInter(res', paramids, args)\end{array}}$$

$$\boxed{\mathit{FuncInter\_E}}\ \frac{res \in \mathbb{Z}}{(mk\_FuncInter(res, paramids, args), \sigma) \xrightarrow{e} res}$$

Predicate Application

$$Expr = \ldots \mid PredInter$$

$$\begin{aligned} PredInter \ :: \quad result &: Expr \\ paramid &: Id^* \\ args &: Expr^* \end{aligned}$$

$$\boxed{\mathit{PredCall\_A}}\ \frac{\textbf{let } i \in \textbf{inds } \mathit{args} \textbf{ in } (\mathit{args}(i), \sigma) \xrightarrow{e} \mathit{args}'(i)}{(\mathit{mk\_PredCall}(\mathit{id}, \mathit{args}), \sigma) \xrightarrow{e} \mathit{mk\_PredCall}(\mathit{id}, \mathit{args}')}$$

$$\boxed{\mathit{PredCall\_E}}\ \frac{[\mathit{args}(i) \mid i \colon \textbf{inds } \mathit{args} \wedge \mathit{args}(i) \in \mathbb{B}] = \mathit{args}}{\begin{array}{c}(\mathit{mk\_PredCall}(\mathit{id}, \mathit{args}), \sigma) \xrightarrow{e} \\ \mathit{mk\_PredInter}(\sigma(\mathit{id}).\mathit{result}, \sigma(\mathit{id}).\mathit{params}, \mathit{args})\end{array}}$$

$$\boxed{\mathit{PredInter\_A}}\ \frac{(\mathit{res}, \sigma \dagger \{\mathit{paramids}(i) \mapsto \mathit{args}(i) \mid i \colon \textbf{inds } \mathit{paramids}\}) \xrightarrow{e} \mathit{res}'}{\begin{array}{c}(\mathit{mk\_PredInter}(\mathit{res}, \mathit{paramids}, \mathit{args}), \sigma) \xrightarrow{e} \\ \mathit{mk\_PredInter}(\mathit{res}', \mathit{paramids}, \mathit{args})\end{array}}$$

$$\boxed{\mathit{PredInter\_E}}\ \frac{\mathit{res} \in \mathbb{B}}{(\mathit{mk\_PredInter}(\mathit{res}, \mathit{paramids}, \mathit{args}), \sigma) \xrightarrow{e} \mathit{res}}$$

A.4 Denotational Semantics

The full LPF denotation semantic definitions are presented in this appendix. They provide set theoretic definitions of the values that are denoted by expressions.

A.4.1 Expressions

The set of expressions used here is Expr^{Δ} which is *Expr* as defined previously but with the addition of the Δ operator.

$$\mathit{Expr}^{\Delta} = \mathit{Expr} \mid \mathit{Delta}$$

where:

$$\mathit{Delta} :: p : \mathit{Expr}$$

A.4.2 Context Conditions

The $\mathit{wf}\text{-}\mathit{Expr}^{\Delta}$ context condition is defined as:

$wf\text{-}Expr^{\Delta} : Expr^{\Delta} \times Types \rightarrow (Type \mid \textsc{Error})$

```
$wf\text{-}Expr^{\Delta}(e, vars) \quad \triangleq$
    **cases** $e$ **of**
                ⋮
    $mk\_Delta(p) \rightarrow$ **if** $wf\text{-}Expr(p, vars) = \textsc{Bool}$
                      **then** $\textsc{Bool}$
                      **else** $\textsc{Error}$
                ⋮

    **others** $\textsc{Error}$
    **end**
```

The rest of the cases follow as presented for *wf-Expr*.

A.4.3 Semantic Object

Σ is updated to:

$$\begin{aligned}\Sigma = {} & Prop \xrightarrow{m} \mathbb{B} \mid \\ & Var \xrightarrow{m} \mathbb{Z} \mid \\ & Fn \xrightarrow{m} Function \mid \\ & Pr \xrightarrow{m} Predicate\end{aligned}$$

where functions and predicates are now defined as:

$$Function = \mathcal{P}(\mathbb{Z}^* \times \mathbb{Z})$$

$$Predicate = \mathcal{P}(\mathbb{Z}^* \times \mathbb{B})$$

where function definitions *Function* and predicate definitions *Predicate* can both still be partial. No context conditions are now needed for function definitions and for predicate definitions.

