INCONSISTENCY AND UNDERDEFINEDNESS IN Z SPECIFICATIONS

A THESIS SUBMITTED TO
THE UNIVERSITY OF KENT AT CANTERBURY
IN THE SUBJECT OF COMPUTER SCIENCE
FOR THE DEGREE
OF DOCTOR OF PHILOSOPHY.

By
Ralph Miarka
December 2002

Abstract

In software engineering, formal methods are meant to capture the requirements of software yet to be built using notations based on logic and mathematics. The formal language Z is such a notation. It has been found that in large projects inconsistencies are inevitable. It is also said, however, that consistency is required for Z specifications to have any useful meaning. Thus, it seems, Z is not suitable for large projects.

Inconsistencies are a fact of life. We are constantly challenged by inconsistencies and we are able to manage them in a useful manner. Logicians recognised this fact and developed so called paraconsistent logics to continue useful, non-trivial, reasoning in the presence of inconsistencies. Quasi-classical logic is one representative of these logics. It has been designed such that the logical connectives behave in a classical manner and that standard inference rules are valid. As such, users of logic, like software engineers, should find it easy to work with QCL.

The aim of this work is to investigate the support that can be given to reason about inconsistent Z specifications using quasi-classical logic. Some of the paraconsistent logics provide an extra truth value which we use to handle underdefinedness in Z. It has been observed that it is sometimes useful to combine the guarded and precondition approach to allow the representation of both refusals and underspecification.

This work contributes to the development of quasi-classical logic by providing a notion of strong logical equivalence, a method to reason about equality in QCL and a tableau-based theorem prover. The use of QCL to analyse Z specifications resulted in a refined notion of operation applicability. This also led to a revised refinement condition for applicability. Furthermore, we showed that QCL allows fewer but more useful inferences in the presence of inconsistency.

Our work on handling underdefinedness in Z led to an improved schema representation combining the precondition and the guarded interpretation in Z. Our inspiration comes from a non-standard three-valued interpretation of operation applicability. Based on this semantics, we developed a schema calculus. Furthermore, we provide refinement rules based on the concept that refinement means reduction of underdefinedness. We also show that the refinement conditions extend the standard rules for both the guarded and precondition approach in Z.

Acknowledgements

This thesis represents four years of work and four years of my life. Many people contributed in a variety of ways. I would like to thank everybody from the bottom of my heart. Unfortunately, I cannot mention all of you by name. Nevertheless, your contribution is not forgotten.

First and foremost I would like to thank both my supervisors Prof. John Derrick and Dr. Eerke Boiten for their guidance and support they have given me. Their availability throughout the last years helped me tremendously and their motivation has been a constant source of energy for me. I could not have done this without them.

Furthermore, I am very grateful to the Computing Laboratory of the University of Kent at Canterbury who funded me for the first three years with an E. B. Spratt Bursary and also provided me with the opportunity to teach in the undergraduate course. I am also very appreciative for the support I have received from the members of the department.

Parts of this work have been presented at the ZB User Meetings in 2000 and 2002. I would like to thank the anonymous referees for their thoughtful comments and all the attendants for their readiness to discuss parts of this work. I am also very grateful to Dr. Anthony Hunter for the inspiring discussions we had on quasi-classical logic.

I also thank the many people who I have had the pleasure to meet and who made my stay in Canterbury even more enjoyable. My friends, in particular, helped me not only to go through the work but also through life. I am not sure whether I could have done without their support and encouragement.

Last but not least, my heartfelt thanks go to my parents, Ernst and Mechthild, for their continuous support they have provided during all my life.

I owe you all very much. Danke!

Ralph Miarka
Canterbury, December 2002

Contents

List of Figures

List of Tables

Chapter 1

Introduction

> Ja, ich sage schon jetzt voraus: es werden mathematische Untersuchungen über Kalküle kommen, die einen Widerspruch enthalten, und man wird sich noch etwas darauf zugute tun, daß man sich auch von der Widerspruchsfreiheit emanzipiert.[1]
>
> Ludwig Wittgenstein
> 30^{th} December 1930

Software engineering is the branch of computer science that is concerned with the development of software. Its aim is to provide engineering methods and techniques to build and maintain software. An analogy commonly drawn is between architects and software engineers. In early stages houses were just built without a systematic knowledge of how to construct them. However, to build sky scrapers that will not collapse a deep mathematical understanding of the statics of such buildings was required. As such, only the formalisation of the methods in architecture allowed new developments.

Software engineering is undergoing a similar metamorphosis. Rather than building software in an ad hoc fashion, a deeper understanding of its requirements and its construction is needed to make software more reliable. Formal methods is the field of software engineering that is aimed at developing techniques to make the meaning of software artifacts mathematically and logically precise in order to improve software reliability.

Formal specifications are the main mathematical objects considered in formal methods. Unfortunately, it has been found that especially large specifications are often inconsistent. Consistency, however, is required for specifications to be meaningful. Taken together, this implies that large specifications are usually not meaningful. The aim of our research is to overcome this problem by handling inconsistencies in a more practical way.

[1] "Indeed, even at this stage, I predict a time when there will be mathematical investigations of calculi containing contradictions, and people will actually be proud of having emancipated themselves from consistency." (Wittgenstein, 1964, p. 332), English translation in (Priest, 2000).

1.1 Managing Inconsistency in Z Specifications

Formal methods are seen as the way forward to more reliable software. Their application in the development process leads to a deeper understanding of the requirements of the software under construction. One of the main objects considered by formal methods are formal specifications. They express the software requirements in terms of logic and mathematics. This foundation enables the formal analysis of the requirements and it provides a possibility to verify whether the requirements are met by the software product.

The development of a specification depends primarily on the sources of information, like designers, engineers and others. Often, several developers' views need to be incorporated into the description of the software product. It has been found that, in particular in large projects, the participants disagree on a range of issues. Furthermore, due to the complexity of large descriptions errors can easily appear. In general, it has been found that

"Inconsistencies are inevitable in large projects."

(Ghezzi and Nuseibeh, 1998)

The Z notation is one of several languages used to develop formal specifications. It is based on logic and mathematics, in particular set theory, and provides a rather elegant way of structuring the mathematics that describe the system at hand. However, considering the argument from above we face a practical problem, because

"Consistency is essential for a Z specification to have any useful meaning."

(Valentine, 1998)

The conjunction of both claims means that the Z notation is not suitable for large projects because they can be inconsistent but a Z specification including an inconsistency would be meaningless or useless. This conclusion is, however, not practical. As a matter of fact, even inconsistent specifications have a desired meaning and an intended use.

Inconsistencies are generally regarded as undesirable in software development and, in particular, in formal specification. A formal specification written in the Z notation is basically a logical description of a system and its behaviour, i.e. it is a logical theory. Logicians, however, often regard inconsistent theories as uninteresting because they allow to derive any conclusion within their language and therefore none can be trusted. This is also the reason for Valentine's claim above.

Because inconsistencies are seen as undesirable, researchers developed tools and techniques to remove inconsistencies as soon as, or soon after they are detected. Another approach is to follow guidelines to prevent the introduction of inconsistencies into specifications in the first place. This research is certainly valuable to minimise the occurrence of inconsistencies. At times, however, such an approach can be impractical.

Recently it has been acknowledged that in practice it is not always possible nor desirable to eradicate inconsistencies immediately, if at all. For example, the engineer who could decide on how to resolve the inconsistency may not be available. This would in turn bring the project to almost a standstill because the specification is considered useless. It could be that no-one knows how to resolve the inconsistency at all. Also, inconsistencies can be useful to guide the future development, pointing out areas that need more attention. Moreover, in particular in large projects, the removal of one inconsistency might bring up another and sometimes a completely consistent stage is unreachable in practice. Thus, we are required to manage inconsistencies in a more general fashion.

The Z notation is founded on standard predicate logic but we identified that inconsistencies cannot be handled appropriately by such a logic. Therefore, it seems natural to investigate other logical foundations for the Z notation. The group of logics that can be used to manage inconsistencies are called paraconsistent logics. The aim of our research is to investigate the formal support we can give to managing inconsistencies in Z specifications using a paraconsistent logic to facilitate useful formal reasoning in the presence of inconsistency.

1.2 Underdefinedness in Z Specifications

We found that there is a wide range of paraconsistent logics. Some of them capture inconsistency rather intuitively by providing an extra logical truth value, often called “both” in the semantics. Furthermore, many of these logics include another truth value, called “neither”, to denote incomplete knowledge. For example, if asked “Who is the current chancellor of Germany?” we can answer “I was told it is Mr. Schröder”, “I was told it is not Mr. Stoiber”, “I was told it is Mr. Schöder and I was told it is Mr. Stoiber” or “I do not know at all”. These four scenarios capture the idea of the four truth values.

An application area for this “don't know” value in formal specification is underdefinedness. This notion refers to those situations where the operation is applied outside its domain. In the common Z specification style operations are, in general, partial relations. The domains of these partial operations are traditionally called preconditions. Depending on the application area there are two possible interpretations of the result of applying an operation outside its domain.

In the traditional interpretation anything may result whereas in the alternative, guarded, interpretation the operation is blocked outside its precondition. It has been observed that it is often convenient to use a combination of the guarded and precondition interpretation to allow both modelling of refusals and underspecification.

1.3 Aims and Objective

We identified two interesting areas of research concerning the Z notation. On the one hand, we found that inconsistencies in Z specifications need to be managed in a more practical fashion, rather than being eradicated. On the other hand, modelling underdefinedness explicitly in the Z notation can be further explored.

Managing Inconsistency in Z using Quasi-Classical Logic

The problem is, that the Z notation cannot deal appropriately with inconsistent situations because it is founded on classical predicate logic. Classical predicate logic allows trivial inferences in the presence of inconsistency. Paraconsistent logics, on the other hand, allow only non-trivial inferences despite the presence of inconsistency. Therefore, it is our aim to investigate whether the Z notation can be founded on a paraconsistent logic to manage inconsistencies more appropriately.

Paraconsistent logics are, in general, weaker than classical logic in the sense that not all classically valid inferences are possible. This is achieved by non-standard behaviour of the logical connectives, by the introduction of new logical connectives, by disallowing established proof rules, like resolution, or by other means. Furthermore, properties of classical logic, like monotonicity or transitivity can fail. We need to find a suitable logic for our task, one that will be acceptable to both specification developers and specification analysts.

Once we have found an appropriate paraconsistent logic we are interested in its application to the analysis of Z specifications. Our aim is to avoid triviality in the presence of inconsistency which means that we opt for deriving less but more useful information. Refinement is concerned with the formal development of concrete specifications from abstract ones. We are interested in providing a meaning for refinement of inconsistent specifications. This should, on the one hand, facilitate the controlled removal of inconsistencies and, on the other hand, the process of living with inconsistencies in Z specifications.

Handling Underdefinedness in Z

The aim of this thesis with respect to underdefinedness is to develop a notation that combines both the guarded and the standard precondition interpretation to model underdefinedness explicitly. We decide to consider a three-valued semantics to capture the intuition that (1) an operation can be blocked by a guard, (2) that the operation can be allowed by the guard but no result has been defined and (3) the operation is applicable and its result is defined. Then we can use existing three-valued logics to investigate specifications based on such an interpretation.

The reduction of underdefinedness and non-determinism is a common goal of refinement. Given such a three-valued interpretation of the applicability of operations it is our aim to find suitable and intuitive refinement conditions to support further specification development. We identified that there are systems which require non-deterministic behaviour. Therefore, we are also interested in refinement that takes such behaviour into account.

The extensive use of schemas to structure specifications has made Z successful. The schema calculus provides a means to combine schemas and to reason about them. It is a further aim of our work to see whether we can construct a schema calculus suitable for the three-valued interpretation of the operations. Such a calculus should be as functional as the standard calculus, i.e. it should facilitate reasoning about the combination of schemas. Note, refinement calculations are also an application area of the schema calculus.

1.4 Contributions

There are several contributions to be found in this work. Essentially they can be grouped according to the notions of inconsistency and underdefinedness. The former consists of work on quasi-classical logic and its application to the analysis of Z specifications, while the latter refers to the work on a schema representation for underdefinedness based on a three-valued logic.

On Quasi-Classical Logic. In order to use quasi-classical logic to analyse Z specifications we were required to develop QCL further. On the one hand, the literature on QCL does not provide a general notion of logical equivalence for QCL. Such a notion is, however, necessary to facilitate the simplification of logical formulae. Therefore we investigate a number of different notions based on the QC consequence relation and QC model classes. Our work results in a strong notion of logical equivalence allowing general replacement of equal formulae.

On the other hand, QCL did not include the notion of equality. However, the use of equality is a common feature in Z specifications. Therefore, we incorporated reasoning about equality into QCL. This led also to the investigation of the one-point rule in QCL and we established it to be a valid reasoning rule. Finally, we developed an automated theorem prover based on the tableau method for QCL.

Quasi-Classical Logic and Z. Quasi-classical logic proved useful in the formal analysis of inconsistent Z specifications. We demonstrated that fewer but more useful inferences from inconsistent specifications are possible. Given the standard definition of a precondition but using QCL, we found a notion of applicability that is able to capture the intended application area of an inconsistently defined operation. This quasi-classical precondition is then used to investigate the process of refinement of inconsistent operations. The result is an applicability rule that prevents some "useless" refinements from inconsistent operations.

Guarded Precondition Schema. Based on a three-valued intuition of the applicability of an operation we developed a Z-like schema representation for both guards and preconditions in an operation thus enabling the specification of underdefinedness. Our schema representation is more expressive than previous developments by allowing after-state variables in the guard. This required the development of rules for calculating the implicit guard and precondition of an operation. Given those, we were able to provide a set of refinement rules for operations and showed that they extend the standard rules with respect to the guarded and precondition interpretation.

A Schema Calculus. To improve the usefulness of guarded precondition schemas we developed a schema calculus considering the standard Z schema operators. We were guided by our three-valued interpretation of the applicability of operations. The definition of most schema operators was straightforward. However, due to the non-classical interpretation, schema implication and entailment turned out to be different. We were, however, able to re-gain a suitable entailment operator to facilitate, for example, refinement calculations.

1.5 Outline

This thesis starts with a short introduction to Z followed by a discussion on inconsistency and underdefinedness in Z specifications. Then we present some insight into paraconsistent reasoning and, in particular, into quasi-classical logic. In the following chapters we apply these logics to reasoning about inconsistencies in Z and to develop a new semantics for handling underdefinedness in Z. Below, we give a more detailed description of the structure of this thesis.

Chapter 2. In Chapter 2 we introduce the Z notation. We provide some background from logic and set theory, including types, relations and functions, and we introduce Z schemas, the basic building blocks of a Z specification. We present the schema calculus as a means to structure Z specifications and to combine schemas. Furthermore, we consider the notion of refinement of Z specifications to develop abstract specifications into concrete ones. Throughout this thesis we use the support of tools which are presented in this chapter. Finally, we discuss briefly the relation of Z to some other formal specification notations.

Chapter 3. In Chapter 3 we describe the aim of our research in more detail. We are interested in the sorts of inconsistencies that can arise in Z specifications. We claim that inconsistencies can be a tool guiding the development of specifications and we look at desired inferences despite the presence of inconsistencies in Z specification. Underdefinedness can be considered to be closely related to inconsistency thus we introduce the concept of underdefinedness in Z specifications and we propose a new way to handle it.

Chapter 4. In Chapter 4 we introduce some background on the notion of paraconsistency, including the different motivations for paraconsistency, two definitions of paraconsistency and some of the approaches to construct a paraconsistent logic. Then we present two closely related four-valued paraconsistent logics, namely the logic *FOUR* by (Belnap, 1977b) and the logic $\mathcal{FOUR}$ by (Damásio and Pereira, 1998). A three-valued subset of the logic $\mathcal{FOUR}$ is used in Chapters 7 and 8 to provide the semantics for our work on underdefinedness. The main part of Chapter 4, however, is devoted to the introduction of quasi-classical logic by (Hunter, 2000) which plays a major role in the following chapter. We contribute to the development of QCL by investigating the notion of logical equivalence in QCL.

Chapter 5. In Chapter 5 we incorporate reasoning about equality into QCL. We introduce the syntax and semantics for equality, including the equality axioms and some investigation of using these axioms as extra assumptions in the reasoning process using QCL. Then we develop the machinery to reflect that we are dealing in fact with equality. We extend the proof system of QCL by extra tableau rules for handling equality and we prove their soundness and completeness. Finally, we present a version of the one-point rule for QCL to further facilitate QCL's applicability to our research.

Chapter 6. In Chapter 6 we bring together QCL and Z. We present a small example of a library system specified using the Z notation. We introduce an inconsistency into the example to use it as an illustration throughout this chapter.

In the classical setting such a specification would be meaningless but not so when using quasi-classical logic. We demonstrate that QCL allows fewer but more useful inferences than standard predicate logic. Then we apply QCL to the process of calculating the precondition of inconsistent operation schemas facilitating a discussion on the refinement process of inconsistent operations. Following the notions of standard refinement, we establish the principles of quasi-classical applicability and QC correctness and thus show that QCL can be used to control the continuous development of inconsistent specifications. Note, some parts of this chapter were previously published by (Miarka et al., 2002).

Chapter 7. In Chapter 7 we link up the Z notation, the problem of underdefinedness and the two interpretations of the meaning of a precondition in Z. We demonstrate by means of two examples, normalisation and a simple money transaction system, that a combination of the traditional and blocking interpretation is sometimes required. Then we define a schema notation including both guards and effect schemas. Based on that we define regions of operation behaviour, i.e. whether an operation is inside or outside the guard, or inside or outside the precondition. These regions can naturally be defined in a three-valued interpretation leading to a simple and intuitive notion of refinement that generalises standard operation refinement. We introduce these refinement rules and show their compatibility to the standard ones. Note, some parts of this chapter were previously published by (Miarka et al., 2000).

Chapter 8. In Chapter 8 we develop a schema calculus for such guarded precondition schemas. We start the chapter with a brief recapitulation of the notion of a guarded precondition schema and we continue with an illustration of its use by presenting a small example of a heat control system. It follows the main part consisting of the development of the schema calculus itself which is based on the standard schema operators. We validate the calculus by proving several laws for our schema operators. Furthermore, we show that some laws of the classical Z schema calculus do not hold within our calculus. We revise the standard notions of schema applicability and we return to investigating operation refinement, using the newly developed schema calculus.

Appendix A. In Appendix A we present work in progress on a tableau-based theorem prover for QCL. The theorem prover, called QC-LeanTaP, is based on leanT^AP which we briefly introduce first. Part of leanT^AP is a small program to calculate the conjunctive negation normal form of a first order predicate formula. We adapt this program for our needs by removing skolemization of existential predicates. Finally we present our tableau-based theorem prover for QCL.

Chapter 2

A Short Introduction to Z

Z is a formal specification notation. It is used to model a system by naming the components and to state the constraints upon them and their relations, thus describing the behaviour of the system. Z is formal in the sense that it uses mathematics, which consists basically of set theory and first-order predicate logic, to specify systems. This foundation enables mathematical reasoning to establish that desired properties are indeed consequences of specifications written in Z.

The main feature of Z, distinguishing it from many other formal notations, is the schema notation. It provides a very elegant way of structuring the mathematics specifying a system as well as to structure the system itself. The Z notation defines a schema calculus to combine schemas. It is also used to reason about the specification. This includes the ability to reason about the development of more concrete specifications from abstract ones, i.e. about refinement.

Z is a notation, not a method, although it is often said to be one. The Z standard (ISO/IEC 13568, 2002) does not say how to use Z in a systematic way and to what Z can be best applied. Neither does the Z standard give any guidance on how to develop a system from a Z specification. Note also that Z specifications are not executable nor, in general, can they be compiled into a running program. Hence, Z is not some kind of a programming language.

The Z notation has been used to specify different kinds of systems. Examples of applying Z successfully include safety critical systems, such as railway signalling and medical devices, security systems, like transaction processing systems, and general hard- and software developments. A comprehensive list of application examples as well as information on tools and other resources can be found on the Z notation home page: `http://www.comlab.ox.ac.uk/archive/z.html`.

The aim of this chapter is to give an overview of the Z notation and to introduce the necessary background to be able to describe those problems we will tackle in the next chapters. We present the schema notation, including the schema calculus, the most common conventions and the notion of refinement in Z. Finally, we discuss briefly tool support for Z and other methods related to Z.

2.1 Introduction

Z is a formal specification language based on Zermelo-Fraenkel set theory and first-order predicate logic. It provides a notation for describing the behaviour of a system using mathematics. The key feature of Z is its schema notation, a way to structure the mathematics elegantly. A Z specification not only consists of mathematical text but also of informal explanatory text, describing the meaning of the mathematical constructs. The purpose of the formality is to avoid ambiguities inherent in informal descriptions and to provide a basis for rigorous reasoning.

The Z notation includes an extensible toolkit of mathematical notation, a schema notation for specifying structures in the system and for structuring the specification itself and a decidable type system which allows extra checks to be performed to reduce the risk of specification errors. Furthermore, Z has a schema calculus for modifying and combining schemas. The schema operators enable the definition of new schemas using existing ones in a compact and readable way.

2.1.1 History of Z

The Z notation grew out of work by (Abrial, 1974) at Oxford University's Programming Research Group. Its development and recognition benefited greatly from being used at IBM UK Laboratories at Hursley Park for the re-specification, re-design and further development of their Customer Information Control System (CICS). (Nix and Collins, 1988) published one of the many studies on this project. (Barrett, 1989) reports on another important project at the time, the use of Z in the formalization of the IEEE standard for binary floating-point arithmetic which formed the basis for the floating-point unit of the Inmos IMS T800 Transputer. Both projects received the UK Queen's Award for Technological Achievement jointly with the Oxford University Computing Laboratory.

Two books helped primarily to establish Z and to stabilise the notation. (Hayes, 1987) edited a collection of case studies which where later substantially revised in (Hayes, 1993). This collection was used as a kind of a reference on how to use Z. Later, (Spivey, 1992) produced a reference manual which became the de facto language definition for many years. For some time now, the Z notation has been undergoing a standardization process. This effort resulted in the recent publication of the International Standard (ISO/IEC 13568, 2002) which "establishes the precise syntax and semantics for some mathematics, providing a basis on which further mathematics can be formalized."

Many books, like (Potter et al., 1991), are aimed at the introduction to formal specification and Z. (Barden et al., 1994), for example, provide some useful advice on how to use Z in practice. (Jacky, 1997) demonstrates the way of Z through a series of short studies, introducing the essential features of the notation quickly. (Woodcock and Davies, 1996) look more deeply at the development process based on Z specifications. This aim has been taken further by (Derrick and Boiten, 2001) who present a thorough account on refinement in Z and Object-Z (Smith, 2000), a notation closely related to Z. Common to all these books is their emphasis on understanding Z and making it available to a wider audience.

There is also a regular series of conferences, *ZUM: The Z Formal Specification Notation*, also known as the Z User's Meetings. These conferences are devoted to Z and similar specification notation. Recently conferences where held jointly with the B community. The last conference proceedings were edited by (Bowen et al., 1998), (Bowen et al., 2000) and (Bert et al., 2002).

2.1.2 Motivation

We choose Z for our work because it is a mature notation. It has a rich literature of introductory texts and case studies and it has been an object of research for many years. Z is among the first formal notations to make the crossover from academia to industry. It has been applied successfully in numerous industrial projects, and according to the companies saved them millions. With these industrial applications in mind Z underwent the ISO standardization process. Furthermore, Z is being widely taught, not only at universities.

One of the advantages of Z is that it can be used in a number of different ways according to the application area. This, however, leads to the problem of choosing the right way for the desired application. For example, we will see later in this work that there are at least two ways of interpreting the precondition in Z. The so called disadvantage of Z that it is not a method turns possibly into our favour. Z not dictating a method provides us with more flexibility to investigate Z, abstracting from methodological concerns.

The aim of this chapter is to introduce the Z notation. We focus in our presentation on the background necessary for the remainder of this thesis. The reader familiar with Z can safely skip this chapter as it provides no new insights into the Z notation. The short discussion on Z tools and on related specification methods might, however, be of additional value.

2.1.3 Outline

This chapter is structured as follows. In Section 2.2 we provide some background on logic and set theory, including types, relations and functions. In Section 2.3 we introduce Z schemas, the basic building blocks of a Z specification. Schemas can be combined appropriately using the schema calculus which we present in

Section 2.4. The notion of refinement of Z specifications is discussed in Section 2.5. The Z notation is also supported by tools. We present a selection of them in Section 2.6. Finally, in Section 2.7, we discuss briefly the relation of Z to some other formal specification notations. This chapter concludes with a short summary.

2.2 Logic, Sets, Types, Relations, Functions

The Z notation is based on set theory and first-order predicate logic. Although we assume general familiarity with these topics, we introduce some background notions frequently used in this work. We cover briefly the logic of Z and then we present some notation from set theory and its application to type theory, relations and functions. Note, that we provide only the terminology used in this work. For a detailed introduction we recommend one of the aforementioned textbooks.

2.2.1 Logic

The Z notation uses propositional and predicate logic to state the relationship between the components of a system and to constrain the behaviour accordingly. The propositional logic used contains the common connectives with their usual meaning and order of precedence: $\neg$ – negation, $\wedge$ – conjunction, $\vee$ – disjunction, $\Rightarrow$ – implication, and $\Leftrightarrow$ – equivalence.

Predicate logic is provided by the usual introduction of quantifiers into the language, together with the notions of free and bound variables. The Z notation is a typed language meaning that every variable belongs to a fixed set of values, thus quantifications need to be typed, too. For example, universal quantification has the form $\forall x : T \mid p \bullet q$ and means that for all x in T satisfying the predicate p, q holds. Existential quantification has the form $\exists x : T \mid p \bullet q$ and means that there exists at least one value of x in T satisfying p such that q holds.

The predicate p restricting q is optional. If p is omitted it is considered to be *true*. The following equivalences hold for the restricted quantifiers: for universal quantification $\forall x : T \mid p \bullet q \;\Leftrightarrow\; \forall x : T \bullet p \Rightarrow q$ and for existential quantification $\exists x : T \mid p \bullet q \;\Leftrightarrow\; \exists x : T \bullet p \wedge q$.

A variable introduced by a quantifier is said to be bound, and the usual scoping laws apply. Variables that are not bound in a predicate are said to be free. As usual, it is possible to replace all bound occurrences of a variable in a predicate. This ensures the correctness of the following frequently used proof rule of $\exists$-elimination, also called the one-point rule (for existential quantification): $\exists x : T \bullet x = t \wedge p(x) \;\equiv\; t \in T \wedge p(t)$, provided that x is not free in t. This law states that if we are required to demonstrate the existence of a variable and a suitable instantiation is given, then we can eliminate the existential quantifier. This law is often used in the simplification of preconditions of operations.

2.2.2 Sets

Set theory is the other cornerstone of the Z notation, in fact, the name Z is derived from Zermelo-Fraenkel set theory. Membership – $\in$ and its converse – $\notin$, empty set – $\varnothing$, subset – $\subseteq$, and equality – $=$ are defined as usual.

Sets can be given by listing their elements, like in $\{\mathit{red}, \mathit{green}, \mathit{yellow}\}$, or by set comprehension. For instance, $\{n : T \mid p\}$ is the set of all n in T satisfying the predicate p, e.g. $\{n : \mathbb{Z} \mid n \geq 0\}$ describes the set of all natural numbers. Furthermore, $\{x : S \mid P(x) \bullet Q(x)\}$ is the set of all x of type S satisfying the predicate P such that Q is satisfied, too. Note, $P(x)$ is omitted when $P(x) = \text{true}$ and $Q(x)$ is omitted when $Q(x) = \text{true}$. The size of a finite set is determined by its cardinality ($\#$), e.g. $\#\{\mathit{red}, \mathit{green}, \mathit{yellow}\} = 3$, considering all elements of this set are distinct.

Furthermore, we can use the common set operators, like power set construction – $\mathbb{P}$, Cartesian product – $\times$, set union – $\cup$, set intersection – $\cap$ and set difference – $\setminus$. These operators are all defined as usual. For example, $\mathbb{P}\, S$ is the set of all subsets of S, e.g. $\mathbb{P}\{\mathit{red}, \mathit{green}\} = \{\varnothing, \{\mathit{red}\}, \{\mathit{green}\}, \{\mathit{red}, \mathit{green}\}\}$, and the Cartesian product $S \times T$ is the set of ordered pairs whose first element is in S and whose second element is in T, e.g. $\{1, 2\} \times \{\mathit{red}, \mathit{green}\} = \{(1, \mathit{red}), (1, \mathit{green}), (2, \mathit{red}), (2, \mathit{green})\}$.

2.2.3 Types

Z is a typed language or, in logical terms, it is based on many-sorted first-order predicate calculus. Every expression in Z has a unique type assigned. Basically, types constrain the use of any kind of value. For example, when x is declared as $x : S$ then the type of x is the largest set containing S. Thus, types are sets and every set is contained in exactly one type. Note, however, that the symbol $\varnothing$ denotes the empty set of all possible types.

Types are important because they allow to detect a wide range of specification mistakes. For example, $(1, 2) \in \mathbb{N}$ is a type error in Z, because $(1, 2)$ is a tuple whereas $\mathbb{N}$ is a set of numbers, not of tuples. The type system of Z is decidable, thus it is possible to calculate automatically the types of expressions and to check whether they make sense. There are several tools (see Section 2.6) to support type checking.

Built-in Type. Z provides a single built-in type $\mathbb{A}$, called arithmos, supplying values for use in specifying number systems. For example, the integer numbers are defined as $\mathbb{Z} : \mathbb{P}\,\mathbb{A}$, thus the set of integers, $\mathbb{Z}$, is a subset of $\mathbb{A}$. The set of natural numbers is defined as $\mathbb{N} : \mathbb{P}\,\mathbb{Z}$, thus the number 7 is not of type $\mathbb{N}$ but of type $\mathbb{Z}$ and subsequently of type $\mathbb{A}$. The type $\mathbb{A}$ has been introduced by the current Z standard. Before, the set of integers, $\mathbb{Z}$, was considered to be the only given type and it is still common to consider $\mathbb{Z}$ as the "super-type" as done here.

Note, Z has no built-in Boolean type, though a type $\mathbb{B}$ consisting of *true* and *false* is, for illustrative purpose, frequently used. This, however, is strictly speaking a type error, because, in Z, *true* and *false* are defined as predicates, not expressions.

Given sets. Although Z provides only a single built-in type, a specifier has a number of ways to define new types relevant to the particular specification. One way is to simply declare them. A given set is a declaration of the form

$$[\mathit{TYPE}]$$

introducing a new type TYPE. For example,

$$[\mathit{NAME}, \mathit{BOOK}]$$

defines two new sets NAME and BOOK. At this stage, no further information about values or relationships between these sets are given.