A.4.4 Denotational Semantic Definition 1

The semantic relation is defined as:

$$\mathcal{E} \colon \mathcal{P}((Expr^{\Delta} \times \Sigma) \times Value)$$

$\mathcal{E}$ is defined in parts:

$$\begin{aligned}\mathcal{E} = {} & \mathcal{E}value \cup \mathcal{E}id \cup \mathcal{E}arith \cup \mathcal{E}equality \cup \mathcal{E}cond \cup \mathcal{E}or \cup \mathcal{E}not \cup \mathcal{E}delta \cup \\ & \qquad \mathcal{E}Delta \cup \mathcal{E}exists \cup \mathcal{E}funccall \cup \mathcal{E}predcall\end{aligned}$$

where:

$$\begin{aligned}&\mathcal{E}value = \\ &\quad \{((e, \sigma), e) \mid e \in Value\}\end{aligned}$$

$$
\begin{aligned}
&\mathcal{E}\,id = \\
&\quad \{((v,\sigma),\sigma(v)) \mid v \in \mathit{Prop} \wedge v \in \mathbf{dom}\,\sigma\} \cup \\
&\quad \{((v,\sigma),\sigma(v)) \mid v \in \mathit{Var}\}
\end{aligned}
$$

$$
\begin{aligned}
&\mathcal{E}\,arith = \\
&\quad \{((mk\_Arith(a,op,b),\sigma), [\![op]\!](a',b')) \mid \\
&\qquad ((a,\sigma),a') \in \mathcal{E} \wedge ((b,\sigma),b') \in \mathcal{E} \wedge op \in \{+,-,\times\}\} \cup \\
&\quad \{((mk\_Arith(a,\div,b),\sigma), [\![\div]\!](a',b')) \mid \\
&\qquad ((a,\sigma),a') \in \mathcal{E} \ \wedge \ ((b,\sigma),b') \in \mathcal{E} \ \wedge \ b' \neq 0\}
\end{aligned}
$$

$$
\begin{aligned}
&\mathcal{E}\,equality = \\
&\quad \{((mk\_Equality(a,b),\sigma), [\![=]\!](a',b')) \mid \\
&\qquad ((a,\sigma),a') \in \mathcal{E} \wedge ((b,\sigma),b') \in \mathcal{E}\}
\end{aligned}
$$

$$
\begin{aligned}
&\mathcal{E}\,cond = \\
&\quad \{((mk\_Cond(p,a,b),\sigma),a') \mid \\
&\qquad ((p,\sigma),\mathbf{true}) \in \mathcal{E} \wedge ((a,\sigma),a') \in \mathcal{E}\} \cup \\
&\quad \{((mk\_Cond(p,a,b),\sigma),b') \mid \\
&\qquad ((p,\sigma),\mathbf{false}) \in \mathcal{E} \wedge ((b,\sigma),b') \in \mathcal{E}\}
\end{aligned}
$$

$$
\begin{aligned}
&\mathcal{E}\,not = \\
&\quad \{((mk\_Not(p),\sigma),\mathbf{false}) \mid ((p,\sigma),\mathbf{true}) \in \mathcal{E}\} \cup \\
&\quad \{((mk\_Not(p),\sigma),\mathbf{true}) \mid ((p,\sigma),\mathbf{false}) \in \mathcal{E}\}
\end{aligned}
$$

$$
\begin{aligned}
&\mathcal{E}\,delta = \\
&\quad \{((mk\_delta(p),\sigma),\mathbf{true}) \mid (p,\sigma) \in \mathbf{dom}\,\mathcal{E}\}
\end{aligned}
$$

$$
\begin{aligned}
&\mathcal{E}\,Delta = \\
&\quad \{((mk\_Delta(p),\sigma),\mathbf{true}) \mid \\
&\qquad (p,\sigma) \in \mathbf{dom}\,\mathcal{E}\} \cup \\
&\quad \{((mk\_Delta(p),\sigma),\mathbf{false}) \mid \\
&\qquad (p,\sigma) \in (\{(p,\sigma) \mid \sigma \in \Sigma\} \setminus \{(p,\sigma) \mid (p,\sigma) \in \mathbf{dom}\,\mathcal{E}\})\}
\end{aligned}
$$

$$
\begin{aligned}
&\mathcal{E}\,or = \\
&\quad \{((mk\_Or(p,q),\sigma),\mathbf{true}) \mid ((p,\sigma),\mathbf{true}) \in \mathcal{E}\} \cup \\
&\quad \{((mk\_Or(p,q),\sigma),\mathbf{true}) \mid ((q,\sigma),\mathbf{true}) \in \mathcal{E}\} \cup \\
&\quad \{((mk\_Or(p,q),\sigma),\mathbf{false}) \mid ((p,\sigma),\mathbf{false}) \in \mathcal{E} \wedge ((q,\sigma),\mathbf{false}) \in \mathcal{E}\}
\end{aligned}
$$

$$
\begin{aligned}
&\mathcal{E}\,exists = \\
&\quad \{((mk\_Exists(x,p),\sigma),\mathbf{true}) \mid \\
&\qquad \mathbf{true} \in \mathbf{rng}\,(\{(p,\sigma \dagger \{x \mapsto i\}) \mid i\colon\mathbb{Z}\} \lhd \mathcal{E})\} \cup \\
&\quad \{((mk\_Exists(x,p),\sigma),\mathbf{false}) \mid \\
&\qquad \mathbf{rng}\,(\{(p,\sigma \dagger \{x \mapsto i\}) \mid i\colon\mathbb{Z}\} \lhd \mathcal{E}) = \{\mathbf{false}\}\}
\end{aligned}
$$

$$
\begin{aligned}
&\mathcal{E}\mathit{funccall} = \\
&\quad \{((\mathit{mk\_FuncCall}(f, al), \sigma), \mathit{res}) \mid \\
&\qquad \forall i\colon \mathbf{inds}\ al \cdot ((al(i), \sigma), vl(i)) \in \mathcal{E} \wedge \\
&\qquad (vl, \mathit{res}) \in \sigma(f)\}
\end{aligned}
$$

$$
\begin{aligned}
&\mathcal{E}\mathit{predcall} = \\
&\quad \{((\mathit{mk\_PredCall}(p, al), \sigma), \mathit{res}) \mid \\
&\qquad \forall i\colon \mathbf{inds}\ al \cdot ((al(i), \sigma), vl(i)) \in \mathcal{E} \wedge \\
&\qquad (vl, \mathit{res}) \in \sigma(p)\}
\end{aligned}
$$

A.4.5 Denotational Semantic Definition 2

The $\mathcal{E}$ semantic function is defined as:

$$\mathcal{E} : \mathit{Expr}^{\Delta} \to \mathcal{P}(\Sigma \times \mathit{Value})$$

$$\mathcal{E}(e) \quad \triangleq$$

cases e **of**

The cases are defined below.

end

where:

$$e \in \mathit{Value} \to \{(\sigma, e) \mid \sigma \in \Sigma\}$$

$$e \in \mathit{Prop} \to \{(\sigma, \sigma(e)) \mid \sigma \in \Sigma \wedge e \in \mathbf{dom}\,\sigma\}$$

$$e \in \mathit{Var} \to \{(\sigma, \sigma(e)) \mid \sigma \in \Sigma\}$$

$$
\begin{aligned}
&\mathit{mk\_Arith}(a, op, b) \to \\
&\qquad \{(\sigma, [\![op]\!](a', b')) \mid \\
&\qquad\qquad (\sigma, a') \in \mathcal{E}(a) \wedge (\sigma, b') \in \mathcal{E}(b) \wedge \\
&\qquad\qquad\quad op \in \{+, -, \times\}\} \cup \\
&\qquad \{(\sigma, [\![\div]\!](a', b')) \mid \\
&\qquad\qquad (\sigma, a') \in \mathcal{E}(a) \wedge (\sigma, b') \in \mathcal{E}(b) \wedge \\
&\qquad\qquad\quad op = \div \wedge b' \neq 0\}
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{mk\_Equality}(a, b) \to \\
&\qquad \{(\sigma, [\![=]\!](a', b')) \mid \\
&\qquad\qquad (\sigma, a') \in \mathcal{E}(a) \wedge (\sigma, b') \in \mathcal{E}(b)\}
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{mk\_Cond}(p, a, b) \to \\
&\qquad \{(\sigma, a') \mid \\
&\qquad\qquad (\sigma, \mathbf{true}) \in \mathcal{E}(p) \wedge (\sigma, a') \in \mathcal{E}(a)\} \cup \\
&\qquad \{(\sigma, b') \mid \\
&\qquad\qquad (\sigma, \mathbf{false}) \in \mathcal{E}(p) \wedge (\sigma, b') \in \mathcal{E}(b)\}
\end{aligned}
$$