Type construction. Starting with existing types there are various ways to construct new types. The power set operator $\mathbb{P}$ is an elementary type constructor often used. For example, the set $\{\mathit{alice}, \mathit{bob}, \mathit{charlie}\}$ is of type $\mathbb{P}\,\mathit{NAME}$, given that each of the names is in the set NAME, i.e. of type NAME. The Cartesian product is another frequently used type constructor. For example, $\mathit{NAME} \times \mathbb{N}$ is a type consisting of ordered pairs, e.g. $(\mathit{alice}, 2)$ is of type $\mathit{NAME} \times \mathbb{N}$.

Free types. Another important type constructor is the free type. Basically, free types can be transformed into other Z constructs. However, it makes it easier to describe certain structures, in particular recursive structures like lists and trees. Here, we only consider free types over constants. For example,

$$\mathit{Report} ::= \mathit{Ok} \mid \mathit{Failure}$$

denotes a type *Report* containing exactly two different constants *Ok* and *Failure*. Alternatively, this could have been defined by a given type $[\mathit{Report}]$ and the constraint $\mathit{Ok}, \mathit{Failure} : \mathit{Report} \mid \mathit{Ok} \neq \mathit{Failure} \land \forall x : \mathit{Report} \bullet x = \mathit{Ok} \lor x = \mathit{Failure}$. For more details on free type construction see (Spivey, 1992, pp. 82).

Another kind of type in the Z notation is the so called schema type, which we will introduce in Section 2.3.7 after presenting the notion of a schema.

2.2.4 Relations

Relations are among the most important and most extensively used mathematical constructs in Z. A relation is a set of ordered pairs. $X \leftrightarrow Y$ denotes the set of all relations between the sets X and Y, that is, the set of all sets of ordered pairs whose first elements are members of X and whose second elements are members of Y. $X \leftrightarrow Y$ is defined as $\mathbb{P}(X \times Y)$. When defining relations, the maplet notation $x \mapsto y$ is often used for (x, y).

Assume that our sets of names contains $\{alice, bob, charlie\} \subseteq NAME$. Then we can define a relation *letters* describing the number of letters in the name, e.g. $letters == \{alice \mapsto 5, bob \mapsto 3, charlie \mapsto 7\}$.

For any ordered pair first and second component projection, denoted *first* and *second* are provided. For example, $first\ (alice, 5) = alice$ and $second\ (bob, 3) = 3$. The domain of a relation $R : X \leftrightarrow Y$ is the set of first components of the ordered pairs in R, i.e. $\operatorname{dom} R = \{p : R \bullet first\ p\}$. The range of the relation R is the set of second components of the ordered pairs in R, i.e. $\operatorname{ran} R = \{p : R \bullet second\ p\}$. For example, given the relation *letters* we have $\operatorname{dom} letters = \{alice, bob, charlie\}$ and $\operatorname{ran} letters = \{3, 5, 7\}$.

Often, it is useful not to consider the whole of the domain or range of a set but restricted subsets. The domain restriction of a relation $R : X \leftrightarrow Y$ by a set $S : \mathbb{P}\, X$, denoted $S \lhd R$, is the set of pairs in R whose first components are in S, i.e. $S \lhd R = \{r : R \bullet first\ r \in S\}$. For example, $\{alice, charlie\} \lhd letters = \{alice \mapsto 5, charlie \mapsto 7\}$. The domain anti-restriction, or domain subtraction, of a relation $R : X \leftrightarrow Y$ by a set $S : \mathbb{P}\, X$ is the set of pairs whose first components are not in R, i.e. $S \ntriangleleft R = \{r : R \bullet first\ r \notin S\}$. Similarly defined are range restriction and range subtraction of a relation $R : X \leftrightarrow Y$ by a set $T : \mathbb{P}\, Y$, denoted $R \rhd T$ and $R \ntriangleright T$ respectively, but with respect to the second component of R.

It is often useful to specify that a relation only changed marginally. Applications of such operation include, for example, database updates. For a relation this means to replace some of the pairs by new ones. The operation to do this is called overriding. If R and S are both relations between X and Y, the relational overriding of R by S is the whole of S together with those members of R that have no first components that are in the domain of S, i.e. $R \oplus S = ((\operatorname{dom} S) \ntriangleleft R) \cup S$. For example, $letters \oplus \{alice \mapsto 6\} = \{alice \mapsto 6, bob \mapsto 3, charlie \mapsto 7\}$. Note, if the domains of the relations R and S are disjoint then overriding coincides with set union, e.g. $letters \oplus \{dan \mapsto 3\} = letters \cup \{dan \mapsto 3\} = \{alice \mapsto 5, bob \mapsto 3, charlie \mapsto 7, dan \mapsto 3\}$.

There are many more operators on relations defined in the Z standard. Arguably, there are even more important operators than the presented ones. However, we have only introduced those that will be valuable to us subsequently. We refer to the aforementioned textbooks for more information.

2.2.5 Functions

Functions are relations with particular properties, namely that each element in the domain is mapped to at most one element of the range. Therefore, the operators above and all the other relational operators are all defined for functions, too. There are different kinds of functions distinguished by further properties. Each kind of function has a name and a symbol assigned.

The set of all partial functions $X \pfun Y$ from X to Y is the set of all relations between X and Y such that each x in X is related to at most one y in Y. Basically, the terms "function" and "partial function" are used synonymously. A function f from X to Y is said to be total, denoted $f : X \to Y$, if $\operatorname{dom} f = X$, i.e. if it relates each member of X to exactly one member of Y. For example, we can write $count : NAME \to \mathbb{N}$ for a function $count$ such that $count(n)$ returns the numbers of letters in a given name n, or $names : \mathbb{N} \pfun \mathbb{P}\, NAME$ for a function that returns all the names of a given length. Every name has a number of letters it consists of, hence $count$ is total but there is at least one natural number such that there cannot be a name of that length, hence $names$ is partial.

Functions have additional properties. They can be injective, surjective or bijective. A function from X to Y is injective, if each y in Y is related to no more than one x in X. A function from X to Y is surjective, if its range is equal to Y. A function is bijective, if it is both injective and surjective. Thus, $count$ is a total injective function and $names$ is a partial surjective function.

This concludes our introduction to some basic background. We introduced the syntax of the logic of Z and some notation from set theory. We covered Z's type constructors as well as the use of relations and functions in Z. Next we turn to the main feature of Z to structure specifications.

2.3 Schemas in Z

The Z specifications we consider will be written in the (usual) "states-and-operations" style. In this style a system is given by operations describing the change of the state of the system. The state of the system and the operations upon it are written using Z schemas structuring the specification into manageable components.

Schema boxes are the most recognizable feature of Z. They provide a structuring mechanism for the powerful mathematical language introduced above. Basically, the specification of a particular operation can be written as one predicate. However, it would be rather difficult to understand the meaning of such a predicate at once. Therefore, it is useful to break it into smaller, manageable, components. That is what schemas are for.

2.3.1 Schema Syntax

A schema consists of a set of declarations and constraints upon them. For example, the schema

Library

$users : \mathbb{P}\, NAME$
$borrowed : NAME \pfun \mathbb{P}\, BOOK$

$users \subseteq \operatorname{dom} borrowed$

introduces *users* which are a collection of something called *NAME* and *borrowed*, a function that assigns to something from the set *NAME* a subset of whatever the set *BOOK* contains. Furthermore, the predicate constrains the set *users* to be a subset of the domain of the function *borrowed*.

In general, a schema box consists of a schema name, a set of declarations above a short line, and a predicate below.

SchemaName

declaration

predicate

The declarations can be split across lines, like above, or they may be put on the same line, separated by semicolon. A predicate split across lines denotes a conjunction, unless another operator is used. For example,

$Example_1$

$n : \mathbb{Z};\ x : \mathbb{Z}$

$n < 5$
$x > 10$

$Example_2$

$n : \mathbb{Z}$
$x : \mathbb{Z}$

$(n < 5) \lor$
$(x > 10)$

the predicate in $Example_1$ means $(n < 5) \land (x > 10)$ whereas the predicate in $Example_2$ stands for $(n < 5) \lor (x > 10)$. We also use indentation to structure predicates appropriately.

Note, that the predicate can be *true*. Then it is omitted from the schema and the schema only provides the declarations. For example, the schema

System

$message : Report$

introduces something named *message* of type *Report*, i.e. something that can be *Ok* or *Failure*, without any further constraints attached.

Schemas can also be written in horizontal form, e.g.

$$OkReport == [message : Report \mid message = Ok]$$

describes that the thing *message* of type *Report* should be assigned *Ok*. The horizontal notation is used for two reasons. On the one hand, the naming of the schema is made more explicit and, on the other hand, they are more compact in notation.

In general, Z schemas are accompanied by a description in natural language to clarify the meaning of the schema. For example, the schema *Library* describes a simple library systems consisting of users who can borrow books. Unless the natural description is given all the components of a schema can be interpreted quite freely, they are only symbols.

2.3.2 Axiomatic Schemas

Axiomatic schemas are used to introduce new objects into a specification which are subject to constraints. These objects will be known throughout the specification, i.e. they are global. For example, the schema

$$\begin{array}{|l}
heat_max, heat_min : \mathbb{Z} \\
\hline
heat_max = 65 \\
heat_min = 45
\end{array}$$

introduces two global constants *heat_max* and *heat_min* of integer type with unique values assigned. In general an axiomatic schema looks like

$$\begin{array}{|l}
declaration \\
\hline
predicate
\end{array}$$

Again, the predicate is optional. If it is not given, it is considered to be set to *true*. An axiomatic schema without a predicate just introduces new global names.

Free types, as introduced above, are formally defined using axiomatic schemas. The earlier definition of

$$Report ::= Ok \mid Failure$$

is an abbreviation for

$[Report]$

$$
\begin{array}{|l}
Ok, Failure : Report \\
\hline
Ok \neq Failure \\
\forall x : Report \bullet x = Ok \lor x = Failure
\end{array}
$$

2.3.3 Generic Schemas

We said already that the symbol $\varnothing$ denotes the empty set for all possible types, thus the symbol $\varnothing$ is defined generically, that is, it has a definition using type parameters. For example,

$$
\begin{array}{|l}
[X] \\
\hline
makesum : (X \leftrightarrow \mathbb{Z}) \rightarrow \mathbb{Z} \\
\hline
\forall x : X;\ y : \mathbb{Z};\ z : (X \leftrightarrow \mathbb{Z}) \bullet \\
\qquad makesum\ \varnothing = 0 \land \\
\qquad makesum\ \{(x, y)\} = y \land \\
\qquad makesum\ (\{(x, y)\} \cup z) = y + makesum\ (z \setminus \{(x, y)\}) \\
\hline
\end{array}
$$

defines a function *makesum* that can take any set of pairs, where the first component is generic but the second component is an integer. The function *makesum* then calculates the sum of all the second components, regardless of what the first components are.

The advantage of generic schemas is their re-usability. Once defined, they apply to many different situations. For example, most operators on sets are defined generically, so that the type of the elements does not matter. However, when using such generic definition at a later stage in the specification, actual sets must be provided to replace the type parameter. Replacing the generic parameter by actual sets is called instantiation. Sometimes the actual sets can be inferred from the context, in some circumstances they must be provided explicitly. In any case, the value for the generic parameter must be clear.

2.3.4 Schema Inclusion

A schema can be included in another schema to form a composed schema. This approach supports structuring of specifications. For example, we define a schema with extra restrictions, like

$$\begin{array}{|l}
\hline
\textit{RestrictedLibrary} \\
\textit{Library} \\
\hline
\forall u : \textit{users} \bullet \#(\textit{borrowed}(u)) \leq 7 \\
\hline
\end{array}$$

by including the schema *Library* and imposing the condition that no user can have more than 7 books on loan. Such a schema is equivalent to one obtained by expanding all declarations and conjoining all predicates, e.g.

$$\begin{array}{|l}
\hline
\textit{RestrictedLibrary} \\
\textit{users} : \mathbb{P}\, \textit{NAME} \\
\textit{borrowed} : \textit{NAME} \mapsto \mathbb{P}\, \textit{BOOK} \\
\hline
\textit{users} \subseteq \operatorname{dom} \textit{borrowed} \\
\forall u : \textit{users} \bullet \#(\textit{borrowed}(u)) \leq 7 \\
\hline
\end{array}$$

Similarly, we can create a new schema by schema inclusion and additionally providing new components and constraints on them. The entire schema then consists of the expansion of the included schema together with the new components and the conjunction of all the predicates.

2.3.5 Decorations and Conventions

In this subsection we record some of the conventions of notation that are often used when writing Z specifications. These conventions include the identification of before and after states, operations on those states and input and output variables. The conventions are permitted but not enforced by the Z standard, though they are documented in it, too.

Primed Variables. Each operation in Z is described as a relation between states, namely the before and after state of the operation. It is therefore necessary to distinguish between the values of state variables before the operation and their values afterwards. The convention in Z is to use unprimed variables, like x, to denote values before the operation and to decorate variables with a dash, like x', to denote values after the operation. Note, however, that the schema predicate can also refer to any global constants.

Primed Schemas. Variables have to be in scope of the operation. If the state has been described in a schema S, then including S in the declaration part of the operation schema brings the state variables into scope. The after-state variables are similarly introduced by including S'. This is a schema obtained from S by decorating every variable in the signature of S with a dash, and replacing every occurrence of such a variable in the predicate part of S by its dashed counterpart. Thus, operations can be described in Z by a schema of the form

Op

S

S'

$\ldots$

Note, the variables from the signature of S are the only ones which are primed. Global constants, types etc. remain unprimed. If S contains a variable which has already been decorated in some way, then an extra dash is added to the existing decoration.

Delta. The inclusion of primed and unprimed copies of the state schema is so common that abbreviation for its use are introduced. The abbreviation $\Delta S == [S, S']$ is used to denote the general inclusion of primed and unprimed state schema, thus the operation schema becomes

Op

ΔS

$\ldots$

For example,

$AddUser$

$\Delta Library$

$name? : NAME$

$name? \notin users$

$users' = users \cup \{name?\}$

$borrowed' = borrowed$

This use of Δ is only a convention. Occasionally some authors like to include additional restrictions in their Δ-schemas, for example that a particular state component never changes. For instance, if S contained a component z, but none of the operations ever changed z, then ΔS could be defined by $\Delta S == [S, S' \mid z' = z]$, thus making it unnecessary to include $z' = z$ in each operation description. Note, however, that we will not use this feature here.

Xi. When enquiry operations, like reading variables, are being described, it is often necessary to specify that no change of state should occur. With the current notation this has to be done explicitly by stating for each component that its after-state value is the same as its before-state value. This is inconvenient and can be avoided using the Ξ-convention. Unless it has been explicitly defined to mean something else, references to ΞS are treated as being equivalent to $[S, S' \mid \theta S = \theta S']$, where the meaning of θ is explained below.

Inputs and outputs. Often, it is convenient to describe relations between inputs and outputs as well. The input values of an operation are provided by 'the environment', and the outputs are returned to the environment. Commonly an additional suffix is used to distinguish a variable intended as an input (?) or an output (!), thus for example, *name*? denotes an input, and *result*! denotes an output.

2.3.6 Normalisation

Earlier, we introduced the Z type system. We mentioned that a type can be constructed from a given type by constraining it. Normalisation is the process of making such constraints explicit. Schema normalisation will produce an equivalent schema where all components are declared to be members of their "maximal" type, rather than of a set contained in those. Consider a schema S with components $x_1 : X_1; \ldots; x_n : X_n$, such that the type of x_i is T_i. The normalisation of S is obtained by replacing all declarations of $x_i : X_i$ by $x_i : T_i$ and conjoining $x_i \in X_i$ with the predicate of S.

For example, the normalisation of the schema $S1$ is given by the schema $S2$.

$S1$

$$a, a' : \mathbb{N}$$

$$(a')^2 \leq a < (a' + 1)^2$$

$S2$

$$a, a' : \mathbb{Z}$$

$$a \in \mathbb{N} \wedge a' \in \mathbb{N}$$
$$(a')^2 \leq a < (a' + 1)^2$$

Schema normalisation plays an important role when combining schemas using the schema calculus.

2.3.7 Schemas as Types

So far we have not made explicit the meaning of a schema. Basically, a schema denotes a set which is contained in some type. The elements of such a set are called bindings. The type of these bindings is the signature of the schema, which, viewed as a set, is the largest set of bindings containing all elements of the schema.

There is a special operator to construct bindings in a context where all the component names are declared. This is the θ-operator. For example,

$$\theta Library = \langle\!| \ users == users, borrowed == borrowed \ |\!\rangle$$

The two occurrences of the names have rather different meanings. The first is local to the binding, just the name of a schema component. The second must refer to a value, namely the value of the variable of that name which must be in context. For example, when applying θ to a decorated schema, like

$$\theta Library' = \langle\!| \ users == users', borrowed == borrowed' \ |\!\rangle$$

it becomes evident that the first name is local and thus not subject to the decoration.

A common use of the θ-operator is to turn an operation into a relation between states. If we have an operation Op on $\Delta State$, then its relational interpretation is given by the set comprehension

$$\{Op \bullet (\theta State \mapsto \theta State')\}$$

This means that for each possible binding of Op a pair consisting of the included bindings of the before state $State$ and those for the after state $State'$ is included. Thus, each operation can be easily interpreted as a relation of before and after states. For example, given the operation $succ == [n, n' : \mathbb{N} \mid n' = n + 1]$ then its relational interpretation is the set of pairs $\{(\langle\!| \ n == 0 \ |\!\rangle, \langle\!| \ n == 1 \ |\!\rangle), (\langle\!| \ n == 1 \ |\!\rangle, \langle\!| \ n == 2 \ |\!\rangle), (\langle\!| \ n == 2 \ |\!\rangle, \langle\!| \ n == 3 \ |\!\rangle), ...\}$.

2.4 The Z Schema Calculus

The main building blocks of a Z specification are schemas. They are used to structure the specification and the systems under consideration. Much of the power of the Z notation derives from the ability to combine schemas. We already witnessed schema inclusion as such a construct. The Z notation, however, provides more operators to combine schema, some of which we present below. The collection and the use of these operators is called the schema calculus.

Combining schemas is subject to one restriction, namely that their declarations are compatible. This includes that the same names are used for the same meaning and, mostly, that the schemas are normalised. Remember, a type definition implicitly contributes not only to the declaration but also to the predicate of the schema.

In this section we consider the application of the schema operators to at most two schemas. This is not a restriction as the operators can be applied successively. For illustrative purpose we use the schemas $U == [Decl_U \mid pred_U]$ and $V == [Decl_V \mid pred_V]$ with their declaration and predicate part.

2.4.1 Renaming

Renaming schema components is another way to achieve the compatibility of the schema declarations. Schema components can be renamed, provided that the new name is not part of the declaration of the schema. The renaming of a component p by a q in a schema U is denoted $U[q/p]$, thus every occurrence of p will be replaced by q, except if p is bound inside the predicate of the schema. For example, we have

$$
\begin{array}{|l}
\hline
\textit{RestrictedLibrary}[\textit{members}/\textit{users}] \\
\hline
\textit{members} : \mathbb{P}\,\textit{NAME} \\
\textit{borrowed} : \textit{NAME} \rightarrow\!\!\!\!\!\!+\; \mathbb{P}\,\textit{BOOK} \\
\hline
\textit{members} \subseteq \operatorname{dom} \textit{borrowed} \\
\forall\, u : \textit{members} \bullet \#(\textit{borrowed}(u)) \leq 7 \\
\hline
\end{array}
$$

2.4.2 Schema Negation

For any schema U, the schema negation $\neg\, U$, is obtained by keeping the declaration of U and negating the predicate, i.e.

$$\neg\, U == [\textit{Decl}_U \mid \neg\, \textit{pred}_U]$$

Note, schema negation requires normalisation. For example, the negation of $U1 == [x : \mathbb{N} \mid \textit{pred}(x)]$ is $[x : \mathbb{Z} \mid x \notin \mathbb{N} \wedge \neg\, \textit{pred}(x)]$ for some predicate *pred* containing x.

Schema negation on its own is not often used in practice. However, it can play its part in simplifying schema expression when applying schema conjunction and schema disjunction. The schema calculus, like predicate logic, obeys the de Morgan laws and thus some schema simplifications can be expressed using schema negation.

2.4.3 Schema Conjunction

Schema conjunction is closely related to schema inclusion. The schema resulting from the conjunction of the schemas U and V contains both U and V and nothing else, thus

$$U \wedge V == [U;\ V] == [\textit{Decl}_U;\ \textit{Decl}_V \mid \textit{pred}_U \wedge \textit{pred}_V]$$

i.e. the predicates of U and V are conjoined and the declarations are merged appropriately. Schema conjunction does not need normalisation due to the properties of conjunction and normalisation.

However, it is only well-defined when components have compatible types. If the same variable is declared in both schemas but belongs to different sets, then the intersection of those sets needs to be taken. For example, $[x : \mathbb{N}] \land [x : \{-1, 1\}] == [x : \mathbb{N} \cap \{-1, 1\}] == [x : \{1\}]$. If the sets are not compatible, like in $[x : \mathbb{N}]$ and $[x : NAME]$, then the intersection is empty and thus schema conjunction is undefined.

Schema conjunction allows one to specify different aspects of a system separately. It can be usefully applied both on operation and on state schemas to combine those aspects to form a complete description, thus it is used to combine requirements.

For example, the schema $OkOp$ describes that an operation has been successful and it is defined by $OkOp == [message! : Report \mid message! = Ok]$. Then expanding $OkAddUser == AddUser \land OkOp$ is the schema

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{OkAddUser}} \\ \hline
\Delta Library \\
name? : NAME \\
message! : Report \\ \hline
name? \notin users \\
users' = users \cup \{name?\} \\
borrowed' = borrowed \\
message! = Ok \\ \hline
\end{array}
$$

2.4.4 Schema Disjunction

Schema disjunction is rarely used on state schemas. It is often applied on operation schemas to handle separate cases, in particular error handling and other exceptions, thus to develop total operations, i.e. operations that have no constraints upon their applicability. For example, given the operation $OkAddUser$ and the following schema

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{FailAddUser}} \\ \hline
\Xi Library \\
name? : NAME \\
message! : Report \\ \hline
name? \in users \\
message! = Failure \\ \hline
\end{array}
$$

reporting a $Failure$ if the given $name?$ is already contained in the set users, then combining both via disjunction results in a total operation, i.e. $TotalAddUser == OkAddUser \lor FailAddUser$.

Schema disjunction is constructed similarly to conjunction, i.e. combine the declarations and apply disjunction to the predicates, thus schema disjunction for two schemas U and V is defined as

$$U \lor V == [Decl_U;\ Decl_V \mid pred_U \lor pred_V]$$

provided both schemas U and V are normalised. This is necessary to ensure that common component names have not only compatible but identical types. This requirement also follows meta-theoretically because we required normalisation for schema negation and schema disjunction can be expressed in terms of schema conjunction and schema negation.

2.4.5 Schema Implication and Equivalence

Schema implication and equivalence have the usual meaning. They are defined as

$$U \Rightarrow V == \neg\, U \lor V$$

provided the schemas U and V are normalised and

$$U \Leftrightarrow V == U \Rightarrow V \land V \Rightarrow U$$

Both operators are rarely used to combine schemas. However, they prove useful to validate refinement conditions or other relations between operations. For example, for two operations Op_1 and Op_2 on the same state whose only component is $x : X$, the predicate $\forall\, x, x' : X \bullet Op_1 \Rightarrow Op_2$ states that the effect of Op_1 is consistent with Op_2 and $\forall\, x, x' : X \bullet Op_1 \Leftrightarrow Op_2$ states that the effects of both operations are identical. Note, that we quantify over the schema component, which is explained next.

2.4.6 Schema Quantification

The schema quantification of a schema U results in a new schema V containing a subset of the components of U in its declaration, with a predicate that is obtained from U by quantifying over the removed components. Quantification is used to express universal or existential properties of the given schema, like in refinement or in precondition calculation.

Existential Quantification. Given a schema $U == [x : X;\ Decl_U \mid pred_U]$ where $Decl_U$ consists of declarations but for $x : X$, then the existential quantification over x in U is

$$\exists\, x : X \bullet U == [Decl_U \mid \exists\, x : X \bullet pred_U]$$

Thus, $\exists\, x : X \bullet U$ is a schema on all components of U except x. Examples of the value and usage of existential quantification in Z are given below.

Universal Quantification. It is also possible to universally quantify over schemas. This happens less frequently than existential quantification but proves valuable when considering refinement. Given a schema $U == [x : X;\ Decl_U \mid pred_U]$ where $Decl_U$ consists of declarations but for $x : X$, then the universal quantification over x in U is

$$\forall\, x : X \bullet U == [Decl_U \mid \forall\, x : X \bullet pred_U]$$

Thus, $\forall\, x : X \bullet U$ is a schema on all components of U but x.

2.4.7 Schema Hiding, Projection and Composition

The following three schema operators are defined using schema quantification and possibly other schema operators. They are abbreviations to ease the construction of specifications.

Schema Hiding. Hiding of variables $(x_1 : X_1, \ldots, x_n : X_n)$ from a schema U, denoted $U \setminus (x_1, \ldots, x_n)$, is basically identical to existential quantification as such that $U \setminus (x_1, \ldots, x_n)$ stands for the existential quantification of the schema U over the components x_1 to x_n, i.e.

$$U \setminus (x_1, \ldots, x_n) = \exists\, x_1 : X_1, \ldots, x_n : X_n \bullet U$$

Schema Projection. Schema projection of a schema U on a schema V, denoted $U \upharpoonright V$, combines the schemas using conjunction but hides all components from U except those that are part of V. Formally,

$$U \upharpoonright V = (U \wedge V) \setminus (x_1, \ldots, x_n)$$

where $(x_1, \ldots, x_n)$ are components of U not shared by V.

Schema Composition. This operation describes the effect of one operation followed by another, i.e. it is an operation that begins in the before state of an operation Op_1 and ends in the after state of an operation Op_2. It is only meaningful when applied to operation schemas on the same state. The schema composition of two operations Op_1 and Op_2 is denoted $Op_1 \fatsemi Op_2$.

For example, consider the specification of the cursor movement in an editor buffer given by (Jackson, 1995). The operations *csrRight* and *csrLeft* both operate over the state *File* which represents the buffer. Consider the operation *csrRight* is applicable, then applying *csrLeft* after *csrRight* should result in the same position of the cursor as before, i.e. $csrRight \fatsemi csrLeft = \Xi File$. Thus, composition can also be used to validate the usefulness of some definitions in the specification.

Consider $State'$ to be the state after the operation Op_1 was performed. This is also the state immediately before operation Op_2. Lets call this intermediate state $State''$. Then composition is defined as

$$\begin{aligned} Op_1 \fatsemi Op_2 = \exists\, State'' \bullet \\ &(\exists\, State' \bullet [Op_1;\ State'' \mid \theta State' = \theta State'']) \land \\ &(\exists\, State \bullet [Op_2;\ State'' \mid \theta State = \theta State'']) \end{aligned}$$

which is the conjunction of both operations where the intermediate state is hidden. Schema composition can be calculated using renaming and hiding, e.g.

$$Op_1 \fatsemi Op_2 = (Op_1[x''/x'] \land Op_2[x''/x]) \setminus (x'')$$

for all state components. Note, schema composition does not connect inputs and outputs of an operation, which is called piping but not discussed here.

2.4.8 Precondition Calculation

The precondition of an operation characterises all the states and inputs to which the operation can be applied such that there is an after state and output which are related to the states and inputs by the operation. In some specification languages, like VDM (Jones, 1990), preconditions and postconditions are given explicitly. However, this does not apply to Z. In order to make a precondition of a given operation explicit one needs to calculate it.

The precondition, pre Op, of an operation $Op == [\Delta State;\ ins?;\ outs! \mid pred]$ on a state *State* with inputs $ins?$ and outputs $outs!$ is defined by

$$\mathrm{pre}\ Op = \exists\, State';\ outs! \bullet Op$$

Thus, pre Op is another schema on *State* and $ins?$, indicating on which before states and inputs the operation is applicable. The precondition is, based on this definition, a rather abstract predicate. This predicate is usually simplified applying, for example, the one-point rule and other equivalences. An algorithm for calculating a precondition is given by (Woodcock and Davies, 1996, pp. 206).

For example, the precondition for the operation $AddUser$ is $\mathrm{pre}\, AddUser = \exists\, Library' \bullet AddUser$, which can be simplified to the schema $[Library;\ name? : NAME \mid name? \notin users]$.

Discussing the issue of the precondition leads also to consider the notion of a postcondition. Note, Z does not use a single characterisation of the postcondition of an operation. However, in order to apply the refinement calculus (King, 1990), a notion of postcondition was adapted. Given an operation schema $Op == [\Delta State \mid pred]$ satisfying the condition $pred \Rightarrow \mathrm{pre}\, Op$, and a condition P, then P is considered to be a postcondition of Op if $\mathrm{pre}\, Op \land P \Leftrightarrow pred$. In particular this holds if P is equivalent to $pred$ itself, however, other valid postconditions may exist. The notion $\mathrm{post}\, Op$ is used to refer to some possible postcondition of Op.

2.5 Refinement in Z

So far we are able to write a formal specification in the Z notation. While this is a valuable task in its own right we also want to be able to develop a specification towards an implementation. The process of development from an abstract specification towards a more concrete representation is called refinement. To (Woodcock and Davies, 1996), refinement is all about improving specifications. It involves the removal of non-determinism, or uncertainty. A refinement is said to be acceptable provided it is impossible for an observer to notice the replacement.

2.5.1 Operation Refinement

(Derrick and Boiten, 2001) use the term *simple refinement* to describe the refinement of operations where the state schema does not change. This notation is commonly considered as operation refinement. However, *simple* refinement is a more general concept than operation refinement.

Operations in Z are, basically, binary relations over a state space relating a before state and an after state. Operations can be, if necessary, interpreted as total relations. Figure 2.1 shows two graphical representations of the operation $Op = \{(0,0),(0,1),(2,2)\}$ over the state $\{0,1,2\}$. The dotted lines represent the application of the operation for before states that are outside the domain.

Basically, there are two interpretations possible for applying an operation outside the domain. The first graph represents the contractual interpretation in Z, whereas the second one considers the blocking interpretation. Depending on the

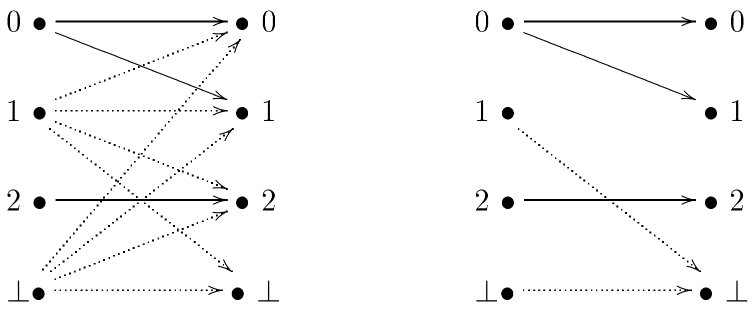


Figure 2.1: Relational Interpretations of the Operation $Op = \{(0,0),(0,1),(2,2)\}$ over the state $\{0,1,2\}$

chosen interpretation, totalisation binds all states not in the domain to all others and $\perp$, a distinguished state representing non-termination, or it binds all states not in the domain only to $\perp$.