$mk\_Not(p) \to$

$$\begin{aligned}&\{(\sigma, \mathbf{true}) \mid (\sigma, \mathbf{false}) \in \mathcal{E}(p)\} \cup\\ &\{(\sigma, \mathbf{false}) \mid (\sigma, \mathbf{true}) \in \mathcal{E}(p)\}\end{aligned}$$

$mk\_delta(p) \to$

$$\{(\sigma, \mathbf{true}) \mid \sigma \in \mathbf{dom}\, \mathcal{E}(p)\} \cup$$

$mk\_Delta(p) \to$

$$\begin{aligned}&\{(\sigma, \mathbf{true}) \mid \sigma \in \mathbf{dom}\, \mathcal{E}(p)\} \cup\\ &\{(\sigma, \mathbf{false}) \mid \sigma \in (\Sigma \setminus \mathbf{dom}\, \mathcal{E}(p))\}\end{aligned}$$

$mk\_Or(p, q) \to$

$$\begin{aligned}&\{(\sigma, \mathbf{true}) \mid (\sigma, \mathbf{true}) \in \mathcal{E}(p)\} \cup\\ &\{(\sigma, \mathbf{true}) \mid (\sigma, \mathbf{true}) \in \mathcal{E}(q)\} \cup\\ &\{(\sigma, \mathbf{false}) \mid (\sigma, \mathbf{false}) \in \mathcal{E}(p) \wedge (\sigma, \mathbf{false}) \in \mathcal{E}(q)\}\end{aligned}$$

$mk\_Exists(x, p) \to$

$$\begin{aligned}&\{(\sigma, \mathbf{true}) \mid\\ &\qquad \sigma \in \Sigma \ \wedge\\ &\qquad \mathbf{true} \in\\ &\qquad\quad \mathbf{rng}\, (\{\sigma \dagger \{x \mapsto i\} \mid i\colon \mathbb{Z}\} \lhd \mathcal{E}(p))\} \cup\\ &\{(\sigma, \mathbf{false}) \mid\\ &\qquad \sigma \in \Sigma \ \wedge\\ &\qquad \mathbf{rng}\, (\{\sigma \dagger \{x \mapsto i\} \mid i\colon \mathbb{Z}\} \lhd \mathcal{E}(p)) =\\ &\qquad\quad \{\mathbf{false}\}\}\end{aligned}$$

$mk\_FuncCall(f, al) \to$

$$\begin{aligned}&\{(\sigma, r) \mid\\ &\qquad \sigma \in \Sigma \wedge\\ &\qquad \forall i\colon \mathbf{inds}\ al \cdot (\sigma, vl(i)) \in \mathcal{E}(al(i)) \wedge\\ &\qquad (vl, r) \in \sigma(f)\}\end{aligned}$$

$mk\_PredCall(p, al) \to$

$$\begin{aligned}&\{(\sigma, r) \mid\\ &\qquad \sigma \in \Sigma \wedge\\ &\qquad \forall i\colon \mathbf{inds}\ al \cdot (\sigma, vl(i)) \in \mathcal{E}(al(i)) \wedge\\ &\qquad (vl, r) \in \sigma(p)\}\end{aligned}$$

Appendix B

Selected Mathematical VDM-SL Notation

This appendix provides details of a subset of the mathematical VDM-SL notation. The notation which is used in this thesis is based on this. This appendix is based upon the VDM Tools Language Manual [Cor10][1] for the ASCII VDM-SL notation, as well as [Jon90] and [JJLM91], which both present a subset of the mathematical VDM-SL notation.

B.1 Type Definitions

A type is defined as:

$$N = E$$

where N is the name of the datatype, and E are the values that belong to it.

B.1.1 Record Type

A record (composite) type is defined as:

$$\begin{aligned} N \;::\; & n_1 \;:\; T_1 \\ & \dots \\ & n_n \;:\; T_n \end{aligned}$$

where N is the name of the record, and each $n_i : T_i$ is a field with a name n_i and a type T_i.

The fields can be accessed through using the dot (.) notation, i.e. $\mathit{record}.n_i$.

A *mk* function is used to construct a record with appropriate values for each field:

$$\mathit{mk\_}N(e_1, \dots, e_n)$$

where e_i is a value of the corresponding type.

B.1.2 The Boolean Data Type

$\mathbb{B}$ is the set of Boolean values $\{\mathbf{true}, \mathbf{false}\}$.