In the contractual interpretation the domain of the operation describes the area in which the operation should be guaranteed to deliver a well-defined result as described by the relation. This area is commonly referred to as the precondition of the operation. Outside the domain, however, the operation may be applied but can return any value, including an undefined one. In the blocking interpretation operations may not be applied outside their domain. Applying the operation anyway leads to an undefined result. In this context, the precondition is often called the guard of the operation.

Consider a particular before state s. A substitution of the operation AOp by an operation COp would be unnoticed if either (1) s is in the domain of AOp, then the after state for COp should be one of the possibilities in the range of AOp. Furthermore, this means that s should also be in the domain of COp otherwise $\perp$ would be allowed by COp but not by AOp; or (2) in the contractual interpretation, if s is not in the domain of AOp, then any possible after state for COp is acceptable. This, in turn, means that s may, or may not, be in the domain of COp.

This intuition is formalised in the following way. An operation COp is an operation refinement of an operation AOp over the same state space $State$ and with the same inputs $x? : X$ and the same outputs $y! : Y$, if and only if

Applicability

$$\forall\, State;\ x? : X \bullet \operatorname{pre} AOp \vdash \operatorname{pre} COp$$

Correctness

$$\forall\, State;\ State';\ x? : X;\ y! : Y \bullet \operatorname{pre} AOp \land COp \vdash AOp$$

Note, we use the turnstile notation as it is more general than implication. We will find in subsequent chapters that consequence and implication are not always interchangeable as in standard predicate logic. The correctness rule above applies within the standard, contractual, interpretation of a Z operation. In the alternative, blocking interpretation, the correctness rule becomes

$$\forall\, State;\ State';\ x? : X;\ y! : Y \bullet COp \vdash AOp$$

There are a few special cases worth considering. First, an operation does not necessarily have to have inputs and outputs. The applicability and correctness conditions simplify accordingly. Furthermore, if the precondition of the concrete and abstract operation are the same, i.e. $\mathrm{pre}\, AOp = \mathrm{pre}\, COp$ then COp is an operation refinement of AOp if and only if

$$\forall\, State;\ State' \bullet COp \vdash AOp$$

i.e. the correctness condition was simplified using $Op = \mathrm{pre}\, Op \wedge Op$. Note, this holds in both the blocking and the contractual interpretation.

Schema conjunction is one way of obtaining operation refinements. This automatically guarantees correctness and only applicability needs to be checked. Thus, the operation $AOp \wedge X$, for operations X and AOp both over $\Delta State$, is an operation refinement of AOp if and only if

$$\forall\, State \bullet \mathrm{pre}\, AOp \vdash \mathrm{pre}(AOp \wedge X)$$

For example, in Subsection 2.4.3 we formed the schema *OkAddUser* by a conjunction of the schemas *AddUser* and *OkOp* and, indeed, we can verify that *OkAddUser* is an operation refinement of *AddUser* using the Z/EVES proof tool.

```
=> try \forall Library @ \pre AddUser \implies \pre OkAddUser;
=> prove by reduce;
```

Proving gives ...

true

In the contractual interpretation, operation refinement allows preconditions to be weakened and non-determinism to be reduced. The applicability condition requires that the concrete operation is defined everywhere the abstract operation was defined. It allows, however, that the concrete operation is defined where the abstract operation was not. The correctness condition requires the concrete operation to map into the range of the abstract operation everywhere the abstract operation is defined. It does not require, however, to cover the whole range of the abstract operation, i.e. it is not necessary for the concrete operation to be identical to the abstract operation.

For example, the operation *TotalAddUser* is an operation refinement of the operation *AddUser*. The operation *TotalAddUser* is applicable everywhere *AddUser* was defined. Additionally, it is also defined in case the user *name*? is already a member of the library.

```
=> try \pre AddUser \shows \pre TotalAddUser;
=> prove by reduce;
```

Proving gives ...

$$true$$

Furthermore, the operation *TotalAddUser* performs every task that *AddUser* does but more. We already showed that *OkAddUser* is an operation refinement of *AddUser*. Because the preconditions of *FailAddUser* and *OkAddUser* are disjoint correctness follows immediately.

```
=> try \pre AddUser \land TotalAddUser \implies AddUser;
=> prove by reduce;
```

Proving gives ...

$$true$$

Besides operation refinement (Derrick and Boiten, 2001) consider two more cases of *simple* refinements. These are concerned with establishing and imposing invariants. Since we are not using such refinements in our work we will not discuss them here.

2.5.2 Data Refinement

In data refinement we are concerned about a more concrete representation of the state. Data refinement, however, is not much considered in this work. Nevertheless, we refer to it and thus we present briefly what data refinement is about. For a thorough introduction to data refinement we recommend (Derrick and Boiten, 2001). Note, for illustrative purpose we consider here only operations with no inputs or outputs.

Simple, operation, refinement was restricted to operations over the same state. However, to move closer to an implementation the definition of the state needs to be refined too. For example, in an abstract specification we use sets frequently, however, a more concrete representation contains lists or arrays instead. Note, changing the data representation will also affect the operations over them.

The standard definition of data refinement for Z schemas whose operations are total relations is now commonly given by using simulations. A simulation is also known as a retrieve relation or abstraction relation. Basically, there are two forms of simulation, called upward and downward simulation.

$$\begin{tikzcd} AState \arrow[r, "AOp"] \arrow[d, dotted, "R"'] & AState' \arrow[d, dotted, "R'"] \\ CState \arrow[r, "COp"'] & CState' \end{tikzcd} \qquad \begin{tikzcd} AState \arrow[r, "AOp"] & AState' \\ CState \arrow[u, dotted, "R"] \arrow[r, "COp"'] & CState' \arrow[u, dotted, "R'"'] \end{tikzcd}$$

Figure 2.2: Refinement Using Downward and Upward Simulation

Figure 2.2 shows two commutative diagrams representing downward and upward simulation. The abstraction R is a relation, the arrows labelling R and R' just indicate the direction to follow around the diagram.

The first diagram describes that the application of the relation R followed by the operation COp can be matched by the operation AOp followed by a mapping R'. In the second graph the simulation is reversed, i.e. the effect of COp followed by R' can be matched by R followed by AOp. In either case, valid applications of the concrete operation can be simulated by applications of the abstract operation.

For Z schemas AOp and COp without input or output, the relation R on $AState \wedge CState$ is a downward simulation from AOp to COp if

Initialisation

$$\forall\, CState' \bullet CInit \vdash \exists\, AState' \bullet AInit \wedge R$$

Applicability

$$\forall\, AState;\ CState;\ \bullet \operatorname{pre} AOp \wedge R \vdash \operatorname{pre} COp$$

Correctness

$$\begin{array}{l} \forall\, AState;\ CState;\ CState' \bullet \\ \qquad \operatorname{pre} AOp \wedge R \wedge COp \vdash \exists\, AState' \bullet R' \wedge AOp \end{array}$$

Note, these rules assume the standard, contractual, interpretation of Z operations. In the blocking interpretation, the correctness rule becomes

$$\forall AState;\ CState;\ CState' \bullet R \land COp \vdash \exists AState' \bullet R' \land AOp$$

This formalisation of downward simulation extends the notion of operation refinement by considering initialisation and also changes in the state space. The intuition behind applicability and correctness remain the same apart from considering the change of state space, which is described by the retrieve relation R.

Downward simulation is the most common way of checking data refinement. However, it has been found that there are valid refinements that cannot be verified using downward simulation but using upward simulation. Upward as well as downward simulation are sound, i.e. if an upward or downward simulation exists between conformal operations AOp and COp then COp is a data refinement of AOp. However, upward and downward simulation are only jointly complete, i.e. refinements are possible which require both kinds of simulations for their proof.

Note, we do not consider data refinement any further in this work. However, we are interested in applying our work to data refinement in the future.

2.6 Tool Support for Z

There are a number of tools available to support the Z notation. These tools offer various degrees of assistance in type setting Z specifications and pretty printing, syntax and type checking, theorem proving and specification animation. The following list of tools is a small sample and certainly not complete. We refer to the Z home page for more details.

oz.sty is a LaTeX macro for Object-Z from the SVRC (Software Verification Research Centre) at the University of Queensland. We used this style to type-set the Z notation in this thesis.

FuZZ is a printing and type-checking system for Z specifications. Using FuZZ together with LaTeX you can input Z specifications as ASCII file, process them for laser printing, check specifications for their conformance to the Z language rules and produce a listing of schemas with components and their types. The FuZZ distribution contains a special LaTeX font of Z symbols and a library containing the standard mathematical tool-kit. FuZZ is fully compatible with the reference manual by (Spivey, 1992). Using FuZZ requires a licence.

ZTC – the Z Type Checker – can determine if there are syntactical and typing errors in Z specifications. It is intended to be compliant to (Spivey, 1992). ZTC accepts as input specifications written in LaTeX using the `oz` or `zed` packages, or its own ZSL notation which is an ASCII version of the Z syntax by the author of ZTC. It is available free of charge for educational and non-profit uses.

Formaliser is a syntax-directed Z editor and interactive type checker. It provides the facilities to interactively query attributes and to view all identifiers with their types. Formaliser is a what-you-see-is-what-you-get type of editor showing all Z symbols as they appear printed. Documents can be exported to LaTeX or its true-type Z font can be used to create MS-Word documents. Formaliser is a commercial tool, developed at Logica (UK), which runs under the Windows operating system.

ProofPower is a specification and proof tool based on an implementation of Higher Order Logic (HOL). It provides support for specification and proof in Z using a semantic embedding of Z in HOL. The distribution provides an interface of ProofPower to TeX and LaTeX, an X Windows front-end, the HOL as well as Z specification and proof development system and, finally, the DAZ tool supporting refinement from Z to the SPARK subset of Ada. ProofPower is available free for academic and personal, non-commercial use from Lemma One (`http://www.lemma-one.com/ProofPower/`).

CADiℤ is a set of integrated tools for preparing, type checking and analysing Z specifications, which is available free of charge from the University of York (UK). It gives direct support for the (ISO/IEC 13568, 2002) Standard Z notation and evolves accordingly. A Z specification is prepared using LaTeX or troff mark-up and imported into CADiℤ. The CADiℤ toolset then provides syntax, scope and type checking, type-setting and specification browsing. It allows to prove conjectures interactively. It provides different decision procedures, like model checking and resolution. Furthermore, the expansion of schemas and an elementary refinement editor are supported. CADiℤ received a BCS Award for outstanding technological achievement in the computing field.

Z/EVES supports the analysis of Z specifications by providing syntax and type checking, schema expansion, precondition calculation, domain checking and general theorem proving. It supports almost the entire Z notation and includes the mathematical toolkit as given by (Spivey, 1992). The Z/EVES theorem prover provides powerful automated reasoning as well as interactive proof development. Users with little experience in theorem proving can use the tool, too. Syntax and type checking, schema expansion and precondition calculation require little interaction.

In the current version (2.1) Z/EVES also includes a graphical user interface that allows Z specifications to be entered, edited, and analysed in their typeset form. It supports the incremental analysis of specifications and it manages the synchronisation of the analysis with modifications to the specification. Z/EVES can be obtain from ORA Canada (`http://www.ora.on.ca/z-eves/`) free of charge for educational use. It runs under the Linux, Windows and Solaris operating systems.

Z/EVES, as decribed by (Saaltink, 1997), is the tool we used to analyse the specifications given in this thesis. We used the tool to type-check all specifications as well as to calculate preconditions, to check properties and to validate refinement conditions.

Recently a new Community Z Tools Initiative (CZT) has been proposed to join the effort of developing a coherent and extensive set of Z tools and as such to support further application of Z in industry.

2.7 Formal Methods and Notations related to Z

Z has some relatives in the world of formal methods and formal notations. As such, we assume that some of the work presented in this thesis may also apply to the notations presented below. The chosen relatives are closely related to Z. The development of Z has benefited from and contributed to the development of these notations. For example, Jean-Raymond Abrial developed Z while being in Oxford together with Cliff Jones, who was largely involved in the development of the Vienna Development Method (VDM). Later, Abrial developed the B-Method, most certainly building upon his experiences gained earlier.

2.7.1 The B-Method

The B-Method has been developed by Jean-Raymond Abrial, also the originator of the Z notation, and others. The B-Method is described in *The B-Book* by (Abrial, 1996). It is a method because it is aimed at the development of program code from a specification which is given in B's own Abstract Machine Notation. The B-Method includes extensive tool support, notably the B-Toolkit by B-Core Ltd and Atelier B. The B-Method has been applied in many significant industrial projects.

The basic building block of a B specification is an abstract machine. The B-Method supports the development of large specifications from small ones by providing a number of structuring mechanisms. B and Z are both based on the same underlying logic and set theory. The B calculus, however, is based on Dijkstra's guarded command language. In B, preconditions are stated explicitly and so is non-determinism. The postcondition in B looks like an assignment in programming languages but its semantics is based on substitution on the state, like in VDM and Z. B provides also a guard construct, thus facilitating both guarded and precondition interpretation. Note, too, that the B-Method incorporates a particular notion of refinement within its language definition.

(Schneider, 2001) provides a textbook introduction to the B-Method. He covers the B approach to software development from specification through refinement, to implementation and code generation, considering verification at each step. In comparison to (Abrial, 1996), he also covers tool support, in particular the B-Toolkit.

2.7.2 The Vienna Development Method

The Vienna Development Method is a set of techniques for modelling computing systems, analysing those models and progressing to detailed design and coding. It originated at the IBM Vienna Laboratory in the mid-1970s. The notation and tools have been continuously developed since and are applied on a wide range of systems. VDM is a method because it emphasises the development of program code and provides the necessary mechanisms. (Jones, 1990) provided one of the standard references, introducing the reader to the systematic software development using VDM and (Jones and Shaw, 1990) present a collection of case studies in VDM.

VDM is based on a three-valued logic, which allows treatment of undefinedness of partial functions not explicitly cared for in Z or B. Furthermore, in VDM, preconditions and postconditions are given explicitly, which does not apply to Z. The advantage is an additional consistency check whether the real precondition of the operation corresponds to the stated one. Invariants in VDM, however, are assumed to be an implicit part of every pre- and postcondition.

B, VDM and Z were compared in the literature by (Bicarregui and Ritchie, 1995), providing a comparison of the VDM and B notations, (Hayes et al., 1993), emphasising on understanding the differences between VDM and Z, and `http://www.b-core.com/ZVdmB.html` comparing all three notations. There are also a VDM+B project at Imperial College and a Z+VDM project at SVRC aimed at combining these notations. More information on VDM, like tools, bibliography and application database can be found on its home page: `http://www.csr.ncl.ac.uk/vdm/`.

2.7.3 Object-Z

Object-Z is an extension of the formal specification language Z, retaining existing syntax and semantics, to accommodate object orientation. The main reason for this extension is to improve the clarity of large specifications through enhanced structuring. It also facilitates modular verification and refinement.

A Z specification, as presented above, defines a number of state and operation schemas. A state schema introduces the variables and defines the relationship between their values. An operation schema defines the relationship between the before and after states corresponding to one or more state schemas. One of the disadvantages of Z is that one is required to examine the signature of all operations to inferring those operation schemas that may affect a particular state schema. In large specifications this is rather impracticable.

Object-Z overcomes this problem by introducing a new class structure which encapsulates a single state schema with all the operations which may affect that state. Each class can be examined and understood in isolation. An Object-Z specification of a system comprises a number of class definitions possibly related by inheritance, a mechanism for class adaptation by modification or extension, and instantiation.

Differences of Z and Object-Z include that the scope of global type and constant definitions in Object-Z is limited to the class in which they are defined. Furthermore, an operation schema extends the notion of a schema in Z by adding to it a Δ-list. The Δ-list holds the primary variables which the operation may change when it is applied to an object of the class. All other primary variables remain unchanged. This results also in a different treatment of the precondition of operations. In Z, being outside the precondition leads to divergence, i.e. the operation can perform anything. In Object-Z, however, operations are blocked outside the precondition and thus cannot change the environment, unless they have been explicitly declared in a so called Delta-list. Note, too, that Object-Z has an operational semantics, unlike Z.

For an introduction to Object-Z the work by (Duke et al., 1994) is recommended. (Stepney et al., 1992) provide a collection of papers describing various approaches of object orientation in Z, including Object-Z. (Smith, 2000) published a reference manual in the style of (Spivey, 1992).

2.8 Summary

Z is a formal specification notation useful for describing computing systems. Z is a model-based notation. A system is modeled by representing its state, i.e. its components and constraints upon them, and operations that can change the state, thus modelling the behaviour of a system. Note, Z is not intended to specify non-functional requirements, like usability, performance, program code size and reliability. It is also not intended for the description of timed or concurrent behaviour.

In this chapter we introduced some basics of the Z specification notation. We covered the logic of Z and the underlying set theory. We went on to introduce the concept of types and their usage in Z. Furthermore, we presented the main features of Z, its schemas notation and the schema calculus, used to modify and combine schemas. Next, we gave some insight into refinement in Z, the development of a more concrete specification from an abstract one. Finally, we introduced some Z tools and other specification notations related to Z.

Details related to the Z notation including information on publications, the Z standardisation process, Z courses, tool support, and other material can be found on the Z home page: `http://www.comlab.ox.ac.uk/archive/z.html`.

Chapter 3

Inconsistency and Underdefinedness in Z

We are faced on an almost daily basis with inconsistent and incomplete knowledge. We have learnt to live with it and to manage it. This does not imply that we accept the status quo and stagnate. Both kinds of deficiencies provide a tool for development and guide research. Most importantly, however, we are able to tolerate both problems until they can be solved. Meanwhile we make use of them to derive as much possible and useful information as we can.

The Z notation is a specification language based on classical logic. Classical logic, however, is not well-designed to handle inconsistent and incomplete knowledge. Inconsistency, for example, leads to the problem of triviality, i.e. that everything can be inferred from a single inconsistency. Z specifications can also be trivialised by inconsistencies. So far, research on handling inconsistency in Z focused mainly on detecting and eradicating them.

Software development, however, requires a more lightweight approach to inconsistencies. On the one hand, they frequently appear in large projects and constant focus on detecting and eradicating inconsistencies is expensive. On the other hand, removing one inconsistency might introduce another one and thus, it is claimed, complete consistent specifications might not be reached in practice. Consequently, inconsistencies need to be managed as we do it on a regular basis too. Thus, Z needs to be extended to facilitate such inconsistency management.

In Z operations are, in general, partial relations. In the traditional interpretation, an operation applied outside its domain can result in any behaviour, thus for any component in the scope of the operation a definite value cannot be known. Alternatively, in the guarded interpretation, no change of the components occur. It has been observed that a combination of both interpretations is sometimes convenient to allow both modelling of refusals and under-specification. We propose an extension to Z to incorporate both interpretations.

3.1 Introduction

Inconsistencies are a matter of every day life. We are constantly challenged by contradicting information. Sometimes we are able to resolve the inconsistency right away; sometimes, however, we have to live with inconsistencies. In such a case we tend not to derive any useless results from it. Often it is quite the contrary and inconsistencies lead to new discoveries. This process suggests that the logic we use to reason in everyday life is able to deal with inconsistencies in a useful manner.

(Valentine, 1998), however, states:

> Consistency is essential for a Z specification to have any useful meaning.

Thus, inconsistent Z specifications are meaningless or useless. This is, however, contrary to practical situations because, as (Ghezzi and Nuseibeh, 1998) found,

> Inconsistencies are inevitable in large projects. [...] A completely consistent state may never be reached in practice

This leads to the conclusion that Z should not be used to specify large projects in practice because they would potentially be inconsistent and thus the specification is meaningless. The problem is, that the Z notation cannot deal appropriately with inconsistent situations.

This impracticality is certainly not desired by the Z community. Research on inconsistent specifications has been an issue for some time. However, common to all approaches is to prevent or eradicate inconsistencies. For example, the Z type system is well designed to prevent many inconsistencies and type checkers complement this task. Furthermore, the work by (Valentine, 1998) is aimed at providing guidelines to the development of consistent specifications.

Another research direction is to divide inconsistent specifications into viewpoints where each viewpoint should be internally consistent. We think, however, that the problem of consistency does not disappear with this approach. On the one hand, a viewpoint could include an unresolvable inconsistency and thus approaches to find and manage this inconsistency are required. One can argue that the viewpoint is further divided thus forming a hierarchy of viewpoints. However, at the end of the development process viewpoints need to be combined and thus the problem of inconsistency reappears.

3.1.1 Motivation

The aim of our work is to supplement current research on inconsistencies in Z specifications. We are interested in a mechanism that can tolerate inconsistencies but still derive useful information. Certainly, an inconsistent specification is never fully correct but sometimes it is the best we can get.

In this chapter we provide some background on the notion of inconsistency in Z specifications and the impact inconsistencies can have on the process of reasoning about Z specifications. We argue that the effect of inconsistencies in Z is not compliant with the perceived effect of inconsistencies in science or in software development practice. We illustrate with some examples what kind of reasoning we intent to facilitate. The aim of our envisioned reasoning system are more useful and reliable inferences in the presence of inconsistency. Additionally, we consider the refinement process of inconsistent operation which is currently rather arbitrary because information present in the specification are not used appropriately. Consequently, we propose to investigate the use of paraconsistent logics for Z.

Contradicting information often needs to be tolerated due to some lack of knowledge. Thus, inconsistency and underdefinedness are closely related topics. Underdefinedness occurs in Z specifications in form of partial operations. There are two opposing interpretations of applying an operation outside its domain. We introduce the two interpretations and we demonstrate that one interpretation alone is not always sufficient to model, in particular, reactive behaviour. Thus, we propose a combination of both.

3.1.2 Outline

This chapter is structured as follows. In Section 3.2 we present some sorts of inconsistencies in Z and how they can arise. Next, in Section 3.3, we discuss that inconsistencies can be a tool to guide development and we look at desired inferences despite inconsistencies in Z specification. Underdefinedness can be considered to be closely related to inconsistency. In Section 3.4 we introduce the concept of underdefinedness in Z specifications and propose a way to handle them. Finally, we provide a short summary in Section 3.5.

3.2 Inconsistency in Z Specifications

A specification is supposed to be a model of some possible system. A specification is inconsistent if it has no models. The notion of inconsistency is central to this thesis. Therefore, we discuss in this section the meaning of inconsistency in Z specifications. (Boiten et al., 1999) refer to the consistency of a single specification as unary consistency. We also consider briefly the problem between specifications, as it occurs in the area of viewpoint specifications.

3.2.1 Global Inconsistency

(Saaltink, 1997) distinguishes basically two different types of inconsistency in Z specifications, called global and local inconsistency. Global inconsistency is more serious because it makes an entire specification unsatisfiable. This occurs if some axiomatic schema, generic schema, or predicate is too strong.

Inconsistent Axiomatic Definitions

Axiomatic definitions are commonly used in Z. They provide definitions that range over the entire specification. Thus, if they are inconsistent they effect the whole specification. For example, any specification containing the axiomatic schema

$$\begin{array}{|l} n : \mathbb{Z} \\ \hline n \neq n \end{array}$$

cannot be satisfied because there is no possible value for n. Inconsistencies are not always as obvious as above. For example, there is no function f satisfying the following description:

$$\begin{array}{|l} f : \mathbb{N} \rightarrow \mathbb{N} \\ \hline \forall\, x, x' : \mathbb{N} \bullet (x < x' \Rightarrow f(x) > f(x')) \end{array}$$

Although the strong type system of Z prevents quite a few errors, it is still possible to write some kind of contradiction, like postulating that an empty set has an element

$$\begin{array}{|l} x : \varnothing[\mathbb{N}] \end{array}$$

or using the fact that a function is a set of pairs, for example

$$\begin{array}{|l} f : \mathbb{N} \rightarrow \mathbb{N} \\ \hline f = \{(1,2),(1,3)\} \end{array}$$

In all these cases, it is possible to check whether such an axiomatic definition is meaningful. As (Saaltink, 1997) shows, to check an axiomatic definition

$$\begin{array}{|l} Decl \\ \hline pred \end{array}$$

for consistency it can be preceded with the conjecture $\exists\, Decl \bullet pred$. For example, proving $\exists f : \mathbb{N} \to \mathbb{N} \bullet f = \{(1,2),(1,3)\}$ results in *false* and thus this axiomatic definition is not meaningful.

All the given examples of axiomatic definitions are inconsistent in themselves, thus it is possible to apply the aforementioned conjecture. However, it is not always as simple. It is possible to construct a number of axioms, each consistent but together they are inconsistent. (Valentine, 1998) provides the following example of two axiomatic definitions and an enumerated type.

$$\begin{array}{|l} x : \mathbb{N} \\ \hline x = 2 + 2 \end{array} \qquad \begin{array}{|l} y : \mathbb{N} \\ \hline y = x \\ y = 5 \end{array}$$

$$Person ::= SamValentine \mid thePope$$

Then it is possible to show, using classical logic, that $\vdash SamValentine = thePope$ holds because of the inconsistency between the two axiomatic definitions. Basically, the proof proceeds over the cardinality of the set $\{SamValentine, thePope\}$, which is 2. However, due to the inconsistency it is possible to show that $2 = 1$, thus the cardinality of the set is one, which means the elements must be the same.

Inconsistent Free Types

(Spivey, 1992, p. 84) points out that free types can be inconsistent, too, because of cardinality problems. For example, the data type definition

$$T ::= atom\langle\!\langle \mathbb{N} \rangle\!\rangle \mid fun\langle\!\langle T \to T \rangle\!\rangle$$

is inconsistent. Basically, no such set T can exist because there are many more functions from T to T than there are members of T. An even simpler example is given by the definition

$$BigSet ::= makeSet\langle\!\langle \mathbb{P}\, BigSet \rangle\!\rangle$$

which has no model because it specifies that *BigSet* is isomorphic to its power set. This is impossible, as the power set of a set always has more elements than the set itself. Although we introduced the problem of inconsistencies through free types in Z, we will not consider it any further. (Arthan, 1992), (Smith, 1992), and (Spivey, 1992, p. 84) describe restrictions on free type definitions that guarantee consistency.

In the standard theory of Z, no theorem that has been proved in a globally inconsistent specification can be trusted because its proof is potentially based on impossible assumptions. Our general aim, however, is to investigate possibilities to reduce the impact of inconsistencies such that there will be proofs of theorems that can be trusted.

3.2.2 Local Inconsistency

Set declarations, abbreviations and schema definitions do not introduce global inconsistency. However, schema definitions can be locally inconsistent, i.e. they contain an unsatisfiable predicate. This kind of error is local in the sense that the specification of other components of the system may still be meaningful.

Inconsistent Operation Schema

A schema can have an inconsistent, i.e. unsatisfiable, predicate. If such a schema is an operation schema, then the operation may not guarantee any outcome or only parts of the operation can be determined. For example, consider the following inconsistent operation

$$
\begin{array}{|l}
\multicolumn{1}{l}{Op_{ic}} \\
\hline
x?, y! : \mathbb{N} \\
\hline
x? = 1 \Rightarrow y! = 2 \\
x? = 1 \Rightarrow y! = 3 \\
\hline
\end{array}
$$

The above schema includes the contradiction that $y!$ cannot be 2 and 3 at the same time. The precondition for this operation is $[x? \in \mathbb{N} \mid x? \neq 1]$, i.e. it should not be applied when $x? = 1$. Thus, the operation is not "completely" inconsistent.

Inconsistent State Schema

If a schema describing the state of a system is inconsistent then it is impossible to build that particular system. For example, in the state schema

$$
\begin{array}{|l}
\multicolumn{1}{l}{S1_{ic}} \\
\hline
x : \mathbb{N} \\
\hline
3 \leq x \leq 2 \\
\hline
\end{array}
$$

the state constraints cannot be satisfied. This error can be shown easily because $\exists\, x : \mathbb{N} \bullet S1_{ic}$ fails as there is no x that can satisfy the state schema. However, state inconsistencies are not always as simple. For example, the state schema

$$
\begin{array}{|l}
\multicolumn{1}{l}{S2_{ic}} \\
\hline
x, y : \mathbb{Z} \\
\hline
x \bmod 2 = 0 \Rightarrow y < x \\
x \bmod 2 \neq 0 \Rightarrow y = x + 1 \land y \bmod 2 \neq 0 \\
\hline
\end{array}
$$

is meant to ensure that two numbers are always in a particular relation to each other. However, $S2_{ic}$ is partially over-constrained. It is possible to find even numbers x such that $S2_{ic}$ is satisfied but no odd numbers. Thus, it is possible to build a system based on $S2_{ic}$ but, possibly, not the intended one.

The Initialisation Theorem

The initialisation theorem plays an important role in checking specifications for consistency. (Saaltink, 1997), for example, states: "many specifications give an initialization schema of the form $Init\_S \mathrel{\widehat{=}} [S \mid P]$, where the predicate P further constrains the state. In such a case, showing $\exists\, S' \bullet Init\_S$ not only shows that S is satisfiable, it also shows that initial states are possible."

Unfortunately, the initialisation theorem does not prevent specification of partially inconsistent state descriptions, like in $S2_{ic}$. For example,

$$Init\_S2_{ic} \mathrel{\widehat{=}} [S2_{ic}' \mid x' = 2 \land y' = 1]$$

is a valid initialisation which can be proved using the above conjecture.

3.2.3 Inconsistency between Viewpoint Specifications

It is generally agreed that a system of realistic size cannot be modelled in a single specification. It rather has to be decomposed into several specifications of reasonable size where each such specification will have to be developed separately. (Jackson and Jackson, 1996) argue that unlike in programming, where hierarchical or functional decomposition is often used, systems should be decomposed into different aspects, called viewpoints. Each viewpoint forms a partial descriptions of the system, the combination of all viewpoints form the model of the whole system. The viewpoints can, however, overlap and thus consistency between viewpoints becomes an issue.

Unification is a method to combine viewpoint specifications in Z proposed by (Derrick et al., 1995). It has been subsequently developed by (Boiten et al., 1995), (Bowman et al., 1996) and (Boiten et al., 1999). Two specifications are said to be consistent if it is possible for at least one implementation to exist that conforms to both specifications. Refinement is used to check whether an implementation meets the requirements of a specification. The least common refinement of two specifications is their unification. Thus, two specifications are consistent if their unification exists. If they are inconsistent then it is not possible to construct the unification and, therefore, their implementation.

A Digital Clock Example

We give a small, simplified example of an engineering task. Given is a timer device, i.e. a clock. Two engineers are each asked to give a model of a device that can initiate events within intervals of maximal 12 hours.