The operators include, where p and q are Boolean-valued expressions:

[1]Available at: www.vdmtools.jp

$\neg p$	negation
$p \wedge q$	conjunction
$p \vee q$	disjunction
$p \Rightarrow q$	implication
$p \Leftrightarrow q$	iff
$p = q$	equality
$p \neq q$	inequality

B.1.3 The Numeric Data Types

The numeric data types include:

$\mathbb{N}$	natural numbers
$\mathbb{N}_1$	natural numbers excluding 0
$\mathbb{Z}$	integers
$\mathbb{Z}_1$	integers excluding 0
$\mathbb{R}$	real numbers
$\mathbb{R}_1$	real numbers excluding 0

where the operators available are the standard mathematical operators, e.g. $+$ and $\geq$.

B.1.4 The Set Type

A set is defined as:

$$S = T\textbf{-set}$$

where T is a type.

The operators include:

$\{\}$	the empty set
$\{e_1, \ldots, e_n\}$	set enumeration
$\{x \mid x : S \wedge P[x]\}$	a set comprehension
$\{1, \ldots, n\}$	a set of values from the range 1 to n
$x \in S$	in set (set membership)
$x \notin S$	not in set
$S_1 \cup S_2$	set union
$S_1 \cap S_2$	set intersection
$S_1 \setminus S_2$	set difference
$S_1 \subseteq S_2$	subset
$S_1 \subset S_2$	proper subset
card S	cardinality
$\mathcal{P}(S)$	powerset
$S_1 = S_2$	equality
$S_1 \neq S_2$	inequality

where $P[x]$ is a predicate that may make use of x, and x is a value of the appropriate type.

B.1.5 The Sequence Type

A sequence is defined as:

$$Sq = T^*$$

where T is a type.

The operators include:

$[]$	the empty sequence
$[e_1, \ldots, e_n]$	sequence enumeration
$[x \mid x : S \wedge P[x]]$	a sequence comprehension
hd Sq	the head element
tl Sq	a sequence of tail elements
len Sq	length
elems Sq	the set of elements
inds Sq	the set of indices
$Sq_1 \curvearrowright Sq_2$	sequence concatenation
$Sq(i)$	sequence application, where i is of the type $\mathbb{N}_1$
$Sq(i, \ldots, j)$	sub-sequence
$Sq_1 = Sq_2$	equality
$Sq_1 \neq Sq_2$	inequality

The first sequence subscript is 1.

B.1.6 The Map Type

A map is defined as:

$$M = T_1 \xrightarrow{m} T_2$$

where T_1 and T_2 are types.

The operators include:

$\{\}$	the empty map
$\{e_1 \mapsto c_1, \ldots, e_n \mapsto c_n\}$	map enumeration
$\{x \mapsto f(x) \mid x{:}\, S \wedge P[x]\}$	a map comprehension, where x maps to $f(x)$
$\textbf{dom}\, M$	the domain of the map
$\textbf{rng}\, M$	the range of the map
$M_1 \dagger M_2$	map override
$M(e)$	map application
$S \lhd M$	domain restrict
$M \rhd S$	range restrict
$M_1 = M_2$	equality
$M_1 \neq M_2$	inequality

B.1.7 The Union Type

A union type is defined as:

$$U = T_1 \mid \ldots \mid T_n$$

where U is the name of the type, and thus the type U contains all values of the types $T_1, \ldots, T_n$.

B.1.8 Pairs

The type of an ordered pair of values is defined as:

$$(T_1 \times T_2)$$

where T_1 and T_2 are types.

A pair is of the form (e_1, e_2).

Given a set of pairs $s = \{(e_1, e_2), (e_3, e_4)\}$, then:

$$\textbf{dom}\, s = \{e_1, e_3\}$$

and:

$$\textbf{rng}\, s = \{e_2, e_4\}$$

B.2 Expressions

B.2.1 Quantified Expressions

A universal quantification expression is defined as:

$$\forall x \colon S \cdot P[x]$$

An existential quantification expression is defined as:

$$\exists x \colon S \cdot P[x]$$

B.2.2 The Conditional Expression

A conditional expression is defined as:

$$\begin{array}{l} \textbf{if } e \\ \textbf{then } a \\ \textbf{else } b \end{array}$$

where e is a Boolean valued expression, and a and b are expressions of any type.

B.2.3 The Cases Expression

The cases expression is defined as:

$$\begin{array}{l} \textbf{cases } a \textbf{ of} \\ \ p_1 \to b_1 \\ \ldots \\ \ p_n \to b_n \\ \\ \textbf{others } b_{n+1} \\ \textbf{end} \end{array}$$

where a is an expression, p_i is a pattern matched against a, and b_i is an expression.

B.2.4 The Let Expression

There are two types of let expressions used. Firstly:

$$\textbf{let } e_1 = c_1, \ldots, e_n = c_n \textbf{ in } e$$

which is a local definition, and:

$$\textbf{let } x \in S \textbf{ in } e$$

which arbitrarily selects a value x from the set S.

B.3 Function Definitions

VDM contains both implicit and explicit function definitions. Only explicit function definitions are used in this thesis.

Explicit functions are defined as:

$$f : T_1 \times \ldots \times T_n \to T$$
$$f(e_1, \ldots, e_n) \quad \triangleq \quad \ldots$$

B.4 Inference Rules

An inference rule is defined as:

$$\frac{\textit{hypotheses}}{\textit{conclusion}}$$

where the *hypotheses* are separated by a ;. The *conclusion* holds when all of the *hypotheses* hold. This is the form of inference rule that is used in [BFL+94].

An inference rule can also be named:

$$\boxed{\textit{name}}\frac{\textit{hypotheses}}{\textit{conclusion}}$$

B.5 Proofs

Selected proofs in this thesis are written in the style that is used in [BFL+94].

Such proofs are of the form:

	from	*assumptions*	
1		*assertion*$_1$	justifications$_1$
2		*assertion*$_2$	justifications$_2$
		. . .	
	infer	*conclusion*	justifications$_{n+1}$

The use of **from** is to identify the assumptions, which are separated by a ;. Assumptions are referred to later in subsequent proof steps as hi, where i is the number of the assumption based upon the order that the assumptions are written in.

The use of **infer** is to identify a conclusion.

Steps within a proof are numbered, so that the assertions can be referred to later in subsequent proof steps by their number.

The justifications are references to rules to justify an assertion/conclusion, which are separated by a ,.

Proofs can contain subproofs:

```
      from $assumptions_1$
1           from $assumptions_2$
1.1               $assertion_{2.1}$                      justifications$_{2.1}$
1.2               $assertion_{2.2}$                      justifications$_{2.2}$
                  ...
            infer $conclusion_2$                         justifications$_2$
            ...
      infer $conclusion_1$                               justifications$_1$
```

An assumption hi from a subproof is referred to by $n.hi$, where n is the number of the subproof. Any assumptions that are given in a subproof are only in scope in that subproof, so for example, an assumption from assumptions$_2$ is not in scope for use in justifications$_1$.

Appendix C

Glossary

This glossary is by no means complete. Many terms that are used widely throughout this thesis are defined here. Terms that are not used throughout this thesis are defined as they arise in the main content of this thesis.

Big-Step Structural Operational Semantics

A semantic definition that shows how overall results are obtained, in contrast to a Small-Step Structural Operational Semantic definition.

Generally referred to as a natural semantics or a big-step semantics.

Clausal Form

A formula represented as a set of sets of literals. A set of clauses (implicit conjunction), where each clause is a set of literals (implicit disjunction).

Conjunctive Normal Form

A formula that is a conjunction of disjunctions of literals.

The defined domain of a function

The set of values to which a function may be applied, where the function will yield a defined result.

Denotational Semantics

A semantic definition whereby "the meaning of a program is modelled by mathematical objects that represent the effect of executing the constructs" [NN92].

Denotes a value

The value of the term (e.g. function application) for instance is defined.

The domain of a function

The set of values to which a function may be applied.

"gap"

The absence of a defined value, a term as used in [Bla80].

Monotonicity

A operator is monotone if it denotes a defined value, and such a value will still hold if any undefined operand was to become defined.

Non-denoting term

See undefined term.

Partial function

A function that may not yield a result for every member of its domain.

Partial term

See undefined term.

Prenex Normal Form

A predicate formula where all quantifiers occur which are then followed by the quantifier free part of the formula (known as the matrix).

Refutation

Proving the validity of a formula by refuting its negation.

Resolution rule

A rule that takes two clauses that contain contradictory literals and from this infers a new clause.

Small-Step Structural Operational Semantics

A semantic definition that describes how individual steps take place, in contrast to a Big-Step Structural Operational Semantic definition.

Generally referred to as a Structural Operational Semantics or a small-step semantics.

Strictness

A construct that is undefined if any of its operands are undefined, e.g. strict equality, is undefined if either of its operands are undefined.

Total function

A function that is defined on all values that are within its domain.

Undefined term

A term that does not denote a value.