State. Both engineers rely on the same given clock, named *Digi*12 with fields for minutes and hours, denoted m and h respectively. We model both as restricted integers. Thus, the state schema is already normalized.

$$
\begin{array}{|l}
\hline
\textit{Digi}12 \\
m, h : \mathbb{Z} \\
\hline
0 \leq m \leq 59 \\
0 \leq h \leq 23 \\
\hline
\end{array}
$$

Initialisation. Initially, the clock starts at noon, thus

$$\textit{InitDigi}12 \mathrel{\widehat{=}} [\textit{Digi}12' \mid m' = 0 \land h' = 12]$$

The initialisation condition holds for the given clock, i.e. the initial state exists, which can easily be verified.

Operations. The two engineers, however, decide to model the *Tick* operation differently. The operation specifies the state change of the given clock and thus it is concerned with the behaviour of the clock when one minute has passed. This includes to update the values of the minutes m and hours h accordingly.

$$
\begin{array}{|l}
\hline
\textit{Tick}1 \\
\Delta \textit{Digi}12 \\
\hline
m < 59 \Rightarrow \\
\quad m' = m + 1 \land h' = h \\
m = 59 \Rightarrow \\
\quad m' = 0 \land \\
\quad (h < 23 \Rightarrow h' = h + 1) \land \\
\quad (h = 23 \Rightarrow h' = 0) \\
\hline
\end{array}
\qquad
\begin{array}{|l}
\hline
\textit{Tick}2 \\
\Delta \textit{Digi}12 \\
\hline
m < 59 \Rightarrow \\
\quad m' = m + 1 \land h' = h \\
m = 59 \Rightarrow \\
\quad m' = 0 \land \\
\quad (h < 12 \Rightarrow h' = h + 1) \land \\
\quad (h = 12 \Rightarrow h' = 1) \\
\hline
\end{array}
$$

Minutes range from 0 to 59 and are incremented with each *Tick*. Once 59 is reached they go back to 0 and the hour is incremented, too. In viewpoint one, the clock counts the hours from 0 to 23. When it has reached 23:59, another *Tick* sets it to 0:00. In viewpoint two, hours range from 1 to 12. At 12:59 a *Tick* sets it to 1:00.

We developed two different viewpoints of a particular problem. Consider that these viewpoints describe only one part of a larger system in which they need to be integrated. Thus, we are required to check whether both viewpoints can be satisfied. Unification is the method to apply.

The unification of both viewpoints, however, fails. To hold, state consistency, initialisation consistency and operation consistency for both viewpoints must be satisfied. We omit the state and initialisation conditions because they are trivially satisfied for this example. However, operation consistency fails.

Two operations Op_1 and Op_2 both operating over the same state S with input $x? : X$ and output $y! : Y$ are *operation consistent* if and only if the following holds

$$\forall S;\ x? : X \bullet \text{pre}\ Op_1 \land \text{pre}\ Op_2 \Rightarrow \exists S';\ y! : Y \bullet Op_1 \land Op_2$$

Applying this to both operations $Tick1$ and $Tick2$ it is easy to see that they are inconsistent in the case of $m = 59$ and $h = 12$ and another *Tick*. Thus unification fails for these two viewpoints.

3.3 Inconsistency and Information

(Valentine, 1998) states the common assumption that "Consistency is essential for a Z specification to have any useful meaning." In this section we challenge this commonly accepted view. We start by providing some analogy to other sciences dealing with complex descriptions. Then, we present some inconsistent specifications in Z which, as we argue, do have a meaning.

3.3.1 Inconsistencies in Science

A Z specification is a formal description of a possibly complex system. In practice, large specifications are likely to contain inconsistencies. This problem is not limited specifically to formal specification. There are other areas dealing with describing complex phenomena formally. For example, the natural sciences are mostly concerned with describing, i.e. specifying, phenomena occurring in the real world. They, too, have to face inconsistencies on a regular basis. These sciences, however, have somehow learnt to live with inconsistencies, to manage and to utilise them.

Bohr's Theory of the Atom

The sciences of Physics and Chemistry are concerned with the formal description of mostly complex systems. It is here, in the history of science, that we find many inconsistent but non-trivial theories. (Priest and Tanaka, 1996) present as one example the well-known theory of the atom by Niels Bohr. According to this theory, an electron orbits the nucleus of the atom without radiating energy. However, according to Maxwell's equations, which were an integral part of Bohr's theory, an electron which is accelerating in orbit must radiate energy. Hence, Bohr's description of the behaviour of the atom was inconsistent. However, it was still possible to infer useful results from this theory, while other non-useful conclusions were rejected. In science, inconsistencies are often accepted to simplify a model as long as these inconsistencies do not lead to wrong conclusion.

Clausius's Proof of Carnot's Theorem

(Meheus, 2002) presents an example of reasoning in the presence of inconsistency. The problem considered is Clausius's proof of Carnot's theorem: "no engine is more efficient than a reversible engine." At the time, two incompatible approaches to thermodynamic phenomena existed. On the one hand, the theory by Carnot stated that the production of work in a heat engine results from the mere transfer of heat from a hot to a cold reservoir. On the other hand, Joule advocated that the production of work in a heat engine results from the conversion of heat into work. Both approaches combined lead to several contradictions, e.g. the production of work results from the mere transfer of heat and from the conversion of heat.

Carnot's proof of his theorem is based on Reductio ad Absurdum, i.e. he supposed that the negation of his theorem holds and shows that this would lead to a contradiction. Thus, the hypothesis must be rejected on the basis of this contradiction and the opposite must hold. This pattern of proof is well accepted and often applied in mathematical reasoning. Clausius developed two proofs of Carnot's theorem both based on this concept and both are very similar. However, he rejected the first of his proofs. Both proofs are based on Carnot's and Joule's premises, however, the first proof does need the hypothesis to derive the contradiction, while his second proof does. Thus, he found a useful and valid way of reasoning in the presence of inconsistency.

A Little Experiment

The following is a little experiment to demonstrate how easily inconsistencies can appear in life. Consider three water tanks, filled with hot, medium and cold water respectively. Put one of your hands in the hot water tank, the other in the cold one. Leave your hands in there for a while, until you do not feel any difference in temperature anymore. Now, put both hands at the same time in the third water tank with the water of medium temperature. You will perceive on the one hand that the water is hot and on the other hand that the water is cold. This is certainly inconsistent with your knowledge of the water being of the same temperature.

Psychology, in particular, uses such phenomena regularly to investigate the mind. Often inconsistent phenomena are presented to a person and it is investigated how humans solve these problems. The example above is one such phenomena, Escher pictures are another. It has, however, not been reported that the subjects derived unrelated or useless information despite the inconsistencies.

Inconsistency implies Action

We presented some examples of inconsistent but useful theories as well as the human ability to derive useful conclusions from inconsistent premises. We do not claim that inconsistencies are desirable but they are not as useless as often thought. Inconsistencies are an important tool in science. They guide researchers to develop better theories and they instigate the natural process of learning. Inconsistencies cannot always be resolved, however, they can be managed. This is, what (Gabbay and Hunter, 1991) mean when they state:

> Inconsistency implies Action

3.3.2 Inconsistencies in Software Development

Inconsistencies are a fact of life. They occur frequently in the software development process. The need for managing inconsistency in software development has been acknowledge by many researchers. (Ghezzi and Nuseibeh, 1998) and (Ghezzi and Nuseibeh, 1999), for example, present two special issues in *IEEE Transactions on Software Engineering* covering this topic and there have been two international workshops on "Living with Inconsistency" as presented in (IWLWI, 1997) and (Easterbrook and Chechik, 2001a).

Making Inconsistency Respectable

(Nuseibeh et al., 2001) argue that maintaining consistency at all times is counterproductive. It is usually computationally expensive, descriptions evolve and thus inconsistencies re-appear, individual descriptions can be ill-formed and various degrees of formality make inconsistency checking difficult. "In many cases, it may be desirable to tolerate or even encourage inconsistency to facilitate distributed teamwork and to prevent premature commitment to design decisions, and to ensure all stakeholder views are taken into account.” Inconsistencies can also be used as a tool for learning and guiding the development process.

(Nuseibeh et al., 1994) consider inconsistency as any situation in which two descriptions do not obey some relationship that is prescribed to hold between them. (van Lamsweerde et al., 1998), for example, consider divergent goals in requirement engineering. Note, this notion of inconsistency embraces the logical definition of inconsistency. The relation that should hold is the impossibility to derive a contradiction from a set of formulae.

The proposed framework for inconsistency management consists of consistency checking, monitoring and diagnosing inconsistency, handling inconsistency, and measuring inconsistency. Consistency checking is based on a set of consistency rules which need to be obeyed. Monitoring is the process of detecting the violation of the consistency rules. Once an inconsistency is discovered, it is diagnosed. This includes to localise the inconsistency, to identify the cause for it and its classification. The choice of handling strategies includes to ignore, to defer, to circumvent or to ameliorate an inconsistency. The latter means that it may be more cost-effective to improve an inconsistent description without actually resolving all of the inconsistencies. Finally, measuring inconsistency is important to determine the impact of an inconsistency.

In a number of case studies they found that some inconsistencies never get fixed. However, “the decision to repair an inconsistency is risk-based. If the cost of fixing it outweighs the risk of ignoring it, then it makes no sense to fix it.” Tolerating inconsistencies in such circumstances means to re-evaluate the risk continuously. They found too that some inconsistencies are deniable. For example, in their experience developers often debated whether a reported inconsistency really was an issue or that it was already fixed.

Viewpoints for Managing Inconsistencies

Some researchers decided to split contradicting information into viewpoints to manage the inconsistency. For example, (Easterbrook, 1993) suggests to use hierarchies of viewpoints to represent alternative, conflicting views of information. A viewpoint is a self-contained consistent description of an area of knowledge with an identifiable originator. Viewpoints do not correspond to people but to a description of the world from a particular angle. Viewpoints in this case are merely seen as an organisational tool.

Later, (Easterbrook and Nuseibeh, 1996) are more concerned with inconsistency management using viewpoints. The paper demonstrates how inconsistency management is used as a tool for requirements elicitation and how viewpoints provide help. First, there is no requirement for changes to one viewpoint to be consistent with other viewpoints. Therefore inconsistency can be tolerated throughout the development process.

However, consistency checking and resolution is still required but consistency checking can be separated from resolution. To manage inconsistency, relationships between viewpoints have to be defined. Basically, rules are used to define partial consistency relationships between the different representations and consistency checking is performed by applying these rules. This allows consistency to be checked incrementally between viewpoints at particular stages of development.

(Easterbrook and Chechik, 2001b) extend their research to multi-valued reasoning over inconsistent viewpoints. Each viewpoint is described using an underlying multi-valued logic. Many-valued logics use additional truth values to represent intermediate values between true and false. These different logical values can then be used to represent different levels of agreement. Their framework is intended as a means of exploring inconsistencies. The analyst is not restricted in any way when concerned with the problem of merging information from different viewpoints.

Analysing Inconsistent Specifications

(Hunter and Nuseibeh, 1997) and (Hunter and Nuseibeh, 1998) present another logic-based approach to managing inconsistent specifications. Classical logic is commonly used to construct formal specifications. Classical logic, however, is trivialised in the presence of inconsistency, i.e. any inference follows from an inconsistent information. Therefore, the authors propose to use quasi-classical logic, developed by (Besnard and Hunter, 1995) to avoid such trivialisation.

The aim of their work is to demonstrate the usefulness of using alternative logical approaches to the problem of reasoning in the presence of inconsistency in the software development process. It provides a formal foundation for supporting a software specification process in which inconsistencies are analysed to determine appropriate actions for further development. Such actions also include the possibility of tolerating inconsistencies.

3.3.3 The Meaning of Inconsistent Z Specifications

We claim that inconsistent specifications do have an intended meaning. Otherwise it is rather pointless to make the effort of writing an inconsistent specification. Classical predicate logic, on which Z is based on, is unfortunately not very suitable to investigate the meaning of inconsistent specifications.

Classical predicate logic, for example, does not distinguish between falsehood and inconsistency. This problem is also carried over to the Z notation. An inconsistent operation, for example, behaves like an operation which has not been specified, i.e. it is set to false. This in turn makes it much harder to analyse the source of failure of an operation. Furthermore, refinements of inconsistent operations can be rather arbitrary.

Operation schemas, the standard precondition interpretation and inconsistency form an interesting combination in Z. An operation applied outside its precondition can result in any behaviour. This is, however, triviality and thus results in the same behaviour as applying an operation in the inconsistent situation. For example, the precondition of the operation Op_{ic} is $[x? \in \mathbb{N} \mid x? \neq 1]$. Thus, applying this operation outside its precondition means to apply it when $x? < 0 \vee x? = 1$.

Note, the way the precondition computation in Z works seems to indicate an ordering of belief, assuming, for example, state schemas to be correct while an operation can be faulty. This leads to operations not being permitted if they are violating the state condition. However, this is not necessarily correct. It could be that the operation is correctly specified but the state specification is flawed. Such a case is, for example, presented in the next subsection.

3.3.4 Examples

Next, we present some examples of inconsistent specifications. As we claimed, we do not think that they are meaningless. Thus, we provide some indication of the kind of inferences we are interested in. Essentially, we want to infer less but more useful information in the presence of inconsistency. Thus, we tend to show what we do not want to infer in comparison with classical logic, rather than what should be inferred.

Tweety the Penguin

The following example appears frequently in the literature on paraconsistent and non-monotonic reasoning. It is about Tweety, the bird who is a penguin that can but cannot fly. We decided not to provide a Z encoding of the problem because this would add some syntactical overhead not necessary for our illustration. Thus consider this example as an introduction to the topic of reasoning about inconsistent specifications.

Classically, the Tweety example is given as a universal theory in first-order predicate logic by the first four rules:

(1) $\mathit{bird}(X) \rightarrow \mathit{flies}(X)$

(2) $\mathit{penguin}(X) \rightarrow \neg\mathit{flies}(X)$

(3) $penguin(X) \rightarrow bird(X)$

(4) $penguin(Tweety)$

(5) $hungry(Tweety)$

Clause (1) states that all birds can fly. Penguins, however, according to Clause(2) cannot fly although, as Clause (3) states, they are birds. These three clauses are not inconsistent, as long as no penguins would exist. Therefore, in Clause (4) we give a particular penguin, named Tweety. These four clauses together cause an inconsistency to arise. Tweety is a penguin and therefore cannot fly but because Tweety is a penguin he is also a bird and therefore can fly. This results in the contradiction, that Tweety can and cannot fly. However, we think this contradiction should not influence any knowledge about Tweety being hungry, as stated in Clause (5).

We denote the above set of rules, i.e. the theory about Tweety, with T. In classical logic it would be possible to show

$$T \vdash \neg\, hungry(Tweety)$$

or even

$$T \vdash \neg\, penguin(Tweety)$$

This seems, however, rather counter-intuitive. On the one hand whether Tweety is hungry is actually not dependent on the issue whether he can fly or not. On the other hand, rejecting that Tweety is a penguin would not lead to the problem of inconsistency. This little specification provides some useful information, namely Tweety is hungry and he is a penguin. However the inconsistency is resolved it should respect this information.

A Flat Tyre

In (Miarka et al., 2002), we present a simplified example from the life of a motorist. The motorist is the owner of a car. To be allowed to drive the car on public roads, the car needs to pass a safety test, part of which is a tyre inspection. The law (in Germany) says that the car must have the same kind of tyre fitted to both the front and rear wheels. We use the set

$$[CAR]$$

as our basic type. The Boolean type is not part of standard Z, hence we define the enumerated type

$$\mathbb{B} ::= True \mid False$$

In the state schema, *Car*, the Boolean *flat* denotes whether any of the tyres are flat. If not the motorist is permitted to *drive* the car. The *Law* states that the *same* tyres should be used on front and back. A single operation is specified, that of changing a tyre. Unfortunately, the spare tyre is of a different type, thus we will break the law as a result of a *Change*.

$$\begin{array}{|l}
\multicolumn{1}{l}{Car} \\
\hline
flat : \mathbb{B} \\
drive : \mathbb{B} \\
wheels : \mathbb{N} \\
\hline
flat = False \Rightarrow drive = True \\
wheels = 4 \\
\hline
\end{array}
\qquad
\begin{array}{|l}
\multicolumn{1}{l}{Law} \\
\hline
same : \mathbb{B} \\
\hline
same = True \\
\hline
\end{array}$$

$$\begin{array}{|l}
\multicolumn{1}{l}{Change} \\
\hline
\Delta Car \\
\Xi Law \\
x! : \mathbb{N} \\
\hline
flat = True \land flat' = False \\
same' = False \\
x! = wheels \\
\hline
\end{array}$$

The *Change* operation is clearly inconsistent in an intuitive sense. Once the tyre has been changed, the car is not allowed on the road by the law because the type of tyre on at least one wheel is now different. We might, however, wish to reason about aspects of this specification, for example, that the car is still driveable, since this only depends on the fact that no tyre is flat. Also, the number of tyres on the car, as reported by $x!$ should be exactly four.

Although this example is small and rather artificial, it illustrates the type of reasoning one might wish to perform. It provides some evidence that reasoning in the presence of inconsistency could be useful. Note, practically the inconsistency is not resolved by dropping the law but by providing a range of exceptions to the law. Nevertheless, any development of the above specification should take into account those aspects that are not directly related to the inconsistency.

Refuel A Car

Another operation often performed by a motorist is to refuel their car. We distinguish three kinds of cars: electric cars, cars with diesel engines and cars running on petrol. The electric car needs a power supply to re-charge, whereas the other cars need fuel which can be divided into unleaded, four star and diesel. Thus we give the following two type definitions.

$$
\begin{array}{l}
CAR\_TYPE ::= electric \mid diesel \mid petrol \\
FUEL\_TYPE ::= unleaded \mid four\_star \mid diesel\_type
\end{array}
$$

We are interested in the state of a car. It can be charged, or it needs a particular amount of some sort of fuel. Given a petrol car we assume by default that unleaded petrol is to be used. This is compliant with current environmental issues.

State

$$
\begin{array}{|l}
charged : \mathbb{B} \\
fuel : FUEL\_TYPE \\
amount : FUEL\_TYPE \rightarrow \mathbb{N}
\end{array}
$$

Choose

$$
\begin{array}{|l}
\Delta State \\
car? : CAR\_TYPE \\
\hline
car? = petrol \Rightarrow \\
\qquad fuel' = unleaded
\end{array}
$$

Refueling a car results in a full energy status. This means, an electric car is to be re-charged and a petrol car has sixty liters of fuel in the tank.

Refuel

$$
\begin{array}{|l}
Choose \\
\hline
(car? = electric \wedge charged' = True) \vee \\
(car? = petrol \wedge amount'(fuel') = 60 \wedge fuel' = four\_star)
\end{array}
$$

This refuel operation is partly inconsistent because we assign two different types of fuel to be taken when the car requires petrol. It is consistent when applied to electric cars; no refuel operation has been specified for diesel cars. Clearly, this looks like a simple specification error, but in a large specification such errors can be hidden.

Despite the inconsistency we are interested in useful inferences. Such inferences include that the amount of fuel should be exactly sixty liters, no matter what fuel type was used. We also need to show that diesel is not an option to be taken as fuel for petrol cars.

3.3.5 Unification of Viewpoint Specifications

Consider the small clock example from before. We noted that the unification of the two viewpoints failed because both engineers could not agree what to do next when the clock reached $12:59$. We are, however, interested in the information this specification provides. For example, we find that no matter which viewpoint we consider the minutes m will be set to zero and nothing else. Thus, reasoning from this inconsistent set of viewpoints should validate this information.

In general, reasoning about viewpoints should facilitate the discovery of the commonalities between the specifications even in the presence of inconsistency. It should provide a mechanism to improve the system. We think, it would even be advantageous to first combine the inconsistent viewpoints and then to develop the resulting specification. Otherwise, separate developments might lead to the introduction of new problems while trying to resolve the old ones.

The unification of viewpoints is supposed to be their common refinement. Thus, investigating unification in the presence of inconsistency leads to the problem of refinement of inconsistent operations. However, this problem can of course be considered independently from unification.

3.3.6 Refinement of Inconsistent Specifications

According to (Woodcock and Davies, 1996), refinement is all about improving specifications. However, we indicated that refinements of inconsistent specifications and in particular of inconsistent operations can be rather arbitrary. Thus, we claim, not all refinements from inconsistent operations actually do improve the specification. This is mainly due to the lack of formal support to consider the information given in an inconsistent operation.

Consider the following two operation schemas

$$
\begin{array}{|l}
\multicolumn{1}{l}{Op2_{ic}} \\
x?, y? : \mathbb{Z} \\
X, X' : \mathbb{Z} \leftrightarrow \mathbb{Z} \\
\hline
X' = X \oplus \{x? \mapsto y?\} \\
\#X' = \#X \\
x? \notin \operatorname{dom} X \\
\hline
\end{array}
\qquad
\begin{array}{|l}
\multicolumn{1}{l}{ROp2_{ic}} \\
x?, y? : \mathbb{Z} \\
X, X' : \mathbb{Z} \leftrightarrow \mathbb{Z} \\
\hline
x? \in \operatorname{dom} X \\
X' = \{x?\} \lhd X \\
\hline
\end{array}
$$

$Op2_{ic}$ is meant to replace a new pair of numbers $(x?, y?)$ within a set of pairs X resulting in the new set X'. Unfortunately, in this large operation an inconsistency occurred. On the one hand, it is desired that the first component $x?$ of the new pair is not in the set X already which leads to the actual addition of one extra pair to X. One the other hand, it is required that the number of elements in the set remain constant. Both requirements, however, cannot be supported at the same time.

The problem we find is, that this operation can be refined by one which attempts the complete opposite effect. $ROp2_{ic}$ removes those pairs from X where $x?$ is the first component. Even in the presence of inconsistency there should be a way to prevent such unreasonable refinements and thus to support an improvement of the specification that is in line with the intended meaning.

In general, resolving inconsistencies can be an expensive and sometimes impossible task. Many participants can be involved each having a different view on the problem. Therefore, it might be difficult to reach an agreement on how to resolve the inconsistency. For the specifier it might thus be helpful to continue analysis and development of the specification despite the presence of inconsistency. An approach to living with inconsistency is required.

3.3.7 Proposal

(Valentine, 1998) states: "Consistency is essential for a Z specification to have any meaning." However, we believe this claim is too strong and undesirable. Even if a Z specification is inconsistent, it still has an intended meaning. The problem we need to solve is to discover the meaning and to make it explicit.

Note, our work is not related to that by (Henson, 1998) where he shows that the standard logic of Z is inconsistent. However, his work supports our claim that inconsistencies do not necessarily lead to trivial results in practice. The standard logic of Z, although inconsistent, has been used successfully to analyse many specifications.

We propose to investigate what formal support can be given to the process of analysing inconsistent specifications written in the Z notation. Such work forms a part in the wider area of research on managing inconsistencies without necessarily eradicating them. Formal support is based on logical reasoning. Thus, we are interested in logics that support reasoning in the presence of inconsistency.

Logicians have developed a range of logics to continue to reason in the presence of inconsistencies. These so called paraconsistent logics allow us to derive less but more useful information despite inconsistencies. It is our intention to investigate the consequences of using a paraconsistent logic to analyse Z specifications. We envision that inconsistent Z specifications can be analysed in more depth than at present and that refinement of inconsistent specifications can be more controlled.

Some of the more interesting candidates of paraconsistent logics have four truth values. The logical truth values represent the four epistemological situations: 'told True', 'told False', 'told True and False', and 'told neither True nor False'. Thus, four-valued logics not only capture the notion of inconsistency but also some form of underdefinedness. It is also our aim to make use of this extra truth-value as discussed below.

3.4 Underdefinedness in Z Specifications

In the common Z specification style operations are, in general, partial relations. The domains of these partial operations are traditionally called preconditions. There are, however, two different interpretations of the precondition according to the behaviour of the operation if applied outside its domain.

The "design by contract" meaning is the *standard* interpretation of a precondition of an operation in Z. This asserts that if the precondition holds and an attempt is made to execute the operation, then the execution will be accepted and it will terminate in a state as specified by the postcondition. If the precondition does not hold, however, and the operation is attempted to be executed then it will be executed but it may not terminate or it can terminate in an arbitrary state. This behaviour is also called "divergence". We usually refer to this standard interpretation by the term *precondition*.

The alternative meaning is the so called *guarded* or firing condition interpretation. If the operation is executed within its precondition it will terminate in a state according to the postcondition. However, if it is called outside the given precondition, then the operation will not be executed at all, i.e. it is blocked, and no state change occurs. This is the standard interpretation in Object-Z.

It has been observed that it is convenient to use a combination of both the guarded and precondition interpretation to allow both modelling of refusals and under-specification. (Josephs, 1991), for example, reports on specifying reactive systems in Z and (Lano et al., 1997) consider non-determinism different from under-specification.

3.4.1 Underdefinedness

Formal specifications are abstract descriptions of the behaviour of a system. They are supposed to leave as much implementation freedom as possible. Non-determinism is a particular tool to achieve this objective. During the refinement process of a specification, however, non-determinism is usually eliminated. Thus, non-determinism relates to the view of under-specification or, as we call it, underdefinedness.

Undefinedness versus Underdefinedness

There might occur a little confusion between the terms undefined and underdefined. Thus we provide some clarification of what undefined stands for. Undefinedness as, for example, considered by (Valentine, 1998) is related to the application of partial functions outside their domain. Valentine presents the following example of an axiomatic definition

$$\begin{array}{|l} total, count, average : \mathbb{N} \\ \hline \neg\ count = 0 \Rightarrow average = total \text{ div } count \end{array}$$

which looks rather reasonable. The problem of division by zero seems to be covered due to the condition. Unfortunately, this is not the case because Z is not operational. The above axiomatic schema is equivalent to the following

$$\begin{array}{|l} total, count, average : \mathbb{N} \\ \hline average = total \text{ div } count \vee count = 0 \end{array}$$

Thus, the problem with dividing by zero can still occur. There are many more examples of undefined expressions in (Valentine, 1998), as well as (Stoddart et al., 1999). In the wider scope undefinedness and underdefinedness are related because underdefinedness is concerned with the problem of applying an operation outside its domain, which is rather similar to the issue of undefinedness. However, undefinedness is not the problem we are interested in here.

3.4.2 Normalisation and Underdefinedness

We introduced normalisation as the process of rewriting a schema such that all the constraint information appear in the predicate part. We presented the following two schema $S1$ and $S2$, where $S2$ is the normalisation schema of $S1$.

$$\begin{array}{|l} \multicolumn{1}{l}{S1} \\ \hline a, a' : \mathbb{N} \\ \hline (a')^2 \leq a < (a'+1)^2 \\ \hline \end{array} \qquad \begin{array}{|l} \multicolumn{1}{l}{S2} \\ \hline a, a' : \mathbb{Z} \\ \hline a \in \mathbb{N} \wedge a' \in \mathbb{N} \\ (a')^2 \leq a < (a'+1)^2 \\ \hline \end{array}$$

Natural numbers are not a basic type of Z but constrained integers. Therefore, a schema declaration referring to naturals can be normalised to use integers and a constraint on the predicate.

However, somehow the interpretation of the schemas may change through that process. As the operation $S1$ is defined on natural numbers, it appears unreasonable to even consider applying it on negative integers, so the blocking interpretation appears quite sensible for this area. However, the normalised schema is formally equivalent to $S1$ but is interpreted in the precondition approach as being fully undefined on negative integers. This means, that the specifier needs to know about normalisation, i.e. which sets are proper types and which are proper subsets of a type, which might not always be the case and somehow should not be necessary in the first place. This example shows that normalisation is more guard, rather than precondition, related and that we might want to deal with it accordingly.

3.4.3 Guards and Preconditions in a Buffer Example

The following example is designed to demonstrate the different meanings of a precondition. We model a little buffer of messages. We use a new type MSG to represent a message because we are not interested in their particular form.

$$[MSG]$$

The state schema $Buffer$ holds the type definitions for the $buffer$ which we model as a sequence of messages. Furthermore, we use a flag r to indicate whether the buffer has been read. The buffer is initially empty and the flag r is set to $True$ to enable the $Write$ operation.

$$\begin{array}{|l}\hline \textit{Buffer} \\ \textit{buffer} : \mathrm{seq}\, MSG \\ r : \mathbb{B} \\ \hline\end{array}$$

$$\begin{array}{|l}\hline \textit{InitBuffer} \\ \textit{Buffer}' \\ \hline \textit{buffer}' = \langle\rangle \\ r' = \textit{True} \\ \hline\end{array}$$

There are two operations possible. On the one hand, messages can be stored in the buffer. This is, however, restricted to the fact that a previous message has been read before. On the other hand, messages can be read. The result of the $Read$ operation is a change in the flag. The content of the buffer after the operation is not relevant. The $Read$ operation can only be invoked on a non-empty buffer and if there is a new message waiting.

$$\begin{array}{|l}\hline \textit{Write} \\ \Delta \textit{Buffer} \\ x? : MSG \\ \hline r = \textit{True} \\ \textit{buffer}' = \textit{buffer} \oplus \{1 \mapsto x?\} \\ r' = \textit{False} \\ \hline\end{array}$$

$$\begin{array}{|l}\hline \textit{Read} \\ \Delta \textit{Buffer} \\ x! : MSG \\ \hline \textit{buffer} \neq \langle\rangle \wedge r = \textit{False} \\ x! = \textit{head buffer} \\ r' = \textit{True} \\ \hline\end{array}$$

In particular in the $Read$ operation the two preconditions have different meanings. The condition $buffer \neq \langle\rangle$ is like a guard. No state change is permitted if the buffer is empty. The condition $r = False$, however, is not as strict. If the operation is applied outside this condition but within the guard then it could be possible to read the content of the buffer again. No harm would occur. Note, the condition $r = True$ in the $Write$ operation determines a synchronous behaviour of the buffer because a message is not overridden before the old one was read. Again, whether this is a guard or a precondition is important for the behaviour outside the condition as well as for future refinements.

3.4.4 Refinement of Underdefined Specifications

The two interpretations of the precondition of an operation lead to two different notions of refinement, too. In the standard interpretation, the precondition can be weakened and thus the domain of the operation can be enlarged. In the guarded interpretation, however, the precondition cannot be weakened but possibly be strengthened. Thus, the domain of the operation is reduced.

Both instantiation of the applicability rule of refinement have, however, one intention, namely to reduce non-determinism. Obviously, both interpretations cannot be used at the same time for one operation schema. (Strulo, 1995), for example, suggests to label the operation schema according to the precondition interpretation that should be used with them.

Example cont.

The precondition interpretation of $r = True$ in the *Write* operation can determine the future behaviour of the *Buffer*. In the standard interpretation it is possible to weaken this condition, thus

$$
\begin{array}{|l}
\hline
\textit{RWrite} \\
\hline
\Delta \textit{Buffer} \\
x? : \textit{MSG} \\
\hline
\textit{buffer}' = \textit{buffer} \oplus \{1 \mapsto x?\} \\
r' = \textit{False} \\
\hline
\end{array}
$$

is a valid refinement. However, this makes the *Buffer* asynchronous. The guarded interpretation would have forbidden such refinement. On the other hand, the guarded interpretation does not permit the less problematic and possibly desired refinement *RRead*1. The standard interpretation, unfortunately allows the dangerous refinement *RRead*2 which suddenly permits to read an empty buffer.

$$
\begin{array}{|l}
\hline
\textit{RRead1} \\
\hline
\Delta \textit{Buffer} \\
x! : \textit{MSG} \\
\hline
\textit{buffer} \neq \langle\rangle \\
x! = \textit{head buffer} \\
r' = \textit{True} \\
\hline
\end{array}
\qquad
\begin{array}{|l}
\hline
\textit{RRead2} \\
\hline
\Delta \textit{Buffer} \\
x! : \textit{MSG} \\
\hline
x! = \textit{head buffer} \\
r' = \textit{True} \\
\hline
\end{array}
$$

Using just the guarded or the precondition interpretation is not always suitable for practical tasks. Like in the *Read* operation where two conditions have different statuses it is difficult to determine which interpretation to choose. After choosing one interpretation, however, refinement can behave in an unwanted fashion not treating the meaning of all given conditions correctly. Specifications should be foremost intuitive, thus we propose to combine guards and preconditions in a single notation.

3.4.5 Proposal

Guards block an operation thus rendering it impossible outside its guard and implicitly do not allow a state change to occur. Preconditions permit operations and guarantee its outcome. Having both, enables the specification of underdefinedness as those situations where the guard permits the operation but the precondition fails, thus no explicit outcome is defined. These three situations give rise to an intuitive semantics based on three logical truth values. Thus, we propose a non-standard semantics of operations, based on a three-valued logic.

However, such an interpretation of operations requires a more expressive notation than normal operations with explicit guards. Thus, we propose to develop a syntax which is sufficiently expressive for this semantics. Using a three-valued logic will also lead to a simple and intuitive notion of operation refinement, where refinement is reduction of underdefinedness. We will define operation refinement rules for this which generalise the traditional ones. Furthermore, we propose an adaption of the schema calculus, based on three-valued logic, to account for the extended syntax.

3.5 Summary

Our aim is to investigate the formal support that can be given to analyse inconsistent specifications written in the Z notation. This includes also the process of refinement in the presence of inconsistencies. We propose to adopt one of the logics that facilitate the process of reasoning in the presence of inconsistency without leading to triviality, the so called paraconsistent logics.

Some of the investigated logics also provide a truth value for handling underdefinedness. Operations in Z are, in general, partial descriptions. If the precondition of an operation holds, the specified results are guaranteed. However, if the precondition is not satisfied there are two interpretations possible. On the one hand, in the standard interpretation everything can happen. Note, this notion also relates to triviality. On the other hand, the operation can be blocked and thus no state change occurs.

We propose to use the extra truth value to represent underdefinedness. This enables us to construct an intuitive semantics for operations containing both guards and preconditions. Underdefinedness is then characterised as the region between the guard and the precondition of an operation.

Chapter 4

Paraconsistency and First-Order Quasi-Classical Logic

The Z notation is based on classical first-order predicate logic. The problems arising from inconsistencies in Z specifications can be attributed to the way classical logic handles contradictions. In particular, given a single contradiction in a classical theory, it is possible to derive any formula from that theory. Thus, to formally manage inconsistencies in Z specifications we can look at some general approaches of managing inconsistency in logical formulae.

The group of logics which support the process of useful reasoning despite the presence of inconsistencies are called paraconsistent logics. This group can be further subdivided according to the kind of weakening of the logic used. For example, some logics use a different negation operator, some change the meaning of implication, sometimes new truth values are introduced, and sometimes the proof theory of the logic is altered. However, the common aim is to develop a paraconsistent logic as close as possible to classical logic.

One such paraconsistent logic is called quasi-classical logic (QCL). QCL has been introduced by (Besnard and Hunter, 1995) and fully developed in (Hunter, 2000) and (Hunter, 2001). In QCL the meaning of all the logical operators remains unchanged. Furthermore, the deduction rules within the proof theory of QCL are classical, too. These properties suggest that QCL is a prime candidate for a logic to support reasoning in the presence of inconsistencies in formal specification.

In this chapter we review some of the approaches of reasoning with inconsistent and incomplete knowledge. We focus on the presentation of paraconsistent logics, in particular quasi-classical logic, as they offer a novel approach to reasoning about inconsistencies in Z. Some of these logics are also meant to deal with incomplete knowledge. This is relevant for our work on underdefinedness in Z.

4.1 Introduction

In the last chapter we found that software development requires a new approach to handling inconsistencies which is not only based on detecting and eradicating them but on managing the information provided. This is required because inconsistencies frequently appear in large projects and sometimes it might not even be possible in practice to reach a completely consistent specification.

In fact, inconsistencies are a matter of every day life. We are constantly challenged by contradicting information. Sometimes we are able to resolve the inconsistency right away; sometimes, however, we have to live with inconsistencies. In such a case we tend not to derive any useless results from it. Often it is quite the contrary and inconsistencies lead to new discoveries. This process suggests that the logic we use to reason in everyday life is able to deal with inconsistencies in a useful manner. Such practical reasoning from inconsistent information is, however, not well supported by classical logic.

The Z notation is a specification language which is based on classical logic. Thus, Z is limited by its logic to deal with inconsistencies usefully and not to derive arbitrary conclusions. This problem has been recognised by researchers on formal logics and they developed so called paraconsistent logics. These logics reject the classical principle of explosion, often referred to as *Ex contradictione quodlibet*, i.e. from a contradiction follows everything.

Paraconsistent logics provide an interesting alternative to classical logic for reasoning about inconsistent theories. However, all paraconsistent logics are weaker than classical logic in either their logical connectives or in the derivation rules. Thus, it is not possible to simply replace the standard logic of Z with a paraconsistent one but it is required to investigate the impact of such a change carefully.

4.1.1 Motivation

The aim of this chapter is to introduce the notion of paraconsistent reasoning and some paraconsistent logics. Thus we provide the formal background for the following chapters. Paraconsistency emphasizes a shift of concern from contradictory to trivial theories. It is triviality that we most dislike in formal reasoning because it has no restrictions and does not distinguish between different contradictions. Paraconsistency, however, allows to differentiate between contradictions. As a result, one inconsistency does not corrupt all information. Hence, it facilitates more useful conclusions in the presence of inconsistency than classical logic.

There are many different ways to construct a paraconsistent logic. We present some of the approaches to give some insight into the development of paraconsistent logics and into the limitations they can possess. Thus, we build a foundation for an informed decision on which paraconsistent logic to select for our application towards analysing inconsistent specifications. It is out of the scope of this work to present a full overview of all the different paraconsistent logics. We recommend, for example, the collections by (Priest et al., 1989) and (Batens et al., 2000) for further information on this subject.

It is our aim to support both reasoning about overdefined and underdefined specifications. Many-valued logics, in particular four-valued ones, provide an intuitive semantics to capture the notions of over- and underdefinedness. Thus, we investigate two representatives of these group of logics further. We find them, unfortunately, unsuitable for our needs to reason about inconsistency but they do prove useful for our work on underdefinedness.

We present Hunter's quasi-classical logic in detail because we decided to apply it to reasoning about inconsistent specifications. One of the main advantages of QCL over other paraconsistent logics is that all connectives are interpreted classically as Boolean connectives and that the QC deduction rules hold in classical logic, too. The logic is, however, weaker than classical logic in the way it is used. We believe that QCL's advantage is vital for its acceptance as a new logic in such an established field as formal methods, because the specifiers need not change their way of writing specifications. Therefore, QCL is our prime candidate for a logic to support reasoning in the presence of inconsistencies.

4.1.2 Outline

This chapter is structured as follows. In Section 4.2 we cover some background on the notion of paraconsistency, including the different motivations for paraconsistency, two definitions of paraconsistency and the approaches to construct a paraconsistent logic. In Section 4.3 we present two four-valued paraconsistent logics, namely the logic $FOUR$ by (Belnap, 1977b) and the logic $\mathcal{FOUR}$ by (Damásio and Pereira, 1998). The main part of this chapter consists of Section 4.4 introducing quasi-classical logic by (Hunter, 2000). We contribute to the development of QCL by providing an extended discussion on logical equivalence presented in Section 4.4.5. We briefly summarize this chapter and discuss our choice for QCL in Section 4.5.

Note, we extend the work on QCL in Chapter 5 by introducing equality and we apply QCL to reason about inconsistent Z specification in Chapter 6. Furthermore, a three-valued subset of the logic $\mathcal{FOUR}$ is used in Chapters 7 and 8 to provide the semantics for our work on underdefinedness.

4.2 Inconsistency, Triviality and Paraconsistency

Before venturing into the presentation of some paraconsistent logics we need to establish some background on the notion of paraconsistency. There is first the issue of the motivation for paraconsistency. According to the different motivations there are several definitions of the term paraconsistency. Fortunately, there is at least one basic objective all paraconsistent logicians agree on, namely to avoid triviality. A brief investigation into the source of triviality leads to a categorisation of the different paraconsistent logics and provides also a motivation for the logics we present.

4.2.1 Motivations for Paraconsistent Logics

Paraconsistent logics are suitable for reasoning from inconsistent theories without collapsing into triviality. There are several motivations why such a logic is necessary. We provide a brief classification following (Urbas, 1990) of the different positions.

Dialetheism. According to (Priest, 1998): "A dialethia is a true contradiction, a statement, A, such that both it and its negation, $\neg A$, are true." Dialetheism is thus the position that some contradictions are true. This view rejects also the classically validated inference from inconsistent premises to an arbitrary conclusion.

The most common example of a dialethia is the "liar's paradox". Consider the sentence: "This sentence is not true." According to standard logic there are two possibilities, either the sentence is true or it is not. If the sentence is true, however, then what it says is correct, i.e. it is not true. Suppose the sentence is not true. But this is what the sentence says, i.e. it is true. Thus, in either case, the sentence is both true and not true.

Relevantism. The main interest for relevantist logicians is with the inference relation. They insist on a connection of relevance or commonality of content between the premises and conclusions. Though this is not directly related to the question of inconsistency it too restricts inferences from contradictory premises. The most notable representatives of relevantism are (Anderson and Belnap, 1975).

Pragmatism. This position recognises that there are many interesting systems that are inconsistent but non-trivial. This includes our beliefs and judgements, a range of scientific theories and legal codes. In fact, the likelihood of inconsistencies seems to increase with the expressiveness of the theories. Nevertheless, some mechanism prevents the deduction from arbitrary conclusions from such inconsistent theories. The pragmatic approach is not to abandon theories once they are discovered to be inconsistent but to accommodate them until a better alternative is found by means of a logic that functions plausibly in the presence of inconsistency. An important advocate of this motivation for paraconsistency is (da Costa, 1974). From the discussion in the last chapter it follows that we too subscribe to this pragmatic position.

4.2.2 Definition of Paraconsistency

The different motivations for paraconsistency lead almost naturally to different definitions of the terms paraconsistency and paraconsistent logic. (Béziau, 2000), for example, analyses some of the occurring definitions.

A theory T is a set of formulae expressed in some, normally formal, language which is closed under the consequence relation $\vdash$ of the underlying logic, i.e. if the formulae $A_1, \dots, A_n$ are in T and B is a consequence of $A_1, \dots, A_n$, denoted $\{A_1, \dots, A_n\} \vdash B$, then B is also in T.

The following is an intuitive definition of paraconsistency often presented in the literature. A theory is *inconsistent* if it contains some formula A together with its negation $\neg A$, i.e. there is an A such that $T \vdash A$ and $T \vdash \neg A$, where $\neg$ is a negation connective which is intended as a "contradiction-forming operator". A theory is *trivial* if it contains every formula of its language, i.e. for every A it holds $T \vdash A$, otherwise T is said to be non-trivial. A theory T is *paraconsistent* if it is inconsistent and non-trivial. A logic is paraconsistent if it supports the study of paraconsistent theories.

This definition, however, has been generalised because it requires the consequence relation to be transitive to ensure non-triviality. Thus, the minimal and most widely accepted definition amongst the paraconsistent logicians is now based on the rejection of the principle know as

ex contradictione quodlibet (ECQ)

i.e. from a contradiction follows everything. Based on the equivalence of falsehood and contradiction in classical logic this principle is also commonly referred to as: "ex falso quodlibet".

The formalisation of the principle of ECQ is that for any theory T and formulae A and B it follows $T \cup \{A, \neg A\} \vdash B$. The same principle without mentioning the theory T is just a special case of it. A logic is paraconsistent if it rejects ECQ, i.e. if not every formula B follows from an inconsistent premise ($T \cup \{A, \neg A\} \nvdash B$). Otherwise the logic is said to be explosive or trivialising.

4.2.3 Approaches to Paraconsistency

One can imagine that there are many different ways to avoid ECQ. All proposed solutions are based on some kind of weakening of classical logic.

Lewis's Proof of Ex Contradictione Quodlibet

The principle of ECQ is central to the notion of paraconsistency, thus a detailed analysis on how it arises is appropriate. The proof of ECQ by (Lewis and Langford, 1932) provides some insight. It proceeds by deploying various classical reasoning rules:

(1)	$p \wedge \neg p$	Assumption
(2)	p	by 1, $\wedge$-Elimination
(3)	$\neg p$	by 1, $\wedge$-Elimination
(4)	$p \vee q$	by 2, $\vee$-Introduction
(5)	q	by 3,4, $\vee$-Elimination

This derivation can be prevented, by blocking any of the rules in line (2), (3), (4) or (5). Thus various strategies are open to weaken classical logic.

The most common proposal is to reject (5), i.e. $\vee$-Elimination which is also called disjunctive syllogism. Consequently, if implication $A \Rightarrow B$ is defined in the usual way as $\neg\ A \vee B$ then modus ponens fails, too. For example, the logics by (da Costa, 1974) and (Belnap, 1977a) both reject disjunctive syllogism. However, modus ponens is valid in (da Costa, 1974) because implication cannot be expressed in terms of disjunction and negation.

The other two options are to block $\vee$-Introduction, favoured by logicians interested in analytic implication, and to block $\wedge$-Elimination, as investigated by so called connexive logicians. Note, for example, that the logic by (Belnap, 1977a) does not support $\wedge$-Elimination either. Thus, a combination of these options can also occur.

Another approach is not to generally block any of the rules but to restrict the ordering in which these rules can be applied. The derivation above requires $\vee$-Introduction to be applied before $\vee$-Elimination. The logic by (Besnard and Hunter, 1995), for example, is based upon the restriction that decompositional rules like $\vee$-Elimination must not be applied after $\vee$-Introduction. The advantage is to keep all classically valid reasoning rules including disjunctive syllogism and the classical definitions of the logical operators.

Weakly Negative Systems

In classical logic, the conflict $A \wedge \neg A$ is equivalent to falsity, often denoted $\perp$. More generally, if A and B are two formulae, then $A \Rightarrow (B \Rightarrow \perp)$ expresses that A and B are in conflict, i.e. they are inconsistent. Conflict can be represented in classical logic by using a negation symbol. Then $\{A\} \vdash \neg B$ represents the same inconsistency as above. Thus, negation and inconsistency are closely related in classical logic.

This type of reasoning lead to much research into the nature of negation. (Gabbay and Hunter, 1999), for example, explore the relationship between negation and contradiction to develop better techniques for handling inconsistent information. (Béziau, 2000) is also mainly concerned with the negation operator with respect to paraconsistency. Thus, it is not surprising that a number of paraconsistent logics are based on a weaker notion of negation than classical logic.

One important representative is the logic C_ω proposed by (da Costa, 1974). The main idea is to use the positive part of some logic, say classical or intuitionistic, but to allow negation in an interpretation to behave non-truth-functionally, i.e. the truth value of $\neg A$ is independent of that of A. This, in particular, allows both to take the value 1, i.e. both can be "true". Negation is rather weak under such an interpretation. Many classical equivalences, like the definition law for implication, double negation and the contraposition law do not hold in C_ω. Furthermore, rules like modus tollens and disjunctive syllogism fail. However, modus pones is valid and therefore weakly-negative logics are considered useful for rule-based reasoning with information.

Many-Valued Systems

Problably one of the simplest and intuitive ways to produce paraconsistent systems is to use a many-valued logic, i.e. a logic with more than two truth values. The formulae that hold in a many-valued interpretation are those which have a truth value that is said to be "designated". A paraconsistent many-valued logic is thus one which allows both a formula and its negation to be designated. The simplest form is to use three truth values, namely "true" and "false", which function in a classical way, and "both". One can also add a fourth value, "neither", to capture the problem of incomplete knowledge. We present two representatives of such four-valued logics next.

4.3 Four-Valued Paraconsistent Logics

Many-valued systems are rather intuitive. They provide a natural way of dealing with over-determined and under-determined knowledge. It is mainly the established Western philosophy that rejects extra truth values. Eastern philosophy, on the contrary, is founded on four truth values.

(deCharms, 1997, p. 26), for example, discusses the Tibetan view of mind. "For many Westerners [and classical logicians] these [following] two statements would seem to cover all of the relevant possibilities, with one or the other (but not both) being necessarily correct."

(1) A phenomenon exists (has individual existence).
(2) The phenomenon does not exist.

"From the Tibetan viewpoint, there are two additional possible (and philosophically important) viewpoints"

(3) The phenomenon both exists and does not exist.
(4) The phenomenon neither exists nor does not exist.

Thus, the Tibetan view corresponds to a four-valued approach as presented below.

4.3.1 Belnap's Logic FOUR

(Belnap, 1977b; Belnap, 1977a) introduces "A Useful Four-Valued Logic" to capture the idea of "How A Computer Should Think". Belnap considers the following situation. First, the reasoner is a computer and, therefore, need not to rely on familiarity with classical logic. Second, the computer answers questions based on given facts and deductions. Third, the facts the computer has, were given to it, which means, the computer can only reason about what it was told, i.e. about epistemic information.

The latter is surely the case in requirements engineering because the specifier usually has to accept what was told to him. This computer, however, is not a complete reasoner in the sense that it will not do anything else but report an inconsistency. This means, no automated belief revision will take place. Considering its application in requirements engineering, this is not a problem because it forces the specifier to go back and to discuss certain issues further with the applicant.

Truth Values

First, we fix the truth values of the logical system. Based on the epistemic information a computer is given, we have four situations: 'told True', 'told False', 'told True and False', and 'told neither True nor False'. Note, this corresponds to the subsets obtained by forming the powerset of the classical truth values. The truth values are given by the set $\{t, f, \top, \bot\}$ respectively.

These truth values can be ordered according to the amount of knowledge or information that each truth value exhibits. This ordering is denoted $\prec_k$ and it holds: $\bot \prec_k f \prec_k \top$, and $\bot \prec_k t \prec_k \top$. It can be observed that the four truth values form a complete lattice under the knowledge (or information) ordering.

A complete lattice is a set, for example A, on which a partial ordering $\preceq$ exists and for arbitrary subsets X of A there always exists least upper bounds $\sqcup X \in A$ and greatest lower bounds $\sqcap X \in A$. A function f from one complete lattice into another is monotonic if it preserves the lattice ordering, i.e. $a \preceq b$ implies $f(a) \preceq f(b)$. We need this property to explain how the truth tables for this logic arise.

Truth Tables

Table 4.1 presents the truth tables for Belnap's logic. In case there is no contradiction or incompleteness present, everything should be as in classical logic. Furthermore, all these truth functions shall be monotonic on the lattice over the knowledge ordering. This, however, does not determine all resulting truth values. It turns out that a minimal relationship between conjunction and disjunction is needed to uniquely determine every value in the truth tables. The natural relation is the following, classical, equivalence:

$$a \wedge b = a \Leftrightarrow a \vee b = b$$
$$a \wedge b = b \Leftrightarrow a \vee b = a$$

i.e. having $\wedge$ as greatest lower bound and $\vee$ as least upper bound of the lattice.

The truth values for the negation of $\top$ and $\bot$ are forced by monotonicity of negation over the knowledge ordering and $\top$ and $\bot$ in the truth tables for conjunction and disjunction are also forced by monotonicity. Furthermore, t is an identity element with respect to conjunction, i.e. $a \wedge t = a$. Thus $a \vee t = t$ must hold by the above obligation. Similar considerations fill in the rest of the tables except the corners. They are, again, forced by monotonicity. Since $f \preceq_k \top$ it follows by monotonicity that $(f \wedge \bot \preceq_k \top \wedge \bot)$ and hence $f \preceq_k (\top \wedge \bot)$. Similarly, $\bot \preceq_k f$ leads to $(\top \wedge \bot) \preceq_k (\top \wedge f)$, i.e. $(\top \wedge \bot) \preceq_k f$, and by antisymmetry $(\top \wedge \bot) = f$.

Therefore, we derive the following truth tables for negation, conjunction and disjunction:

A	$\neg A$
$\top$	$\top$
t	f
f	t
$\bot$	$\bot$

$\wedge$	$\top$	t	f	$\bot$
$\top$	$\top$	$\top$	f	f
t	$\top$	t	f	$\bot$
f	f	f	f	f
$\bot$	f	$\bot$	f	$\bot$

$\vee$	$\top$	t	f	$\bot$
$\top$	$\top$	t	$\top$	t
t	t	t	t	t
f	$\top$	t	f	$\bot$
$\bot$	t	t	$\bot$	$\bot$

Table 4.1: Negation, Conjunction, and Disjunction of the Logic FOUR

These tables constitute the so called logical lattice, denoted **L4**, with the following, related truth ordering: $f \prec_t \top \prec_t t$, and $f \prec_t \bot \prec_t t$. The truth ordering reflects the difference in the "measure of truth" that every value represents. A double Hasse diagram of both knowledge and truth ordering of the logic FOUR is given in Figure 4.1.

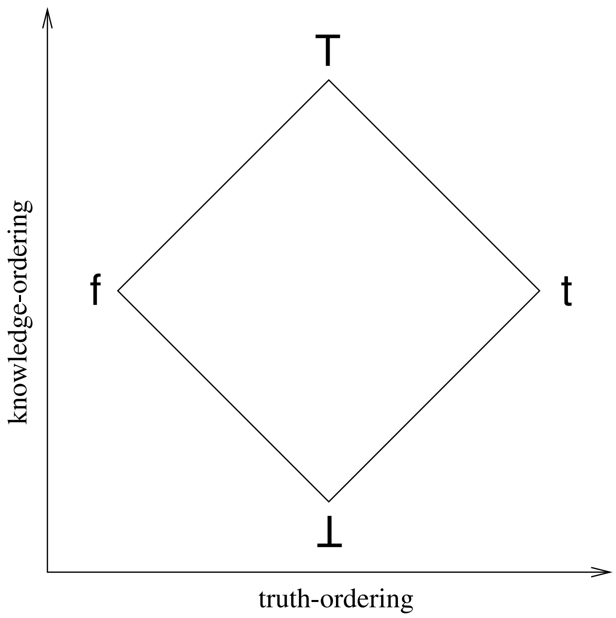


Figure 4.1: The Truth and Knowledge Ordering of FOUR

The propositional language of Belnap's logic is composed of a countable set of propositional letters and the logical connectives $\neg$, $\wedge$ and $\vee$. Formulae in this logic are constructed in the standard way. A Belnap theory is a set of formulae in this logic. For the finite case, a Belnap theory can be seen as a single formula given by the conjunction of all the formulae in that particular theory. For example, the formula $p \wedge (\neg\, p \vee q) \wedge (r \wedge \neg\, q)$ is a finite theory in this logic.

Semantics

We define the semantics of this logic in the normal way by using the notion of an interpretation mapping from the propositional symbols into the set of truth values as well as truth-valuation for a generalisation to arbitrary formulae. Interpretations are ordered by the usual extension to sets of literals of the knowledge ordering among literals. Furthermore, we use the notion of designated truth values from many-valued logic.

Let I be an interpretation in the logic FOUR and val_I the corresponding truth-valuation (Belnap uses the term set-up for I). Let F be an arbitrary propositional formula containing $\neg$, $\vee$, $\wedge$. We say that I satisfies F, denoted by $I \models_4 F$, iff $val_I(F) \in \{t, \top\}$, where $\{t, \top\}$ forms the set of designated truth-values. An interpretation I is a model of a theory iff it satisfies all the formulae in the theory. $I \not\models_4 F$ denotes that I does not satisfy F.

$$
\begin{array}{lllll}
I \models_4 A \wedge B & \text{iff} & I \models_4 A & \text{and} & I \models_4 B \\
I \models_4 A \vee B & \text{iff} & I \models_4 A & \text{or} & I \models_4 B \\
I \models_4 \neg A & \text{iff} & I \not\models_4 A & &
\end{array}
$$

The notion of entailment is based on the partial ordering associated with the logical lattice. In **L4** entailment goes up hill. That means, a sentence A entails or implies a sentence B iff for each assignment of one of the truth-values to variables, the value of A does not exceed the value of B, in symbols:

$$A \text{ entails } B \text{ iff } val_I(A) \preceq_t val_I(B) \text{ for every interpretation } I$$

Proof Theory

Proof theoretically, Belnap's logic is characterised by a finite axiomatization. Given are the formulae A, B and C consisting of $\wedge$, $\vee$, and $\neg$. The expression $A \to B$ denotes that A entails B, i.e. that the inference from A to B is valid. The expression $A \leftrightarrow B$ denotes that A and B are semantically equivalent. The following axiomatization is known to be sound and complete with respect to the semantics of the logic presented earlier.

$$
\begin{gathered}
A_1 \wedge \ldots \wedge A_m \to B_1 \vee \ldots B_n \text{ provided some } A_i \text{ is some } B_j \text{ (sharing)} \\
A \to B \text{ and } B \to C \text{ implies } A \to C \\
A \leftrightarrow B \text{ and } B \leftrightarrow C \text{ implies } A \leftrightarrow C \\
A \to B \text{ iff } \neg B \to \neg A
\end{gathered}
$$

$$\neg\neg A \leftrightarrow A$$

$$\begin{array}{rl} \neg(A \wedge B) \leftrightarrow \neg A \vee \neg B & \quad \neg(A \vee B) \leftrightarrow \neg A \wedge \neg B \\ A \vee B \leftrightarrow B \vee A & \quad A \wedge B \leftrightarrow B \wedge A \\ A \vee (B \vee C) \leftrightarrow (A \vee B) \vee C & \quad A \wedge (B \wedge C) \leftrightarrow (A \wedge B) \wedge C \\ A \wedge (B \vee C) \leftrightarrow (A \wedge B) \vee (A \wedge C) & \quad A \vee (B \wedge C) \leftrightarrow (A \vee B) \wedge (A \vee C) \end{array}$$

$$\begin{array}{c} (A \vee B) \rightarrow C \text{ iff } A \rightarrow C \text{ and } B \rightarrow C \\ A \rightarrow (B \wedge C) \text{ iff } A \rightarrow B \text{ and } A \rightarrow C \\ A \rightarrow B \text{ iff } A \leftrightarrow (A \wedge B) \text{ iff } B \leftrightarrow (A \vee B) \end{array}$$

The first block of expressions captures the reflexivity, transitivity and contrapositive properties of the consequence relation. The second block of expressions corresponds to standard classical properties of negation, disjunction and conjunction (e.g., commutativity, associativity, de Morgan laws). Finally, the last expressions correspond to standard classical rules for introduction and elimination of $\vee$ and $\wedge$ respectively.

The similarity between the above rules and classical rules shows that this four-valued logic is very close to standard classical logic. However, the following 'paradoxes of implication' are not derivable, nor semantically valid, from the set of entailment rules: $A \wedge \neg A \rightarrow B$ and $A \rightarrow B \vee \neg B$. This means that the problem of triviality was resolved and, thus, Belnap's logic is paraconsistent.

Belnap's logic is strictly weaker than classical logic as it does not incorporate modus ponens nor $\wedge$-Elimination. Furthermore, implication cannot be defined in terms of the other logical operators, nor does the deduction theorem hold. This logic is, however, "normal" because the Tarskian properties of reflexivity, monotonicity and transitivity hold.

Beyond Belnap

Belnap's four-valued logic had a great impact on the research of paraconsistent logics and it had been a constant source for further investigations. (Rodrigues and Russo, 1998), for example, present a translation method for Belnap's logic into first-order predicate logic based on two principle predicates $holds(A, tt)$ and $holds(A, ff)$ for any formula A. (Arieli and Avron, 1998) use Belnap's logic as a basis for a discussion on the general usefulness of four truth values. They find that four values are just right. They are strictly more expressive than three truth values but incorporate the investigated three-valued logics. There are also a number of related approaches to Belnap's logic, one of which is presented next.

4.3.2 Damasio's Logic $\mathcal{FOUR}$

Belnap's logic does not validate the use of modus ponens. However, this is an often applied reasoning rule. Thus, Belnap's logic is not suited for some applications, for example, in logic programming. To overcome this deficiency (Damásio and Pereira, 1998) present in their survey of paraconsistent semantics for logic programs a variation of Belnap's logic.

Truth Table for Implication

The interpretation of the logical connectives $\wedge$, $\vee$ and $\neg$ is the same as in Belnap's logic as given in Table 4.1. The logic $\mathcal{FOUR}$ by (Damásio and Pereira, 1998) then differs primarily in the definition of the consequence relation and the inclusion of the implication connective which is presented in Table 4.2.

$\rightarrow$	$\top$	t	f	$\bot$
$\top$	t	t	f	f
t	t	t	f	f
f	t	t	t	t
$\bot$	t	t	t	t

Table 4.2: Truth Table for Implication in the Logic $\mathcal{FOUR}$

Let I be a $\mathcal{FOUR}$ interpretation, val_I the corresponding truth-valuation and F an arbitrary formula. Then I satisfies F, denoted $I \models_4 F$, if and only if $val_I(F) \in \{t, \top\}$, where $\{t, \top\}$ forms the set of designated truth values. As usual, an interpretation I is a model of a theory T if and only if it satisfies all the formulae in T. Furthermore, $I \not\models_4 F$ denotes that I does not satisfy F.

Note that the implication operator above always evaluates to either t or f. It is defined in such a way that the following equivalences plus modus ponens are valid:

$$
\begin{array}{lllcl}
I \models_4 A \wedge B & \text{iff} & I \models_4 A & \text{and} & I \models_4 B \\
I \models_4 A \vee B & \text{iff} & I \models_4 A & \text{or} & I \models_4 B \\
I \models_4 A \rightarrow B & \text{iff} & I \not\models_4 A & \text{or} & I \models_4 B
\end{array}
$$

Note the similarities to (Herre and Pearce, 1992) and (Herre, 1998). Each of the two papers consider one half of this work. The first is concerned with partial logical programs and the latter with inconsistent logic programs. Both papers together can be used to extend the work by (Damásio and Pereira, 1998) to the first-order case.

Logical Equivalence

To characterise this logic further, we introduce the notion of equivalence. Actually, in multi-valued logics one can define at least two notions of equivalence, one based on the truth-valuation function (called strong equivalence and denoted $\equiv_4$) and another based on the consequence relation (referred to as weak equivalence, $\models\!\mid_4$). Given two formulae A and B of the language $\mathcal{FOUR}$, we say $A \equiv_4 B$ iff $val_I(A) = val_I(B)$ for every interpretation I. Furthermore, we say $A \models\!\mid_4 B$ iff for every interpretation I it holds $I \vDash_4 A$ iff $I \vDash_4 B$. Otherwise $A \models\!\mid_4 B$ is false.

Note, for an arbitrary many-valued logic it holds that *if* $A \equiv B$ *then* $A \models\!\mid B$, whenever $\equiv$ is defined as truth-value equality and $\models$ is expressed by means of a set of designated truth values. In the remainder of this subsection we mean weak equivalence when we just say equivalence.

The equivalences holding in $\mathcal{FOUR}$ are similar to the ones holding in classical logic. (Damásio and Pereira, 1998) present a list of valid equivalences. However, a number of laws do not hold, like the law of the excluded middle ($A \vee \neg A \models\!\mid t$), the law of contradiction ($A \wedge \neg A \models\!\mid f$), the definition law ($A \rightarrow B \models\!\mid \neg A \vee B$), i.e. the possibility to define implication in terms of the other connectives, and the contraposition law ($A \rightarrow B \models\!\mid \neg B \rightarrow \neg A$). Furthermore, modus tollens ($(\neg B \wedge A \rightarrow B) \rightarrow \neg A$) and disjunctive syllogism ($A \wedge (\neg A \vee B) \rightarrow B$) fail. Interestingly, all axioms of propositional logic hold but

$$(A \rightarrow B) \rightarrow ((A \rightarrow (\neg B)) \rightarrow (\neg A))$$

which corresponds to the introduction rule for negation of the natural deduction calculus. Finally, we note that the logic presented is neither daCosta's C_ω system, because the law of the excluded middle is not satisfied, nor Belnap's logic, because modus ponens is a sound rule now.

Logical Consequence

Given the above, we present the correspondence between the consequence relation (also called satisfaction relation) and the truth-valuation function of propositional symbols, as well as between the truth-valuation function and models in $\mathcal{FOUR}$. Let A be a propositional symbol and I an interpretation in a language containing A, then:

$$\begin{array}{llllll}
I \vDash_4 A & \text{and} & I \vDash_4 \neg A & \text{iff} & val_I(A) = \top \\
I \vDash_4 A & \text{and} & I \nvDash_4 \neg A & \text{iff} & val_I(A) = t \\
I \nvDash_4 A & \text{and} & I \vDash_4 \neg A & \text{iff} & val_I(A) = f \\
I \nvDash_4 A & \text{and} & I \nvDash_4 \neg A & \text{iff} & val_I(A) = \bot
\end{array}$$

This means, a literal L is entailed by an interpretation I iff $val_I(L)$ maps to t or $\top$. The complement of L, i.e. $\neg L$, holds iff $val_I(L)$ maps to f or $\top$.

To find the value of the truth-valuation function applied to the propositional symbol A, we construct the set of all possible $\mathcal{FOUR}$ models of a given theory T, i.e. $Mod^{\models_4}(T)$. Then, we take the least $\mathcal{FOUR}$ model M of $Mod^{\models_4}(T)$ with respect to the knowledge ordering, i.e. $M \in Mod^{\models_4}(T) \wedge \forall N \bullet N \in Mod^{\models_4}(T) \Rightarrow M \prec_k N$. The value of the truth-valuation of a propositional symbol A is then:

$$\begin{array}{llllll} val_I(A) = \top & \text{iff} & A \in M & \text{and} & \neg A \in M \\ val_I(A) = t & \text{iff} & A \in M & \text{and} & \neg A \notin M \\ val_I(A) = f & \text{iff} & A \notin M & \text{and} & \neg A \in M \\ val_I(A) = \bot & \text{iff} & A \notin M & \text{and} & \neg A \notin M \end{array}$$

The Tweety Example

Consider the rules (1)-(4) of the Tweety example which we presented in Chapter 3. By applying the equivalence rules of $\mathcal{FOUR}$ and modus ponens we can infer only one and thus least model:

$$M = \{penguin(Tweety), bird(Tweety), flies(Tweety), \neg flies(Tweety)\}$$

This, in turn, leads to the following assignments of truth values:

$$\begin{array}{l} val_I(flies(Tweety)) = \top \\ val_I(penguin(Tweety)) = t \\ val_I(bird(Tweety)) = t \end{array}$$

which corresponds to our introduction of Tweety as a penguin and bird that can and cannot fly.

4.4 Quasi-Classical Logic

The development of quasi-classical logic (QCL) was influenced by the need to handle beliefs rather than the truth. As such, it seems particularly suitable for reasoning about specifications because specifications are artifacts of belief. In general, a specification is a collection of information, often provided by multiple sources, on how a system which has yet to be developed should work. Therefore, belief in the information is crucial as there does not exist anything providing the ultimate truth about the future system. In such a context, the sources of information may possibly contradict on some issues and it may well be that such contradictions cannot be resolved immediately. Hence, there is a need for a logic dealing with inconsistent information.

The logics FOUR and $\mathcal{FOUR}$ were also designed to handle beliefs. Two extra truth values were introduced to capture inconsistency and incompleteness. We found, however, that practical reasoning rules, like modus pones and disjunctive syllogism, do not hold in these logics. Furthermore, the definition law, relating implication with negation and disjunction is not valid either. We think that specification developers would like to rely on classical correspondences and specification analysts prefer to rely on standard inference rules. Therefore, we require a paraconsistent logic that is more practical in such respects.

(Hunter, 2000) states that QCL has been developed for applications, in particular for reasoning about requirements specifications that might be inconsistent. For example, (Hunter and Nuseibeh, 1997) advocate and illustrate the use of QCL to handle and manage inconsistent specifications. The specifications presented as examples in the work on QCL are written in first-order predicate logic. Our aim in this thesis is to utilise QCL to reason about inconsistent specifications written in a richer language, specifically the Z notation.

4.4.1 Syntax of Quasi-Classical Logic

To the reader familiar with first-order predicate logic (FOPL) only little will be new in this section. For those who like to recapitulate FOPL we recommend (Fitting, 1996) or (Ben-Ari, 2001) for a short introduction. Both text books present an introduction to predicate logic and, in particular, to the tableau method which we use later, too.

The language of quasi-classical logic is that of first-order predicate logic. It is defined in the usual way. We start by presenting the alphabet of the language. Based on the alphabet, we define the notions of a term, an atomic formula and, finally, formulae belonging to the language of QCL.

Alphabet. The alphabet of the language of quasi-classical logic consists of: the common logical connectives, like $\wedge, \vee, \Rightarrow, \Leftrightarrow$ and $\neg$, including the two quantifiers $\forall$ and $\exists$; a set of variables; a set of predicate symbols; a set of function symbols; and, finally, some punctuation symbols, like '(' and ')', used to form formulae. Each relation and function symbol is associate with a positive integer, its arity. Function symbols with arity zero are also called constant symbols. We assume that there is at least one constant symbol in the set of function symbols. Note, the Boolean constants *true* and *false* are not given in the QC language.

Term. The basic building block for a formula is a term. First, any variable is a term and, second, if f is an n-ary function symbol with $n \geq 0$ and $t_1, \ldots, t_n$ are terms then $f(t_1, \ldots, t_n)$ is a term, too. Note, that it follows from the second case that constant symbols are terms as well. For example, if $+$ is a two-place function symbol, ^{2} is a one-place function symbol, x and y are variables, and 0 and 1 are constants, then $x + y, x^2 + 1, (1 + 0)^2, ((x + y)^2 + (1 + y)^2)^2, \ldots$ are terms. Sometimes we may use the infix notation for writing terms, like in the example above. For instance, we write $x + y$ rather than $+(x, y)$.

Atom. Having defined terms we move on to define formulae. The simplest of its kind is an atomic formula, also called an atom. If P is an n-ary predicate symbol with $n \geq 0$ and $t_1, \ldots, t_n$ are terms, then $P(t_1, \ldots, t_n)$ is an atomic formula.

Formulae. Given atomic formulae we use the logical operators available to construct more complicated formulae. Formulae are well-formed if they meet the following conditions. First, any atom is a formula and, second, if ϕ and ψ are formulae and x is a variable then the following are also formulae: $(\neg\phi)$, $(\phi \wedge \psi)$, $(\phi \vee \psi)$, $(\phi \Rightarrow \psi)$, $(\phi \Leftrightarrow \psi)$, $(\forall\, x.\phi(x))$, $(\exists\, x.\phi(x))$.

We let $\mathcal{L}$ denote a set of formulae formed in such an inductive way. For later reference we introduce some more vocabulary. Any atomic formula or any negation of an atomic formula is called a literal. A disjunction of literals is called a clause. A term or an atomic formula is ground if and only if it contains no variables and a sentence is a formula with no free-variable occurences. Furthermore, we omit brackets according to the general conventions.

The notion of a focus is possibly new to those acquainted with FOPL. The focus is a syntactical rule to remove a particular disjunct from a clause. We use the focus later as a means to introduce a particular form of disjunction with a built-in resolution rule.

Focus. Let $\alpha_1 \vee \ldots \vee \alpha_n$ be a clause that includes a literal α_i. The focus of $\alpha_1 \vee \ldots \vee \alpha_n$ by α_i, denoted $\otimes(\alpha_1 \vee \ldots \vee \alpha_n, \alpha_i)$, is defined as the clause obtained by removing the disjunct α_i from the clause $\alpha_1 \vee \ldots \vee \alpha_n$. In the case of a clause with just one disjunct we consider the focus to be undefined.

Basically, the focus of a clause is just the original formula without a particular disjunct. For example, consider the clause $\alpha \vee \beta \vee \gamma$, then the focus of this clause by β, denoted $\otimes(\alpha \vee \beta \vee \gamma, \beta)$, is $\alpha \vee \gamma$. The focus $\otimes(\alpha \vee \alpha, \alpha)$ is undefined, because $\alpha \vee \alpha$ contracts to α.

4.4.2 Semantics of Quasi-Classical Logic

One of the main ideas behind some paraconsistent logics is to separate the truth and falsehood of a formula from each other, i.e. knowing the formula φ is true does not necessarily imply that φ is not false. Quasi-classical logic follows the same approach. Basically, we construct a set of all possible atomic formulae that can be built using the symbols in the set of assumptions. Any such set is a possible model. Then, we define two semantic relations, called strong and weak satisfaction to interpret QC formulae. Finally, we define the quasi-classical satisfaction relation based on strong and weak satisfaction.

Quasi-Classical Model

The notion of a model in first-order quasi-classical logic is based on a form of Herbrand models. Herbrand models are special in the sense that they associate each ground term with its name. Every model has a domain, which in this case is called the Herbrand universe.

Definition 4.4.1 (Herbrand Universe)
The Herbrand universe $U(\mathcal{L})$ for a set of formulae $\mathcal{L}$ is the set of ground terms that can be formed using the function and constant symbols in $\mathcal{L}$. As mentioned before, we can always assume that there exists a constant symbol. If there is none we add one, say c.

For example, consider the set of formulae $\mathcal{L} = \{Q(a), P(a, f(x), g(y, b))\}$ with predicate symbols P, Q, function symbols f, g, constants a, b, and variables x, y. Then $U(\mathcal{L}) = \{a, b, f(a), f(b), f(f(\ldots(f(a))\ldots)), g(a, a), g(a, b), \ldots\}$ is the Herbrand universe of $\mathcal{L}$. Note, if $\mathcal{L}$ contains a function symbol with arity greater than zero then the Herbrand universe is infinite. The Herbrand universe of the set of ground formulae $\Delta = \{\neg P(a), P(a) \vee P(b), P(a) \vee \neg P(b)\}$ with predicate symbol P and constants a and b is $U(\Delta) = \{a, b\}$.

Definition 4.4.2 (Herbrand Base)
Given is the Herbrand universe $U(\mathcal{L})$ for a set of formulae $\mathcal{L}$. The Herbrand base $B(\mathcal{L})$ is the set of ground atoms that can be formed using the predicate symbols in $\mathcal{L}$ and the terms in $U(\mathcal{L})$.

For example, the Herbrand base for the set of formulae $\mathcal{L}$ from above $B(\mathcal{L}) = \{Q(a), Q(f(a)), P(a, f(a), g(b, b)), P(f(a), b, a), ...\}$. The Herbrand base $B(\Delta)$ of the set of formulae Δ is $B(\Delta) = \{P(a), P(b)\}$.

In standard predicate logic, every Herbrand model over $\mathcal{L}$ can be described as a subset of the Herbrand base $B(\mathcal{L})$. Because we deal with a paraconsistent logic, we need to go a step further. The idea in many paraconsistent logics is to separate formulae and their negation. To do so, we use a set of positive and negative objects constructed from the Herbrand base.

Definition 4.4.3 (Object)
Given is the Herbrand base $B(\mathcal{L})$ for a set of formulae $\mathcal{L}$. $O(\mathcal{L})$ is a set of objects defined as follows, where $+\alpha$ is a positive object, and $-\alpha$ is a negative object.

$$O(\mathcal{L}) = \{+\alpha \mid \alpha \in B(\mathcal{L})\} \cup \{-\alpha \mid \alpha \in B(\mathcal{L})\}$$

Consider the set of formulae Δ from above. The set of objects is given by $O(\Delta) = \{+P(a), -P(a), +P(b), -P(b)\}$. Any set of such positive and negative objects can be a quasi-classical model.

Definition 4.4.4 (Model)
Given a set of objects $O(\mathcal{L})$, then any $E \subseteq O(\mathcal{L})$ is called a model.

This means that a model E can contain both positive and negative objects. We consider the following meaning for positive objects $+\alpha$ and negative objects $-\alpha$ being in some model E or not:

$+\alpha \in E$ means α is "satisfiable" in the model

$-\alpha \in E$ means $\neg\, \alpha$ is "satisfiable" in the model

$+\alpha \notin E$ means α is not "satisfiable" in the model

$-\alpha \notin E$ means $\neg\, \alpha$ is not "satisfiable" in the model

This semantics can also be regarded as giving one of the four truth values Both, True, False and Neither to the elements of the Herbrand base, i.e. to the ground atoms, as in the four-valued logic by (Belnap, 1977b). For an atom α

α is Both if both α and $\neg\, \alpha$ are "satisfied"

α is True if α is "satisfied" and $\neg\, \alpha$ is not "satisfied"

α is False if α is not "satisfied" and $\neg\, \alpha$ is "satisfied"

α is Neither if neither α nor $\neg\, \alpha$ is "satisfied"

Hunter, however, introduces a different semantics based on a two-valued interpretation. To continue, we formalise the notion of satisfiability and extend it to formulae of the language using the following definitions.

Quasi-Classical Herbrand Interpretation

As usual, an assignment A is a function from the set of variables in $\mathcal{L}$ to the universe $U(\mathcal{L})$. Given an assignment A, an x-variant assignment B is the same assignment as A except perhaps in the assignment for x.

Definition 4.4.5
For an assignment A, terms in $\mathcal{L}$ are interpreted as follows, where $[.]^A$ is a function from the terms in $\mathcal{L}$ to $U(\mathcal{L})$.

$[c]^A = c$, where c is a constant symbol.

$[x]^A = x^A$, where x is a variable symbol.

$[f(t_1, \ldots, t_n)]^A = f([t_1]^A, \ldots, [t_n]^A)$, where f is a function symbol and $t_1, \ldots, t_n$ are terms.

Thus, each ground term in $\mathcal{L}$ is interpreted as the equivalent term in $U(\mathcal{L})$, hence a model with such an interpretation is a Herbrand model. A subset of the set of objects is a model of a particular literal, if the corresponding positive or negative object is a member of the model itself.

Definition 4.4.6 (Herbrand satisfaction)
Let $\models_h$ be a satisfiability relation called Herbrand satisfaction. For a model E and an assignment A, an atom $\alpha(t_1, \ldots, t_n)$ in $\mathcal{L}$ over terms $t_1, \ldots, t_n$ is interpreted as follows:

$$\begin{array}{lll}(E, A) \models_h \alpha(t_1, \ldots, t_n) & \text{iff} & +\alpha([t_1]^A, \ldots, [t_n]^A) \in E \\ (E, A) \models_h \neg\, \alpha(t_1, \ldots, t_n) & \text{iff} & -\alpha([t_1]^A, \ldots, [t_n]^A) \in E\end{array}$$

This definition of Herbrand satisfaction is the base case for the two satisfaction relations QCL is built upon. We continue by defining strong satisfaction first.

Strong Satisfaction Relation

The main idea behind QCL is that proofs in QCL are a two-stage affair. A proof is separated into decompositional steps, including resolution, followed by compositional steps. To capture this idea we need to establish the semantics for both stages. Here we present the notion of strong satisfaction which corresponds to the decompositional phase.

Definition 4.4.7 (Strong satisfaction)
Let $\models_s$ be a satisfiability relation called strong satisfaction. For a model E, and an assignment A, we define $\models_s$ as follows, where $\alpha_1, \ldots, \alpha_n$ are literals in $\mathcal{L}$, and α is a literal in $\mathcal{L}$.

$$\begin{array}{l}(E, A) \models_s \alpha \text{ iff } (E, A) \models_h \alpha \\ (E, A) \models_s \alpha_1 \vee \ldots \vee \alpha_n \text{ iff} \\ \qquad [(E, A) \models_s \alpha_1 \text{ or } \ldots \text{ or } (E, A) \models_s \alpha_n] \text{ and} \\ \qquad\quad \forall\, i \text{ s.t. } 1 \leq i \leq n\ [(E, A) \models_s \neg\, \alpha_i \text{ implies} \\ \qquad\qquad\qquad (E, A) \models_s \otimes(\alpha_1 \vee \ldots \vee \alpha_n, \alpha_i)]\end{array}$$

We clarify the meaning of this disjunction rule with an example. If $\alpha \vee \beta$ is the given clause, then the above definition reduces to

$$\begin{aligned}(E,A) \models_s \alpha \vee \beta \quad \text{iff} \quad & [(E,A) \models_s \alpha \ \text{ or } (E,A) \models_s \beta] \\ & \text{and } [(E,A) \models_s \neg\,\alpha \ \text{ implies } (E,A) \models_s \beta] \\ & \text{and } [(E,A) \models_s \neg\,\beta \ \text{ implies } (E,A) \models_s \alpha]\end{aligned}$$

Strong satisfaction is more restricted than classical satisfaction because the link between a formula and its negation has been decoupled. To provide a meaning for resolution, this link is put into the semantics of strong satisfaction via the treatment of disjunction.

(Hunter, 2000) provides a slightly different view on disjunction, too. Given a model E and literals $\alpha_1, \ldots, \alpha_n$, then

$$\begin{aligned}& E \models_s \alpha_1 \vee \ldots \vee \alpha_n \ \text{ iff} \\ & \qquad \text{(1) for some } \ \alpha_i \in \{\alpha_1, \ldots, \alpha_n\}, +\alpha_i \in E \ \text{ and } \ -\alpha_i \notin E \\ & \qquad \text{or (2) for all } \ \alpha_i \in \{\alpha_1, \ldots, \alpha_n\}, +\alpha_i \in E \ \text{ and } \ -\alpha_i \in E\end{aligned}$$

Hunter proves that both definitions are equivalent by expanding the above definition. In essence, the disjunction rule of strong satisfaction provides a semantic account for paraconsistent reasoning using resolution. We now continue defining strong satisfaction, considering arbitrary formulae.

Definition 4.4.7 (continued)
For formulae $\alpha, \beta, \gamma \in \mathcal{L}$, we extend the definition of strong satisfaction as follows:

$$\begin{aligned}&(E,A) \models_s \alpha \wedge \beta \ \text{ iff } \ (E,A) \models_s \alpha \ \text{ and } \ (E,A) \models_s \beta \\ &(E,A) \models_s \neg\,\neg\,\alpha \vee \gamma \ \text{ iff } \ (E,A) \models_s \alpha \vee \gamma \\ &(E,A) \models_s \neg\,(\alpha \wedge \beta) \vee \gamma \ \text{ iff } \ (E,A) \models_s \neg\,\alpha \vee \neg\,\beta \vee \gamma \\ &(E,A) \models_s \neg\,(\alpha \vee \beta) \vee \gamma \ \text{ iff } \ (E,A) \models_s (\neg\,\alpha \wedge \neg\,\beta) \vee \gamma \\ &(E,A) \models_s \alpha \vee (\beta \wedge \gamma) \ \text{ iff } \ (E,A) \models_s (\alpha \vee \beta) \wedge (\alpha \vee \gamma) \\ &(E,A) \models_s \alpha \wedge (\beta \vee \gamma) \ \text{ iff } \ (E,A) \models_s (\alpha \wedge \beta) \vee (\alpha \wedge \gamma) \\ &(E,A) \models_s (\alpha \Rightarrow \beta) \vee \gamma \ \text{ iff } \ (E,A) \models_s (\neg\,\alpha \vee \beta) \vee \gamma\end{aligned}$$

Let B be an x-variant assignment of A, then

$$\begin{aligned}&(E,A) \models_s (\exists\, x.\alpha(x)) \vee \beta \ \text{ iff } \ \text{for some } B,\ (E,B) \models_s \alpha \vee \beta \\ &(E,A) \models_s (\forall\, x.\alpha(x)) \vee \beta \ \text{ iff } \ \text{for all } B,\ (E,B) \models_s \alpha \vee \beta \\ &(E,A) \models_s (\neg\ \exists\, x.\alpha(x)) \vee \beta \ \text{ iff } \ \text{for all } B,\ (E,B) \models_s \neg\,\alpha \vee \beta \\ &(E,A) \models_s (\neg\ \forall\, x.\alpha(x)) \vee \beta \ \text{ iff } \ \text{for some } B,\ (E,B) \models_s \neg\,\alpha \vee \beta\end{aligned}$$

For a model E we polymorphically extend strong satisfaction as follows

$$E \models_s \varphi \ \text{iff} \ \text{ for all assignments } A,\ (E,A) \models_s \varphi$$

Such an E is said to be a strong model of φ.

For example, $\{-a, +a, -b, +b\}$ is the only strong model of the set of ground formulae $\Delta = \{\neg\, a, a \vee b, a \vee \neg\, b\}$. Note, every formula φ has a strong model even if it is classically inconsistent.

In the definition of strong satisfaction, the disjunction rule applies only to clauses. We show that this restriction is necessary. We demonstrate on an example that a weakening of this rule to arbitrary formulae leads to a contradiction. Consider, for example, the propositional model $E = \{+\beta, -\beta, +\gamma\}$ for the objects $+\beta$, $-\beta$ and $+\gamma$. Using a weakened disjunction rule we establish that $E \models_s \alpha \vee (\beta \wedge \gamma)$, because $E \models_s \beta \wedge \gamma$ and $E \not\models_s \neg\, (\beta \wedge \gamma)$. According to the disjunction rule we do not need $E \models_s \alpha$ which would not hold. However, $E \not\models_s (\alpha \vee \beta) \wedge (\alpha \vee \gamma)$, because $E \not\models_s \alpha \vee \beta$, which is due to $E \models_s \neg\, \beta$ but $E \not\models_s \alpha$. Together, this contradicts distributivity of disjunction, if we would allow a weakening of the disjunction rule.

Note, the equivalences in strong satisfaction allow for any formula in $\mathcal{L}$ to be rewritten into conjunctive normal form and then into clauses which can be evaluated with respect to the objects in the model.

Weak Satisfaction Relation

Strong satisfaction corresponds to the decompositional rules. Now we need to capture the compositional rules. The definition of weak satisfaction is similar to strong satisfaction. The main difference is that disjunction is less restricted, because it does not incorporate focusing. Indeed, weak satisfaction seems closer to a classical notion of satisfaction.

Definition 4.4.8 (Weak satisfaction)
Let $\models_w$ be a satisfiability relation called weak satisfaction. For a model E, an assignment A, and a literal α in $\mathcal{L}$, we define $\models_w$ as follows.

$(E, A) \models_w \alpha$ iff $(E, A) \models_h \alpha$

For formulae $\alpha, \beta \in \mathcal{L}$, we extend the definition as follows:

$(E, A) \models_w \alpha \vee \beta$ iff $(E, A) \models_w \alpha$ or $(E, A) \models_w \beta$
$(E, A) \models_w \alpha \wedge \beta$ iff $(E, A) \models_w \alpha$ and $(E, A) \models_w \beta$
$(E, A) \models_w \neg\,\neg\, \alpha$ iff $(E, A) \models_w \alpha$
$(E, A) \models_w \neg\, (\alpha \wedge \beta)$ iff $(E, A) \models_w \neg\, \alpha \vee \neg\, \beta$
$(E, A) \models_w \neg\, (\alpha \vee \beta)$ iff $(E, A) \models_w \neg\, \alpha \wedge \neg\, \beta$
$(E, A) \models_w \alpha \Rightarrow \beta$ iff $(E, A) \models_w \neg\, \alpha \vee \beta$

Let B be an x-variant assignment of A, then

$$
\begin{array}{l}
(E,A) \models_w \exists\, x.\alpha(x) \text{ iff for some } B,\ (E,B) \models_w \alpha \\
(E,A) \models_w \forall\, x.\alpha(x) \text{ iff for all } B,\ (E,B) \models_w \alpha \\
(E,A) \models_w \neg\ \exists\, x.\alpha(x) \text{ iff for all } B,\ (E,B) \models_w \neg\ \alpha \\
(E,A) \models_w \neg\ \forall\, x.\alpha(x) \text{ iff for some } B,\ (E,B) \models_w \neg\ \alpha
\end{array}
$$

For a model E we polymorphically extend weak satisfaction as follows

$$E \models_w \varphi \text{ iff for all assignments } A,\ (E,A) \models_w \varphi$$

Such an E is said to be a weak model of φ.

For example, the set of ground formulae $\Delta = \{\neg\ a, a \vee b, a \vee \neg\ b\}$ has the following weak models: $\{-a, +a\}$, $\{-a, +b, -b\}$, $\{-a, +a, +b\}$, $\{-a, +a, -b\}$ and $\{-a, +a, +b, -b\}$. Note, every strong model of a formula φ is a weak model of φ but the converse does not hold in the general case.

Observe that the definition of weak satisfaction differs slightly from the one given by (Hunter, 2000; Hunter, 2001). Disjunction is here applicable for formulae and not only for literals. This change is necessary for the following property proved in (Hunter, 2000). This property justifies Hunter's use of the rules above rather than his more restricted definition.

Lemma 4.4.1
Distributivity is implied by the definition of weak satisfaction, i.e. for any formulae $\alpha, \beta, \gamma \in \mathcal{L}$, and any model E, the following distribution properties hold:

$$
\begin{array}{l}
(E,A) \models_w \alpha \vee (\beta \wedge \gamma) \text{ iff } (E,A) \models_w (\alpha \vee \beta) \wedge (\alpha \vee \gamma) \\
(E,A) \models_w \alpha \wedge (\beta \vee \gamma) \text{ iff } (E,A) \models_w (\alpha \wedge \beta) \vee (\alpha \wedge \gamma)
\end{array}
$$

Proof
Assume $(E,A) \models_w \alpha \vee (\beta \wedge \gamma)$. So $(E,A) \models_w \alpha$ or $((E,A) \models_w \beta$ and $(E,A) \models_w \gamma)$. By distributivity of the classical connectives "or" and "and", we have $((E,A) \models_w \alpha$ or $(E,A) \models_w \beta)$ and $((E,A) \models_w \alpha$ or $(E,A) \models_w \gamma)$. Hence, $(E,A) \models_w (\alpha \vee \beta) \wedge (\alpha \vee \gamma)$. The other case follows similarly. □

Quasi-Classical Entailment

Now we have established all the building blocks for the definition of quasi-classical entailment. Basically, QC entailment is of the same form as classical entailment except that strong satisfaction is used for the assumptions and weak satisfaction is used for the conclusion.

Definition 4.4.9 (QC entailment)
Given a set of formulae $\varphi_1, \ldots, \varphi_n$ and a formula ϕ. Let $\models_Q$ be an entailment relation, called quasi-classical entailment, such that $\models_Q \subseteq \mathbb{P}(\mathcal{L}) \times \mathcal{L}$, and defined as follows:

$$\{\varphi_1, \ldots, \varphi_n\} \models_Q \phi$$
$$\text{iff for all models } E, \text{ if } E \models_s \varphi_1 \text{ and } \ldots \text{ and } E \models_s \varphi_n \text{ then } E \models_w \phi$$

Consider the set of ground formulae $\Delta = \{\neg\, a, a \vee b, a \vee \neg\, b\}$ and recall that its only strong model is $E = \{+a, -a, +b, -b\}$. The model E is also a weak model of a, hence Δ quasi-classically entails a, i.e. $\{\neg\, a, a \vee b, a \vee \neg\, b\} \models_Q a$. Similarly, we can show $\{\neg\, a, a \vee b, a \vee \neg\, b\} \models_Q \neg\, a \vee c$ as well as $\{\neg\, a, a \vee b, a \vee \neg\, b\} \models_Q a \wedge b$. However, we cannot establish $\{\neg\, a, a \vee b, a \vee \neg\, b\} \models_Q d$, because the model E from above is still a strong model of Δ but it is not a weak model of d.

An alternative way of defining entailment corresponds to model set inclusion. The advantage is that we can make use of the standard properties of set inclusion when reasoning about QC entailment.

For a set of formulae $\varphi_1, \ldots, \varphi_n$, its class of strong models Mod_s is defined as the set of all its strong models E, i.e.

$$Mod_s(\varphi_1, \ldots, \varphi_n) = \{E \mid E \models_s \varphi_1 \text{ and } \ldots \text{ and } E \models_s \varphi_n\}$$

and the class of weak models Mod_w for a formula ϕ is the set of its weak models E, i.e.

$$Mod_w(\phi) = \{E \mid E \models_w \phi\}$$

Then QC entailment $\models_Q$ is defined as inclusion of the class of strong models in the class of weak models, i.e.

$$\{\varphi_1, \ldots, \varphi_n\} \models_Q \phi \text{ iff } Mod_s(\varphi_1, \ldots, \varphi_n) \subseteq Mod_w(\phi)$$

4.4.3 The Semantic Tableau Method for First-Order QCL

A proof in QCL is a two stage affair. First, a set of assumptions is transformed into clauses and decomposed into literals. Then, the compositional stage follows, forming clauses from the assumptions and derived literals using disjunction or conjunction introduction, possibly followed by some rewrite steps to form equivalent formulae. Any such obtained formula is a non-trivial inference from the assumptions. We consider the strong satisfaction relation to capture the decomposition of the set of assumptions and weak satisfaction to capture the composition of the conclusion.

The Semantic Tableau Method

The proof theory of first-order quasi-classical logic is based on the notion of semantic tableau. A semantic tableau is a tree-like structure where nodes are labeled with formulae. The idea is that each branch represents the conjunction of the formulae appearing in it and that the tree itself represents the disjunction of its branches. We refer to (Smullyan, 1968) and (Fitting, 1996) who present a thorough overview of the techniques of the semantic tableau method.

The semantic tableau proof procedure is based on refutation, i.e. to prove a formula φ is satisfiable, we begin with $\neg \varphi$ and produce a contradiction. This is done by expanding $\neg \varphi$ such that inessential details of its logical structure are removed until a contradiction appears or no expansion rule can be applied. Such expansion results in a tableau tree. For example, to prove the tautology $q \Rightarrow (p \Rightarrow q)$ we construct the following tree:

$$\begin{array}{c} \neg\,(q \Rightarrow (p \Rightarrow q)) \\ | \\ q, \neg\,(p \Rightarrow q) \\ | \\ q, p, \neg\, q \end{array}$$

and observe the contradiction between the literals q and $\neg\, q$. The tableau is closed and thus the tautology is proven.

However, this approach does not work directly for QCL since the truth and falsehood of a predicate are decoupled. Therefore, the atom q being satisfiable does not mean that $\neg\, q$ is not satisfiable, i.e. it is not possible to construct a contradiction in the same way as above. To overcome this obstacle Hunter introduces signed formulae denoted by *, representing that a formula is unsatisfiable. Then showing q and q^* yields a refutation, as well as $\neg\, q$ and $(\neg\, q)^*$, because a formula cannot be satisfiable and unsatisfiable at the same time.

The idea to use signed formulae is not new. They have often been used in the construction of semantic tableau, for example by (Hähnle et al., 1994). New is that the link between a signed formula and its conjugate has been removed. If this link were put back into the proof theory, QCL would collapse to classical predicate logic.

We formalise the introduced notions. The conjugate of a formula φ is denoted φ^*. Furthermore, the set of signed formulae of $\mathcal{L}$ is denoted $\mathcal{L}^*$ and is defined as $\mathcal{L} \cup \{\varphi^* \mid \varphi \in \mathcal{L}\}$. Given these notions we can define what it means to satisfy the conjugate of a formula.

Definition 4.4.10
For any formula $\varphi \in \mathcal{L}$ we further extend the weak satisfaction and strong satisfaction relations as follows:

$$E \models_s \varphi^* \text{ iff } E \not\models_s \varphi$$
$$E \models_w \varphi^* \text{ iff } E \not\models_w \varphi$$

This means, the formula φ^* is weakly or strongly satisfiable whenever φ is not.

The Tableau S-Rules

In the definition of the quasi-classical (QC) semantic tableau, there are two types of tableau expansion rules, the S-rules and the U-rules. These expansion rules correspond roughly to the two satisfaction relations presented in the last section. First, we introduce the tableau S-rules.

Definition 4.4.11 (S-Rules)
The following are the S-rules for a QC semantic tableau. The $|$ symbol denotes the introduction of a branch point in the QC semantic tableau.

The conjunction S-rule: $\dfrac{\alpha \wedge \beta}{\alpha, \beta}$

The disjunction S-rules:

$$\frac{\alpha_1 \vee \ldots \vee \alpha_n}{(\neg\, \alpha_i)^* \mid \otimes(\alpha_1 \vee \ldots \vee \alpha_n, \alpha_i)} \text{ [where } \alpha_1, \ldots, \alpha_n \text{ are literals]}$$

$$\frac{\alpha_1 \vee \ldots \vee \alpha_n}{\alpha_1 \mid \ldots \mid \alpha_n} \text{ [where } \alpha_1, \ldots, \alpha_n \text{ are literals]}$$

The rewrite S-rules:

$$\frac{\neg\,\neg\,\alpha \vee \gamma}{\alpha \vee \gamma} \qquad \frac{\neg\,(\alpha \wedge \beta) \vee \gamma}{\neg\,\alpha \vee \neg\,\beta \vee \gamma} \qquad \frac{\neg\,(\alpha \vee \beta) \vee \gamma}{(\neg\,\alpha \wedge \neg\,\beta) \vee \gamma}$$

$$\frac{\alpha \vee (\beta \wedge \gamma)}{(\alpha \vee \beta) \wedge (\alpha \vee \gamma)} \qquad \frac{\alpha \wedge (\beta \vee \gamma)}{(\alpha \wedge \beta) \vee (\alpha \wedge \gamma)} \qquad \frac{(\alpha \Rightarrow \beta) \vee \gamma}{(\neg\,\alpha \vee \beta) \vee \gamma}$$

The quantification S-rules:

$$\frac{(\forall\, x.\alpha(x)) \vee \gamma}{\alpha(t) \vee \gamma} \qquad \frac{(\neg\,\exists\, x.\alpha(x)) \vee \gamma}{\neg\,\alpha(t) \vee \gamma}$$

$$\frac{(\exists\, x.\alpha(x)) \vee \gamma}{\alpha(t') \vee \gamma} \qquad \frac{(\neg\,\forall\, x.\alpha(x)) \vee \gamma}{\neg\,\alpha(t') \vee \gamma}$$

where t is a term in $U(\mathcal{L})$ and t' is a term in $U(\mathcal{L})$ but not occurring in the tableau constructed so far.

All the tableau S-rules assume the formula above the line to be satisfiable. Basically, the S-rules correspond to the decompositional rules of a QC proof.

The Tableau U-Rules

The tableau U-rules are a variant of the compositional rules of a QC proof. They correspond roughly to the negation of the weak satisfaction relation. In essence, rather than composing literals to form the conclusion we decompose the conclusion to its literals. As such, all the U-rules assume a formula above the line to be unsatisfiable.

Definition 4.4.12 (U-Rules)
The following are the U-rules for a QC semantic tableau. The | symbol denotes the introduction of a branch point in the QC semantic tableau.

The conjunction U-rule: $\dfrac{(\alpha \wedge \beta)^*}{\alpha^* \mid \beta^*}$

The disjunction U-rule: $\dfrac{(\alpha \vee \beta)^*}{\alpha^*, \beta^*}$

The rewrite U-rules:

$$\frac{(\neg\,\neg\,\alpha)^*}{\alpha^*} \qquad \frac{(\neg\,(\alpha \wedge \beta))^*}{(\neg\,\alpha \vee \neg\,\beta)^*} \qquad \frac{(\neg\,(\alpha \vee \beta))^*}{(\neg\,\alpha \wedge \neg\,\beta)^*} \qquad \frac{(\alpha \Rightarrow \beta)^*}{(\neg\,\alpha \vee \beta)^*}$$

The quantification U-rules:

$$\frac{(\forall\, x.\alpha(x))^*}{(\alpha(t'))^*} \qquad \frac{(\neg\,\exists\, x.\alpha(x))^*}{(\neg\,\alpha(t'))^*} \qquad \frac{(\exists\, x.\alpha(x))^*}{(\alpha(t))^*} \qquad \frac{(\neg\,\forall\, x.\alpha(x))^*}{(\neg\,\alpha(t))^*}$$

where t is a term in $U(\mathcal{L})$ and t' is a term in $U(\mathcal{L})$ but not occurring in the tableau constructed so far.

The QC Semantic Tableau

The S- and U-rules are both decomposition rules for signed formulae. They are applied to construct the semantic tableau for a set of assumptions and a conclusion according to the following definition.

Definition 4.4.13 (Semantic Tableau)
A QC semantic tableau for a set of assumptions Δ and a conclusion φ is a tree such that:

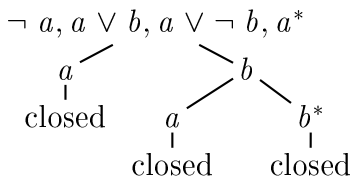
1. the formulae in $\Delta \cup \{\varphi^*\}$ are at the root of the tree;
2. each node of the tree has a set of signed formulae; and

3. the formulae at each node are generated by an application of one of the decomposition rules on a signed formula at ancestors of that node.

This definition is similar to the one for the classical semantic tableau. The major difference is that the root of the classical tableau contains $\Delta \cup \{\neg\, \varphi\}$. The QC tableau does not use this because the link between a formula and its complement has been decoupled.

Definition 4.4.14 (Closed Tableau)
A QC tableau is closed iff every branch is closed. A branch is closed iff there is a formula φ for which φ and φ^* belong to that branch, i.e. both φ and φ^* are on the same path from the root of the tree to the leaf of that branch. A branch is open if there are no more rules that can be applied, and it is not closed. A tableau is open if there is at least one open branch.

For example, to establish $\{\neg\, a, a \vee b, a \vee \neg\, b\} \vdash_Q a$ we take as root the set of formulae $\neg\, a, a \vee b, a \vee \neg\, b, a^*$ and construct the following tableau.



We applied the disjunction S-rule and the disjunction S-rule with focus to construct this tree. Each branch of the tree is closed, hence the tableau is closed and, therefore, $\{\neg\, a, a \vee b, a \vee \neg\, b\} \vdash_Q a$ is valid.

We say $\Delta \vdash_Q \varphi$, i.e. a set of assumptions Δ implies a conclusion φ by QCL, if and only if a QC tableau for Δ and φ is closed. (Hunter, 2001) shows that this QC proof method is sound and complete with respect to the earlier introduced semantics of QCL. We do not prove this statement here, but a generalized version of it after introducing equality into QCL.

Note, all the definitions above are rather similar to the classical form for semantic tableau. In fact, as (Hunter, 2000) points out, the QC semantic tableau collapses to a classical semantic tableau if the following rules are added to the decomposition rules,

$$\frac{\alpha}{(\neg\, \alpha)^*} \qquad \frac{\neg\, \alpha}{\alpha^*} \qquad \frac{\alpha^*}{\neg\, \alpha} \qquad \frac{(\neg\, \alpha)^*}{\alpha}$$

Then we can use the classical definition for closure of a branch, i.e. a branch is closed if it contains both β and $\neg\, \beta$ for some ground atom.

4.4.4 Properties of Quasi-Classical Logic

We now consider some properties of quasi-classical logic. These properties have been presented and proved by (Hunter, 2000) before. However, we recapitulate the arguments to give further insights into QCL. We present the arguments using either QC entailment or QC inference. The decision for one or the other depends on which of the two is more convenient for our purpose.

Paraconsistency

Quasi-classical logic is paraconsistent because it does not allow trivial inferences. That is, given a classical inconsistent set of assumptions Δ, it is not the case that every formula in the language $\mathcal{L}$ is entailed by Δ. For example, let $\alpha, \neg\,\alpha$ and β be ground literals in $\mathcal{L}$. Then it is not the case that $\{\alpha, \neg\,\alpha\} \models_Q \beta$ holds because $E = \{+\alpha, -\alpha\}$ is a possible model such that $E \models_Q \alpha \wedge \neg\,\alpha$ but $E \not\models_Q \beta$.

The only inference rule that allows a new literal, like β, to be introduced is $\vee$-Introduction. QCL is designed such that no decomposition rules can follow $\vee$-Introduction. Therefore, it is not possible to derive the new literal without any context. Hence, QCL does not allow trivial inferences.

Inferences from the Empty Set of Assumptions

In QCL it is not possible to derive any conclusion from the empty set of assumptions, in particular no classical tautologies hold without a given assumption. For example, the tautology $q \Rightarrow (p \Rightarrow q)$ as given in Section 4.4.3 cannot be verified using QCL, i.e. the following tableau is not closed:

$$\begin{array}{c} (q \Rightarrow (p \Rightarrow q))^* \\ | \\ (\neg\, q \vee (p \Rightarrow q))^* \\ | \\ (\neg\, q)^*, (p \Rightarrow q)^* \\ | \\ (\neg\, q)^*, (\neg\, p \vee q)^* \\ | \\ (\neg\, q)^*, (\neg\, p)^*, q^* \end{array}$$

It is not possible to construct a refutation, because an unsatisfiable formula can only be decomposed into unsatisfiable formulae, hence, no contradiction with a satisfiable formula can be derived. Model theoretically this is also easy to see. The empty set is the only strong model satisfying an empty assumption. However, the empty set is not a weak model of any conclusion but the empty one. Therefore, no formula and, in particular, no tautology can be shown from the empty set of assumptions.

It is not clear whether this issue is a drawback for the application of QCL in the context of formal specification, because any practical derivation is likely to be based on a non-empty set of assumptions. Furthermore, when trying to prove a tautology the attempt of performing the proof will indicate a set of necessary assumptions. For example, to close the above tableau, we would need either q, $\neg\, q$ or $\neg\, p$ in the set of assumptions, in particular the classical tautology $q \vee \neg\, q$ is a realistic candidate.

Reflexivity, Monotonicity and Transitivity

Reflexivity, monotonicity and transitivity are often regarded as desired properties of a logic. However, it is well known that there exists a wide range of non-monotonic logics to reason about uncertainty. This indicates that it is possible to give up one or more of these properties if it is practical. Here, we investigate QCL with respect to those three properties.

Quasi-classical logic is reflexive, i.e. for a set of formulae Δ and a formula φ, $\Delta \cup \{\varphi\} \vdash_Q \varphi$ holds. This is easy to see from the root of the corresponding tableau, which is $\Delta, \varphi, \varphi^*$. The tableau is closed immediately, hence the inference holds.

QCL is monotonic, too, i.e. for a set of assumptions Δ and formulae φ and ϕ it holds that $\Delta \vdash_Q \varphi$ implies $\Delta \cup \{\phi\} \vdash_Q \varphi$. This follows simply from set theory, because the set of strong models of $\Delta \cup \{\phi\}$ is included in the set of strong models of Δ which, in turn, are included in the set of weak models of φ, i.e. $Mod_s(\Delta \cup \{\phi\}) \subseteq Mod_s(\Delta) \subseteq Mod_w(\varphi)$. Monotonicity is desired because it allows to add assumptions without retracting conclusions.

The property of transitivity, also called cut, fails in QCL, i.e. for sets of assumptions Δ and Γ and formulae φ and ϕ it holds that $\Delta \cup \{\varphi\} \vdash_Q \phi$ and $\Gamma \vdash_Q \varphi$ does not imply $\Delta \cup \Gamma \vdash_Q \phi$. For example, consider $\{\neg\, \alpha\} \cup \{\alpha \vee \beta\} \vdash_Q \beta$ and $\{\alpha\} \vdash_Q \alpha \vee \beta$, but $\{\alpha, \neg\, \alpha\} \nvdash_Q \beta$.

The failure of transitivity can be regarded as disadvantageous, in particular, with our application in mind. However, (Tennant, 1984) introduced a paraconsistent logic, where transitivity fails, too. In his logic, “transitivity of Proofs fails upon accumulation of Proofs only when the newly combined premises are inconsistent anyway, or the conclusion is a logical truth. In either case, Proofs that show this can be effectively determined from the Proofs given. Thus, transitivity fails where it least matters – arguably, where they ought to fail!” Consequently, we need to investigate the failure of transitivity in QCL with respect to the property of Tennant’s logic. If this holds for QCL, too, then the failure of transitivity may not be a disadvantage anymore.

Consistency Preservation

We discuss the relation of quasi-classical logic to classical logic. First, everything that is derivable in QCL is also derivable in classical logic, i.e. $\Delta \vdash_Q \alpha$ implies $\Delta \vdash \alpha$. For example, $\Delta \vdash_Q \alpha \wedge \neg\, \alpha$ implies $\Delta \vdash \alpha \wedge \neg\, \alpha$. However, the other direction does not hold, i.e. $\Delta \vdash \alpha$ does not imply $\Delta \vdash_Q \alpha$. For example, consider Δ to be empty, then it is possible to show in classical logic $\vdash \alpha \vee \neg\, \alpha$ but this does not hold in QCL.

Even if we restrict considerations to a non-tautological inference of a formula φ that follows classically from a consistent set of formulae, we are not guaranteed that φ also follows in QCL. For example, let $\Delta = \{\alpha\}$, then $\Delta \vdash \beta \Rightarrow (\alpha \wedge \beta)$ is a classical inference but it is not a QC inference. We consider the strong models of α and the weak models of $\beta \Rightarrow (\alpha \wedge \beta)$. One such strong model is $\{+\alpha\}$ but this is not a weak model of the conclusion, hence QC entailment fails.

Further Properties

(Hunter, 2000) presents some more properties, which have been discussed in the context of non-monotonic logics and relevance logics before. It seems interesting to look at these properties to enhance our understanding of QCL. Below, we consider Δ to be a set of formulae and φ, ϕ and ψ are formulae in our language.

And-introduction. The property of and-introduction, i.e. $\Delta \vdash_Q \varphi$ and $\Delta \vdash_Q \phi$ implies $\Delta \vdash_Q \varphi \wedge \phi$, holds in QCL.

Or-elimination. The property of or-elimination, i.e. $\Delta \cup \{\varphi\} \vdash_Q \psi$ and $\Delta \cup \{\phi\} \vdash_Q \psi$ implies $\Delta \cup \{\varphi \vee \phi\} \vdash_Q \psi$, holds in quasi-classical logic.

Furthermore, due to QCL being a weakening of classical logic, some of the laws of classical logic do not hold in QCL. (Hunter, 2000) presents the following classical properties which are not feasible in QCL. Below, we include some counterexamples to give the reader a better understanding of QCL. We consider α, β, and γ to be atomic formulae in our language.

Right modus ponens. The property of right modus ponens, defined as follows, fails in QCL: $\Delta \vdash_Q \varphi$ and $\Delta \vdash_Q \varphi \Rightarrow \phi$ does not imply $\Delta \vdash_Q \phi$. Consider $\Delta = \{\alpha, \neg\, \alpha\}$, then $\Delta \vdash_Q \alpha$, and $\Delta \vdash_Q \alpha \Rightarrow \beta$, but $\Delta \nvdash_Q \beta$.

Deduction Theorem. The property of deduction, defined as follows, fails in QCL: $\Delta \vdash_Q \varphi \Rightarrow \phi$ does not imply $\Delta \cup \{\varphi\} \vdash_Q \phi$. Consider $\Delta = \{\neg\, \alpha\}$, then $\Delta \vdash_Q \alpha \Rightarrow \beta$ but $\Delta \cup \{\alpha\} \nvdash_Q \beta$.

The failure of the deduction theorem has a particular consequence: in formulating properties and theorems the decision whether to use implication or deduction may be crucial. Like other features of QCL, this requires the user of QCL to make its intensions explicit. In general, however, we can consider the main implicative connective of a classical formula as the intended deduction operator.

Conditionalization. The property of conditionalization, defined as follows, fails in QCL: $\Delta \cup \{\varphi\} \vdash_Q \phi$ does not imply $\Delta \vdash_Q \varphi \Rightarrow \phi$. Consider $\Delta = \{\}$. Then $\Delta \cup \{\alpha\} \vdash_Q \alpha$, but $\Delta \nvdash_Q \alpha \Rightarrow \alpha$.

We introduce two further properties. To describe these properties we need to make use of classical predicate logic, because, as mentioned before, QCL does not allow any inferences from the empty set of assumptions.

Right weakening. The property of right weakening, defined as follows, fails in QCL: $\Delta \vdash_Q \varphi$ and $\vdash \varphi \Rightarrow \phi$ does not imply $\Delta \vdash_Q \phi$. Let $\Delta = \{\alpha\}$, then $\{\alpha\} \vdash_Q \alpha$. Furthermore, consider $\vdash \alpha \Rightarrow \beta \vee \neg \beta$. However, $\{\alpha\} \nvdash_Q \beta \vee \neg \beta$.

Left logical equivalence. The property of left logical equivalence, defined as follows, fails in QCL: $\Delta \cup \{\varphi\} \vdash_Q \psi$ and $\vdash \varphi \Leftrightarrow \phi$ does not imply $\Delta \cup \{\phi\} \vdash_Q \psi$. Let $\Delta = \{\}$. $\{\alpha \vee \neg \alpha\} \vdash_Q \alpha \vee \neg \alpha$ and $\vdash (\alpha \vee \neg \alpha) \Leftrightarrow (\beta \vee \neg \beta)$, but $\{\beta \vee \neg \beta\} \nvdash_Q \alpha \vee \neg \alpha$.

4.4.5 Logical Equivalence in Quasi-Classical Logic

Logical equivalences play an important role in simplifying logical formulae. In Chapter 6, for example, we use equivalences to simplify the precondition of an operation given in the Z notation. Despite its importance logical equivalence has not been thoroughly investigated in QCL. For example, the term "equivalent" is used but not defined within QCL. It is referred to classical logic to give it a meaning.

Equivalences and Normal Form

(Hunter, 2000) defines, a formula is in conjunctive normal form (CNF) if and only if it is a conjunction of clauses, i.e. a conjunction of disjuncts of literals.

For example, given the literals α, β and γ then $(\alpha \vee \beta) \wedge \gamma$ is in CNF, whereas $\alpha \vee (\beta \wedge \gamma)$ is not.

It is known that any propositional formula in QCL can be transformed into CNF by application of the following equivalences, in particular distributivity, arrow elimination, double negation elimination and de Morgan laws. We denote this equivalence relation by $\equiv_Q$.

$$
\begin{array}{rclrcl}
\varphi \wedge \varphi & \equiv_Q & \varphi & \varphi \vee \varphi & \equiv_Q & \varphi \\
\varphi \wedge \phi & \equiv_Q & \phi \wedge \varphi & \varphi \vee \phi & \equiv_Q & \phi \vee \varphi \\
\varphi \wedge (\phi \wedge \psi) & \equiv_Q & (\varphi \wedge \phi) \wedge \psi & \varphi \vee (\phi \vee \psi) & \equiv_Q & (\varphi \vee \phi) \vee \psi \\
\neg (\varphi \wedge \phi) & \equiv_Q & \neg \varphi \vee \neg \phi & \neg (\varphi \vee \phi) & \equiv_Q & \neg \varphi \wedge \neg \phi \\
\varphi \vee (\phi \wedge \psi) & \equiv_Q & (\varphi \vee \phi) \wedge (\varphi \vee \psi) & \varphi \wedge (\phi \vee \psi) & \equiv_Q & (\varphi \wedge \phi) \wedge (\varphi \wedge \psi) \\
\neg \neg \varphi & \equiv_Q & \varphi & & & \\
\varphi \Rightarrow \phi & \equiv_Q & \neg \varphi \vee \phi & & & \\
\varphi \Leftrightarrow \phi & \equiv_Q & (\varphi \Rightarrow \phi) \wedge (\phi \Rightarrow \varphi) & & &
\end{array}
$$

(Hunter, 2000) points out that a formula φ is a CNF of a formula ϕ if and only if φ is classically equivalent to ϕ and φ is in CNF. Note, this form of a CNF is often called conjunctive negation normal form (CNNF) because the negation symbol does not apply to formulae but to literals only.

(Hunter, 2001) extends his work to first-order QCL. We follow from his definitions of the strong and weak satisfaction relation that the following two equivalences hold as well.

$$
\begin{array}{rcl}
\neg \forall x.\varphi(x) & \equiv_Q & \exists x.\neg \varphi(x) \\
\neg \exists x.\varphi(x) & \equiv_Q & \forall x.\neg \varphi(x)
\end{array}
$$

Thus, the negation symbol can be pushed inside quantified formulae.

Weak Logical Equivalence

In (Miarka et al., 2002) we defined, two formulae φ and ϕ are equivalent, denoted $\varphi \dashv\vdash_Q \phi$, if and only if $\{\varphi\} \vdash_Q \phi$ and $\{\phi\} \vdash_Q \varphi$. We call this weak equivalence, although this notion is actually not describing an equivalence relation. Consider the following three formulae:

1. $A = \neg \alpha \wedge \alpha \wedge \neg \beta$
2. $B = \neg \alpha \wedge \neg \beta \wedge (\alpha \vee \beta)$
3. $C = \neg \alpha \wedge \neg \beta \wedge \beta$

Then it holds that $A \dashv\vdash_Q B$ and $B \dashv\vdash_Q C$ but $A \not\dashv\vdash_Q C$. This is obvious if we consider the strong and weak model classes of these formulae. Recall, that $\{\varphi\} \vdash_Q \phi$ iff $Mod_s(\varphi) \subseteq Mod_w(\phi)$.

1. $Mod_s(A) = Mod_w(A) = \{\{-\alpha, +\alpha, -\beta\}, \{-\alpha, +\alpha, -\beta, +\beta\}\}$
2. $Mod_s(B) = \{\{-\alpha, +\alpha, -\beta, +\beta\}\}$
   $Mod_w(B) = \{\{-\alpha, +\alpha, -\beta\}, \{-\alpha, -\beta, +\beta\}, \{-\alpha, +\alpha, -\beta, +\beta\}\}$
3. $Mod_s(C) = Mod_w(C) = \{\{-\alpha, -\beta, +\beta\}, \{-\alpha, +\alpha, -\beta, +\beta\}\}$

Now we see that $Mod_s(A) \subseteq Mod_w(B)$ and $Mod_s(B) \subseteq Mod_w(A)$, $Mod_s(B) \subseteq Mod_w(C)$ and $Mod_s(C) \subseteq Mod_w(B)$, but neither $Mod_s(A) \subseteq Mod_w(C)$ nor $Mod_s(C) \subseteq Mod_w(A)$. Hence, the relation $\dashv\vdash_Q$ is not transitive and, therefore, it is not an equivalence relation. Consequently, logical equivalence in QCL cannot be defined in the most straightforward way in terms of the QC consequence relation.

The Absorption Laws

Because the logical equivalence relation in QCL cannot be defined directly in terms of the QC consequence relation, we look at the strong and weak models separately. To gain more understanding we investigate the often applied absorption laws. For the formulae φ and ϕ, the two absorption laws in classical logic are defined as

$$E \models \varphi \vee (\varphi \wedge \phi) \text{ iff } E \models \varphi \quad \text{and} \quad E \models \varphi \wedge (\varphi \vee \phi) \text{ iff } E \models \varphi$$

The absorption laws do not hold for the strong satisfaction relation $\models_s$. Consider the formulae $\alpha \vee (\alpha \wedge \beta)$ and α, then $Mod_s(\alpha) \neq Mod_s(\alpha \vee (\alpha \wedge \beta))$ because $\{+\alpha, -\alpha\} \in Mod_s(\alpha)$ but $\{+\alpha, -\alpha\} \notin Mod_s(\alpha \vee (\alpha \wedge \beta))$. The same applies to the other case.

However, the absorption laws do hold for the weak satisfaction relation. This follows basically from the definition of $\models_w$, in particular from conjunction and disjunction. The proof proceeds by showing the equivalence of the weak model classes, e.g. $Mod_w(\alpha) = Mod_w(\alpha \vee (\alpha \wedge \beta))$, which uses standard set theory.

Equivalence, Weak and Strong Model Classes

To define an appropriate equivalence relation we investigate two equivalence relations based on strong and weak satisfaction. We are interested in finding whether the equivalence relation can be defined in terms of the classical equivalence of the model classes.

For example, we find that if either the strong or the weak models of two formulae are equivalent then they are weakly equivalent. Given two formulae φ and ϕ. If $Mod_s(\varphi) = Mod_s(\phi)$ then $\varphi \dashv\vdash_Q \phi$ and if $Mod_w(\varphi) = Mod_w(\phi)$ then $\varphi \dashv\vdash_Q \phi$.

Proof
($\rightarrow$): $Mod_s(\varphi) = Mod_s(\phi) \subseteq Mod_w(\phi)$, hence $\varphi \vdash_Q \phi$.
($\leftarrow$): $Mod_s(\phi) = Mod_s(\varphi) \subseteq Mod_w(\varphi)$, hence $\phi \vdash_Q \varphi$.
($\rightarrow$): $Mod_s(\varphi) \subseteq Mod_w(\varphi) = Mod_w(\phi)$, hence $\varphi \vdash_Q \phi$.
($\leftarrow$): $Mod_s(\phi) \subseteq Mod_w(\phi) = Mod_w(\varphi)$, hence $\phi \vdash_Q \varphi$.
In either case it follows $\varphi \dashv\vdash_Q \phi$. □

The standard equality relation $=$ is reflexive, symmetric, and transitive. Thus, a notion of equivalence built upon $Mod_s(\varphi) = Mod_s(\phi)$ or $Mod_w(\varphi) = Mod_w(\phi)$ would be an equivalence relation. We find, however, that $Mod_s(\varphi) = Mod_s(\phi)$ does not imply $Mod_w(\varphi) = Mod_w(\phi)$. That $Mod_w(\varphi) = Mod_w(\phi)$ does not imply $Mod_s(\varphi) = Mod_s(\phi)$ has already been established when we investigated the absorption laws.

Proof
Consider the formulae $\varphi = \neg\,\alpha \wedge \neg\,\beta \wedge (\alpha \vee \beta)$ and $\phi = \neg\,\alpha \wedge \neg\,\beta \wedge \alpha \wedge \beta$. Then $Mod_s(\varphi) = \{\{-\alpha, +\alpha, -\beta, +\beta\}\} = Mod_s(\phi) = Mod_w(\phi)$ but $Mod_w(\varphi) = \{\{-\alpha, +\alpha, -\beta\}, \{-\alpha, -\beta, +\beta\}, \{-\alpha, +\alpha, -\beta, +\beta\}\} \neq Mod_w(\phi)$ □

Thus, we cannot define a generally applicable equivalence relation for QCL based only on the strong satisfaction relation.

Strong Logical Equivalence

We have to opt for a stronger definition considering both equivalences over the weak and strong model classes. Thus, if the model classes of two formulae are the same, then these formulae are equivalent, i.e. given two formulae φ and ϕ, then $\varphi \equiv_Q \phi$ iff $Mod_s(\varphi) = Mod_s(\phi)$ and $Mod_w(\varphi) = Mod_w(\phi)$.

Lemma 4.4.2
Strong equivalence in QCL, i.e. $\equiv_Q$, is an equivalence relation.

Proof
To be an equivalence relation, $\equiv_Q$ needs to be reflexive, symmetric and transitive.

- Reflexivity: $\varphi \equiv_Q \varphi$ iff $Mod_s(\varphi) = Mod_s(\varphi)$ and $Mod_w(\varphi) = Mod_w(\varphi)$ by definition of $\equiv_Q$. This holds by reflexivity of equality.
- Symmetry: $\varphi \equiv_Q \phi$ implies $\phi \equiv_Q \varphi$ iff (by definition of $\equiv_Q$) $Mod_s(\varphi) = Mod_s(\phi)$ and $Mod_w(\varphi) = Mod_w(\phi)$ implies $Mod_s(\phi) = Mod_s(\varphi)$ and $Mod_w(\phi) = Mod_w(\varphi)$ This holds by symmetry of equality.
- Transitivity: $\varphi \equiv_Q \phi$ and $\phi \equiv_Q \psi$ implies $\varphi \equiv_Q \psi$ iff $Mod_s(\varphi) = Mod_s(\phi)$ and $Mod_w(\varphi) = Mod_w(\phi)$ and $Mod_s(\phi) = Mod_s(\psi)$ and $Mod_w(\phi) = Mod_w(\psi)$ implies $Mod_s(\varphi) = Mod_s(\psi)$ and $Mod_w(\varphi) = Mod_w(\psi)$. This holds by transitivity of equality.

Hence, the relation $\equiv_Q$ is an equivalence relation. □

Note, this result is compliant with Lemma 5.12 by (Hunter, 2000): For models X and formulae φ and ϕ, if φ is a CNF of ϕ, then the following equivalences hold:

$$X \models_s \varphi \text{ iff } X \models_s \phi,$$
$$X \models_w \varphi \text{ iff } X \models_w \phi$$

Furthermore, it follows from the above investigations that if two formulae φ and ϕ are equivalent, i.e. $\varphi \equiv_Q \phi$ then they are QC consequences of each other, i.e. $\varphi \vdash\!\dashv_Q \phi$. Thus, the relation $\vdash\!\dashv_Q$ is necessary but not sufficient for QC equivalence.

Further Quasi-Classical Equivalences

There are many useful equivalences in classical logic to simplify quantified formulae. For example, the existential quantifier distributes over disjunction in classical logic. We are interested in investigating whether such laws hold in QCL, too.

First, we establish that the universal quantifier distributes over conjunction, i.e.

$$\forall x.(\varphi(x) \wedge \phi(x)) \equiv_Q \forall x.\varphi(x) \wedge \forall x.\phi(x)$$

Proof
To show this, we need to establish that

$$E \models_s \forall x.(\varphi(x) \wedge \phi(x)) \text{ iff } E \models_s \forall x.\varphi(x) \wedge \forall x.\phi(x) \text{ and}$$
$$E \models_w \forall x.(\varphi(x) \wedge \phi(x)) \text{ iff } E \models_w \forall x.\varphi(x) \wedge \forall x.\phi(x)$$

$$E \models_s \forall x.(\varphi(x) \wedge \phi(x))$$
iff {for all assignments A}
$$(E, A) \models_s \forall x.(\varphi(x) \wedge \phi(x))$$
iff {for all x-variant assignments B}
$$(E, B) \models_s \varphi \wedge \phi$$
iff
$$(E, B) \models_s \varphi \text{ and } (E, B) \models_s \phi$$
iff
$$(E, A) \models_s \forall x.\varphi(x) \text{ and } (E, A) \models_s \forall x.\phi(x)$$
iff
$$(E, A) \models_s \forall x.\varphi(x) \wedge \forall x.\phi(x)$$
iff
$$E \models_s \forall x.\varphi(x) \wedge \forall x.\phi(x)$$

The same holds for $\models_w$. □

The next rule is particularly useful when simplifying preconditions in Z. We establish, the existential quantifier distributes over disjunction, i.e.

$$\exists\, x.(\varphi(x) \vee \phi(x)) \equiv_Q \exists\, x.\varphi(x) \vee \exists\, x.\phi(x)$$

Proof
We need to show that

$$E \vDash_s \exists\, x.(\varphi(x) \vee \phi(x)) \text{ iff } E \vDash_s \exists\, x.\varphi(x) \vee \exists\, x.\phi(x) \text{ and}$$
$$E \vDash_w \exists\, x.(\varphi(x) \vee \phi(x)) \text{ iff } E \vDash_w \exists\, x.\varphi(x) \vee \exists\, x.\phi(x)$$

$E \vDash_s \exists\, x.(\varphi(x) \vee \phi(x))$
iff {for all assignments A}
$(E, A) \vDash_s \exists\, x.(\varphi(x) \vee \phi(x))$
iff {for some x-variant assignment C}
$(E, C) \vDash_s \varphi \vee \phi$

$E \vDash_s \exists\, x.\varphi(x) \vee \exists\, x.\phi(x)$
iff {for all assignments A}
$(E, A) \vDash_s \exists\, x.\varphi(x) \vee \exists\, x.\phi(x)$
iff {for some x-variant assignment B}
$(E, B) \vDash_s \varphi \vee \exists\, x.\phi(x)$
iff
$(E, B) \vDash_s \exists\, x.\phi(x) \vee \varphi$
iff {for some x-variant assignment C}
$(E, C) \vDash_s \phi \vee \varphi$
iff
$(E, C) \vDash_s \varphi \vee \phi$

The proof of the weak satisfaction relation is slightly simpler because disjunction is applicable for formulae. □

Other logical equivalences that hold are

$$\begin{array}{rcl}
\exists\, x.(\varphi(x) \wedge \phi) & \equiv_Q & \exists\, x.\varphi(x) \wedge \phi, \text{ provided } x \text{ not in } \phi \\
\forall\, x.(\varphi(x) \vee \phi) & \equiv_Q & \forall\, x.\varphi(x) \vee \phi, \text{ provided } x \text{ not in } \phi \\
\exists\, x.(\varphi(x) \Rightarrow \phi) & \equiv_Q & \forall\, x.\varphi(x) \Rightarrow \phi, \text{ provided } x \text{ not in } \phi \\
\forall\, x.(\varphi \Rightarrow \phi(x)) & \equiv_Q & \varphi \Rightarrow \forall\, x.\phi(x), \text{ provided } x \text{ not in } \varphi \\
\exists\, x.(\varphi \Rightarrow \phi(x)) & \equiv_Q & \varphi \Rightarrow \exists\, x.\phi(x), \text{ provided } x \text{ not in } \varphi
\end{array}$$

4.5 Summary

In this chapter we introduced the notion of paraconsistency as a means to derive non-trivial conclusions from inconsistent information. We presented briefly different ways of weakening classical logic to develop a paraconsistent logic. Then we introduced the paraconsistent logics $FOUR$, $\mathcal{FOUR}$ and QCL, each allowing a slightly different set of conclusions to be derived from inconsistent information.

All paraconsistent logics weaken classical logic in some way. Basically, the application area determines the usefulness of any of the paraconsistent logics, i.e. which weakening least effects the usefulness of the chosen logic. For example, QCL allows too many conclusions for the particular application considered by (da Costa et al., 1995):

John Smith is sick. Dr. Bouvard tells him he has cancer (c). Dr. Pecuchet, however tells him he has not cancer ($\neg\, c$). Both colleagues agree that *If John has got cancer he will die in the next three months* ($c \Rightarrow d$). (da Costa et al., 1995) show that using the logic C_1^+ it is not possible to infer *If John has not got cancer he will not die in the next three months* ($\neg\, c \Rightarrow \neg\, d$). This would be an invalid inference because he could have a car accident. Using QCL, however, it is possible to establish this result:

$$\begin{array}{c} c, \neg\, c, c \Rightarrow d, (\neg\, c \Rightarrow \neg\, d)^* \\ | \\ (c \vee \neg\, d)^* \\ | \\ c^*, (\neg\, d)^* \\ | \\ \text{closed} \end{array}$$

QCL is a relevance logic which is also demonstrated by this example. Because no further information is given about other circumstances that might cause death it is safe to conclude that *If John has not got cancer he will not die in the next three months from cancer.* This example demonstrates the importance of choosing the "right" paraconsistent logic for the envisioned application area.

The four-valued logics provide an intuitive semantics to cope with under- and over-determined information. Thus, we strongly consider their application to handling inconsistency or underdefinedness. Unfortunately, many useful equivalences and derivation rules do not hold in these logics. The former is rather serious for our application in mind because specifiers would need to change their style of writing specifications. The latter influences how specifications are analysed. This might be a smaller problem in comparison to the former. The main application areas of these logics are information systems and logic programming.

We favour Hunter's quasi-classical logic to reason about inconsistent specifications. QCL allows inferences from inconsistent information without resulting in triviality. It has been designed such that all logical connectives behave classically, which enables an easy grasp of the meaning of a formula. It also preserves the derivation rules known from classical logic, however, in QCL the order of application is restricted. The role of resolution in QCL is to decompose clauses into literals to identify those that are involved in an inconsistency. QCL enables the reasoner to distinguish between inconsistent theories, unlike in classical logic.

We not only presented quasi-classical logic but also contributed to its development by discussing the notion of logical equivalence. It turned out that the logical equivalence relation in QCL cannot be defined directly in terms of the QC consequence relation. Thus, we defined a notion of strong logical equivalence for QCL based on strong and weak model classes. We showed that several standard equivalences hold in QCL under strong logical equivalence. We found, however, that the absorption laws known from standard logic do not hold in QCL. In the next chapter we further develop QCL by incorporating a theory of equality between expressions.

Chapter 5

Quasi-Classical Logic with Equality

In the previous chapter we introduced first-order quasi-classical logic to enable useful, non-trivial, reasoning in the presence of inconsistency. Many practical reasoning processes involve the notion of equality. QCL, however, has no explicit way of reasoning about equality. Therefore, we extend the language of QCL by incorporating a theory of equality between expressions in this chapter.

Many relations only make sense when applied to objects of particular types. For example, "taller than" does not apply to colours and "brighter than" not to numbers. The equality relation, however, is universal in the sense that it is meaningful in any domain, like the logical connectives. Thus, the study of equality is generally considered to be part of logic. Therefore, this chapter is of general interest to the studies of QCL.

We have, however, also a more specific reason to study equality in QCL. Our aim is to use QCL to reason about formal specifications written in the Z notation which we briefly introduced in Chapter 2. In Z, equality plays an important role in developing specifications. It is commonly used to relate before- and after-state variables and expressions denoting their values in a specification. Thus, to formally investigate Z specifications using QCL we need to be able to reason about equality.

Based on the notion of equality we can state a useful and often applied rule for reasoning with quantifications. In its most common use it says that if we have an existentially quantified statement, part of which gives a value for the bound variable, then the quantification can be removed and the variable is replaced by its known value wherever it appears. This rule is called the one-point rule and it is often used in the simplification of preconditions in Z. Due to its importance we discuss this rule in the context of QCL towards the end of this chapter.

5.1 Introduction

Equality has often been recognised to be a fundamental logical predicate because it is meaningful no matter what domain of discourse is considered. This distinguishes equality from most other relations that are only applicable in restricted circumstances. For example, the predicate "is red" makes no sense on numbers or the predicate "to the right of" is not meaningful when applied to colours. Equality shares a universality with the logical connectives that makes it generally part of the study of logic.

Equality represents identity, i.e. two things are equal if they denote the same object. For example, "3+3" equals "6" and "the letter occurring in the English alphabet after B" equals "C". Some term t is identical to some other term s, often denoted $t = s$, if we cannot distinguish between them (with respect to all properties). This is known as the Principle of the Indiscernibility of Identicals, or Leibniz's Law. If two things cannot be distinguished then it follows the replacement principle which states that we can replace any occurrence of a term t in a statement by its equal s.

Equality is a two-place relation and it has some basic properties. First, everything is equal to itself, i.e. the equality relation is reflexive. Second, the order of the terms in the equality relation does not matter, i.e. it is symmetric. Third, the property of transitivity: given two things a and b are equal and two things b and c are equal then a and c are equal, too. Finally, if we apply a function to two equal objects then the result will also be equal. All the latter properties can be derived using reflexivity and the replacement principle.

5.1.1 Motivation

Our motivation for studying equality arises from the aim to reason about formal specifications written in the Z notation using QCL. In Z, equalities are commonly used to express the relation between before- and after-states variables and expressions denoting their values. Formal reasoning about Z specifications involves, in particular, reasoning about such equalities. An important consequence of having a notion of equality is the ability to eliminate universal and existential quantification. The latter is know as the one-point rule and it is a frequently used to analyse Z specifications, in particular when simplifying preconditions.

In the context of an inconsistency tolerant logic handling equality could become cumbersome. For example, what does it mean to say that two numbers "1" and "2" are equal, even though we know from mathematics that they are not? How much does such inconsistency influence the reasoning about the given theory? We address these questions at the end of this chapter leading to future work on equality and QCL.

5.1.2 Outline

This chapter is structured as follows. In Section 5.2 we introduce the syntax and semantics for equality, including the equality axioms and some investigation of using these axioms as extra assumptions in the reasoning process using QCL. Section 5.3 provides the basic notations to show that we are dealing in fact with equality. We present extra tableau rules for handling equality in QCL in Section 5.4 and prove their soundness and completeness in Section 5.5. The one-point rule for QCL is discussed in Section 5.6. This chapter concludes with a short discussion and summary in Section 5.7.

5.2 Equality

In this section we present some initial thoughts on equality. This includes the extension of the syntax with a special predicate symbol to denote equality and some initial considerations of the semantics. These are made more concrete by presenting a set of axioms classically required for reasoning about equality. We investigate the effect of these axioms in the context of QCL by considering them as extra assumptions in the set of formulae given as the premise of a QC derivation.

5.2.1 Syntax and Semantics

The syntax of quasi-classical logic with equality is the same as that of QCL but with the addition of the designated two-place relation symbol $\approx$ for denoting the equality relation. Note, we do not use the symbol $=$ to avoid confusion between object language and meta-language. Generally, we use the $\approx$ symbol in infix notation, following the standard convention. For example, given two terms t and u we write $t \approx u$ instead of $\approx (t, u)$.

Giving the extra symbol $\approx$ does not yet enable us to reason about equality. For example, given two constant symbols a and b and a predicate symbol P, then the following consequence $\{a \approx b, P(a)\} \models_Q P(b)$ cannot be directly established in QCL. First, we need to ensure that the symbol $\approx$ really denotes equality. We introduced the notion of a quasi-classical model. Now we are interested in those models only in which the $\approx$ symbol is treated as the equality relation.

Definition 5.2.1 (Normal model)
A model E is called normal provided the relation symbol $\approx$ is interpreted as the equality relation over the domain of E.

The aim is to find a consequence relation $\models_{Q_\approx}$ where $\Delta \models_{Q_\approx} \varphi$ is like $\Delta \models_Q \varphi$, except it takes equality into account, i.e. normal models. This implies, that if

$\Delta \vDash_Q \varphi$ then $\Delta \vDash_{Q\approx} \varphi$. The converse, however, is not true. For example, let $\Delta = \{a \approx b, P(a)\}$, then we want $\Delta \vDash_{Q\approx} P(b)$, but not $\Delta \vDash_Q P(b)$.

5.2.2 Equality Axioms

One of the features of QCL is that assumptions contributing to the reasoning process need to be made explicit. For example, $\{a \approx b, P(a)\} \vDash_Q P(b)$ fails because an important assumption is missing. If we add the predicate $\forall\, x, y.(x \approx y \Rightarrow (P(x) \Rightarrow P(y)))$ to the set of assumptions then we can infer $P(b)$ using QCL. The set of assumptions we need to reason about equality are called the equality axioms.

The basic equality axioms are reflexivity and replacement. Given those, we are able to show that equality is an equivalence relation, i.e. it is reflexive, symmetric and transitive. Basically, we follow in our presentation (Fitting, 1996, p. 276 ff).

Definition 5.2.2 (Reflexivity)
ref is the sentence $\forall\, x.x \approx x$.

The sentence *ref* captures the reflexivity property of equality. Next, we define the replacement property. Note, we define two sets of replacement axioms, one for function symbols and one for predicate symbols.

Definition 5.2.3 (Function replacement axiom)
Let f be an n-place function symbol. The following sentence is a replacement axiom for f: $\forall\, v_1 \dots v_n \forall\, w_1 \dots w_n.(v_1 \approx w_1 \wedge \dots \wedge v_n \approx w_n) \Rightarrow f(v_1, \dots, v_n) \approx f(w_1, \dots, w_n)$.

For example, if $*$ is a two-place function symbol of the language then $\forall\, w, x, y, z.(x \approx z \wedge y \approx w) \Rightarrow (x * y \approx z * w)$ is a particular function replacement axiom, say A. Assuming c is a constant symbol, we show $\{A, \mathit{ref}\} \vDash_Q \forall\, x, z.(x \approx z) \Rightarrow (x * c \approx z * c)$:

$$
\begin{array}{c}
\forall\, w, x, y, z.((x \approx z \wedge y \approx w) \Rightarrow x * y \approx z * w),\\
\forall\, x.x \approx x,\\
(\forall\, x, z.(x \approx z \Rightarrow x * c \approx z * c))^*\\
|\\
(a \approx b \Rightarrow a * c \approx b * c)^*\\
|\\
(\neg\ (a \approx b))^*, (a * c \approx b * c)^*\\
|\\
(a \approx b \wedge c \approx c) \Rightarrow a * c \approx b * c,\ c \approx c\\
|\\
\neg\ (a \approx b) \vee \neg\ (c \approx c) \vee a * c \approx b * c\\
\begin{array}{cc}
\begin{array}{c}
\neg\ (a \approx b) \vee \neg\ (c \approx c)\\
\begin{array}{cc}
\neg\ (a \approx b) & (c \approx c)^*\\
| & |\\
\text{closed} & \text{closed}
\end{array}
\end{array}
&
\begin{array}{c}
a * c \approx b * c\\
|\\
\text{closed}\\
\\
\end{array}
\end{array}
\end{array}
$$

In a first-order language we are not able to quantify over function symbols nor predicate symbols. Thus, we do it indirectly by defining the set of all function replacement axioms.

Definition 5.2.4
For a language $\mathcal{L}$, $fun(\mathcal{L})$ is the set of replacement axioms for all function symbols of $\mathcal{L}$. Members of $fun(\mathcal{L})$ are called function replacement axioms.

Note, there is one function replacement axiom for each function symbol of the language. Therefore, if the language has infinitely many function symbols, the set of function replacement axioms is also infinite. We defined replacement only for the simplest kind of terms but replacement for more complicated terms follows. For example,

$$
\begin{array}{c}
\forall\, x, y.(x \approx y \Rightarrow f(x) \approx f(y)),\\
\forall\, x, y.(x \approx y \Rightarrow g(x) \approx g(y)),\\
(\forall\, x, y.(x \approx y \Rightarrow f(g(x)) \approx f(g(y))))^*\\
|\\
(a \approx b \Rightarrow f(g(a)) \approx f(g(b)))^*\\
|\\
(\neg\ (a \approx b))^*, (f(g(a)) \approx f(g(b)))^*\\
|\\
g(a) \approx g(b) \Rightarrow f(g(a)) \approx f(g(b))\\
|\\
\neg\ (g(a) \approx g(b)) \vee f(g(a)) \approx f(g(b))\\
\begin{array}{cc}
\begin{array}{c}
(g(a) \approx g(b))^*\\
|\\
\forall\, x, y.(x \approx y \Rightarrow g(x) \approx g(y))\\
|\\
a \approx b \Rightarrow g(a) \approx g(b)\\
\begin{array}{cc}
\neg\ (a \approx b) & g(a) \approx g(b)\\
| & |\\
\text{closed} & \text{closed}
\end{array}
\end{array}
&
\begin{array}{c}
f(g(a)) \approx f(g(b))\\
|\\
\text{closed}\\
\\
\\
\\
\\
\end{array}
\end{array}
\end{array}
$$

After we have considered function symbols, we turn to the replacement property of relation symbols.

Definition 5.2.5 (Relation replacement axiom)
Let R be an n-place relation symbol. The following sentence is a replacement axiom for R: $\forall\, v_1 \ldots v_n \,\forall\, w_1 \ldots w_n.((v_1 \approx w_1 \wedge \ldots \wedge v_n \approx w_n) \Rightarrow (R(v_1, \ldots, v_n) \Rightarrow R(w_1, \ldots, w_n)))$.

We defined $\approx$ to be a two-place relation symbol. Its replacement axiom is $\forall\, v_1, v_2, w_1, w_2.((v_1 \approx w_1 \wedge v_2 \approx w_2) \Rightarrow (v_1 \approx v_2 \Rightarrow w_1 \approx w_2))$ which we denote by B for now. It follows the symmetry property for $\approx$, i.e. $\{B, \mathit{ref}\} \vDash_Q \forall\, x, y.(x \approx y \Rightarrow y \approx x)$ We can also show that transitivity is a consequence of B and ref, i.e. $\{B, \mathit{ref}\} \vDash_Q \forall\, x, y, z.((x \approx y \wedge y \approx z) \Rightarrow x \approx z)$.

Again, because we cannot quantify over the relation symbols in a first-order language we collect all relation replacement axioms in an appropriate set.

Definition 5.2.6
For a language $\mathcal{L}$, $\mathit{rel}(\mathcal{L})$ is the set of replacement axioms for all relation symbols of $\mathcal{L}$. Members of $\mathit{rel}(\mathcal{L})$ are called relation replacement axioms.

Reflexivity and the replacement axioms form together the set of all the equality axioms.

Definition 5.2.7 (Equality axioms)
For a language $\mathcal{L}$, by $\mathit{eq}(\mathcal{L})$ we mean the set $\{\mathit{ref}\} \cup \mathit{fun}(\mathcal{L}) \cup \mathit{rel}(\mathcal{L})$. Members of this set are called equality axioms for $\mathcal{L}$.

In standard first-order predicate logic the equality axioms are exactly what is required to reduce the problems about logic with equality to more general questions about first-order logic. This relation is expressed by the following theorem:

Let $\mathcal{L}$ be a first-order language and Δ a set of sentences over $\mathcal{L}$. Then

$$\Delta \vDash_{\approx} \varphi \text{ if and only if } \Delta \cup \mathit{eq}(\mathcal{L}) \vDash \varphi$$

where $\vDash_{\approx}$ is the classical consequence relation that takes equality into account, i.e. $X \vDash_{\approx} S$ provided X holds in every normal model in which S holds. The question that arises is whether this also carries over to quasi-classical logic, i.e. whether we can establish:

$$\Delta \vDash_{Q\approx} \varphi \text{ if and only if } \Delta \cup \mathit{eq}(\mathcal{L}) \vDash_Q \varphi$$

5.2.3 Equality and Strong Satisfiability

We developed a set of equality axioms to support reasoning about equality. To gain some more insight into reasoning with equality we investigate the effect of adding these axioms to the set of assumptions. QCL is monotonic, thus adding these axioms to the set of assumptions would not affect previous inferences.

Recall the definition of QC consequence: given a set of assumptions Δ and a formula φ, then φ is a consequence of Δ, denoted $\Delta \vDash_Q \varphi$, if and only if for all models E, if E strongly satisfies every formula in Δ then E must weakly satisfy φ. Now we add the equality axioms, i.e. we are interested in $\Delta \cup eq(\mathcal{L}) \vDash_Q \varphi$. According to the definition of QC consequence the model E must now strongly satisfy the equality axioms. Thus, for any function symbol f and relation symbol α we have

$$
\begin{aligned}
& E \vDash_s eq(\mathcal{L}) \\
\equiv\ & \{\text{Definition of the Equality Axioms, Consider any assignment } A\} \\
& (E, A) \vDash_s \forall\, x.(x \approx x) \text{ and} \\
& (E, A) \vDash_s \forall\, x, y.(x \approx y \Rightarrow f(x) \approx f(y)) \text{ and} \\
& (E, A) \vDash_s \forall\, x, y.(x \approx y \Rightarrow (\alpha(x) \Rightarrow \alpha(y))) \\
\equiv\ & \{\text{Quantification and Implication, } B \text{ is } x\text{- and } y\text{-variant of } A\} \\
& (E, B) \vDash_s s \approx s \text{ and} \\
& (E, B) \vDash_s \neg\,(s \approx t) \vee f(s) \approx f(t) \text{ and} \\
& (E, B) \vDash_s \neg\,(s \approx t) \vee \neg\,\alpha(s) \vee \alpha(t)
\end{aligned}
$$

Using the definition of strong satisfiability for disjunction and conjunction we break these three conditions further down. Then, because $\approx$ is an atomic relation, we can move on to set membership of relations in the model. Using several laws of formal logic we derive

$$
\begin{aligned}
& +s \approx s \in E \\
\text{and} \\
& -s \approx t \in E \text{ or } +f(s) \approx f(t) \in E \text{ and} \\
& \quad \text{if } +s \approx t \in E \text{ then } +f(s) \approx f(t) \in E \text{ and} \\
& \quad \text{if } -f(s) \approx f(t) \in E \text{ then } -s \approx t \in E \\
\text{and} \\
& -s \approx t \in E \text{ or } -\alpha(s) \in E \text{ or } +\alpha(t) \in E \text{ and} \\
& \quad \text{if } +s \approx t \in E \text{ then} \\
& \qquad -\alpha(s) \in E \text{ or } +\alpha(t) \in E \text{ and} \\
& \qquad\quad \text{if } -\alpha(t) \in E \text{ then } -\alpha(s) \in E \text{ and}
\end{aligned}
$$

$$
\begin{array}{l}
\text{if } +\alpha(s) \in E \text{ then} \\
\quad -s \approx t \text{ or } +\alpha(t) \in E \text{ and} \\
\qquad \text{if } +s \approx t \in E \text{ then } +\alpha(t) \in E \text{ and} \\
\text{if } -\alpha(t) \in E \text{ then} \\
\quad -s \approx t \in E \text{ or } -\alpha(s) \in E \text{ and} \\
\qquad \text{if } +\alpha(s) \in E \text{ then } -s \approx t \in E
\end{array}
$$

The equality axioms restrict the set of possible models to those that fulfill the above conditions. For example, each model E must contain the reflexivity axiom for every term s. Consider the atomic formula $\alpha(c)$, where c is some constant, then the class of all strong models satisfying $\alpha(c)$ and the equality axioms is $Mod_s(eq(\mathcal{L}) \cup \{\alpha(c)\}) = \{\{+c \approx c, +\alpha(c)\}, \{+c \approx c, +\alpha(c), -c \approx c\}, \{+c \approx c, +\alpha(c), -\alpha(c)\}, \{+c \approx c, +\alpha(c), -c \approx c, -\alpha(c)\}\}$

Furthermore, conditions like if $+s \approx t \in E$ then $+f(s) \approx f(t) \in E$ are similar to those used by (Fitting, 1996, p. 280f) to construct the first-order Hintikka sets with equality. Note, the conditions for handling inequality are made explicit. This was expected because a formula is decoupled from its negation in QCL and thus equality should be decoupled from inequality. These derived conditions guide the further development of our theory of equality for QCL.

5.3 Equality and Normal Models

Quasi-classical logic has two satisfiability relations, called strong and weak satisfaction. To add equality to QCL we restrict both satisfiability relations. We show that these restrictions are sufficient such that any model satisfying a formula strongly or weakly is a normal model.

Definition 5.3.1 (Strong satisfaction with equality)
Given definition 4.4.7 of the strong satisfaction relation. For any literal α, terms s and t and function symbol f we require the following properties to hold for every pair (E, A):

$$
\begin{array}{l}
(E, A) \models_s t \approx t \\
(E, A) \models_s s \approx t \text{ and } (E, A) \models_s \alpha(s) \text{ implies } (E, A) \models_s \alpha(t) \\
(E, A) \models_s \neg\, (f(s) \approx f(t)) \text{ implies } (E, A) \models_s \neg\, (s \approx t) \\
(E, A) \models_s \alpha(s) \text{ and } (E, A) \models_s \neg\, \alpha(t) \text{ implies } (E, A) \models_s \neg\, (s \approx t)
\end{array}
$$

Similar, we extend the notion of weak satisfaction to handle equality.

Definition 5.3.2 (Weak satisfaction with equality)
Given definition 4.4.8 of the weak satisfaction relation. For any literal α, terms s and t and function symbol f we require the following properties to hold for every pair (E, A):

$$(E, A) \vDash_w t \approx t$$
$$(E, A) \vDash_w s \approx t \text{ and } (E, A) \vDash_w \alpha(s) \text{ implies } (E, A) \vDash_s \alpha(t)$$
$$(E, A) \vDash_w \neg (f(s) \approx f(t)) \text{ implies } (E, A) \vDash_w \neg (s \approx t)$$
$$(E, A) \vDash_w \alpha(s) \text{ and } (E, A) \vDash_w \neg \alpha(t) \text{ implies } (E, A) \vDash_w \neg (s \approx t)$$

We have to convince ourselves that these conditions are sufficient, i.e. we need to show that they select only models that are normal. Since the definitions above use only literals we can unfold them to consider the models directly. Then we have

Definition 5.3.3 ($\approx$-closed)
Any model E which satisfies the following conditions is said to be $\approx$-closed.

1. for any term t in $\mathcal{L}$, $+t \approx t \in E$
2. if $+s_1 \approx t_1 \in E$ and $\ldots$ and $+s_n \approx t_n \in E$ and $+\alpha(s_1, \ldots, s_n) \in E$ then $+\alpha(t_1, \ldots, t_n) \in E$
3. if $+s_1 \approx t_1 \in E$ and $\ldots$ and $+s_n \approx t_n \in E$ and $-\alpha(s_1, \ldots, s_n) \in E$ then $-\alpha(t_1, \ldots, t_n) \in E$
4. if $-f(s_1, \ldots, s_n) \approx f(t_1, \ldots, t_n) \in E$ then $-s_1 \approx t_1 \in E$ or $\ldots$ or $-s_n \approx t_n \in E$
5. if $+\alpha(s_1, \ldots, s_n) \in E$ and $-\alpha(t_1, \ldots, t_n) \in E$ then $-s_1 \approx t_1 \in E$ or $\ldots$ or $-s_n \approx t_n \in E$

for any literal α, function symbols f and terms $s_1, \ldots, s_n$ and $t_1, \ldots, t_n$ in $\mathcal{L}$.

The literal α can be the two-place $\approx$-relation as well. In that case the second condition, for example, instantiates to:
if $+s_1 \approx t_1 \in E$ and $+s_2 \approx t_2 \in E$ and $+\approx (s_1, s_2) \in E$ then $+ \approx (t_1, t_2) \in E$.
Note the ways of writing the $\approx$-relation symbol in infix and prefix notation to indicate the different intention in usage.

Lemma 5.3.1
The relation $\approx$ is an equivalence relation in an $\approx$-closed model.

Proof
We show that $\approx$ is reflexive, transitive, and symmetric, i.e. for every $\approx$-closed model E it holds

$$+t \approx t \in E$$
$$+s \approx t \in E \text{ iff } +t \approx s \in E$$
$$+s \approx t \in E \text{ and } +t \approx u \in E \text{ implies } +s \approx u \in E$$

Reflexivity: Holds by definition.

Symmetry: We have $+s \approx s \in E$, i.e. $+\approx (s, s) \in E$, and by assumption of symmetry $+s \approx t \in E$. Thus we have $+s \approx t \in E$ and $+\approx (s, s) \in E$ and therefore by definition it follows $+\approx (t, s) \in E$, i.e. $+t \approx s \in E$. The other direction is similar.

Transitivity: By assumption of transitivity we have $+s \approx t \in E$ and $+t \approx u \in E$, i.e. we have $+s \approx t \in E$ and $+\approx (t, u) \in E$. Then by symmetry and definition it follows $+\approx (s, u) \in E$, i.e. $+s \approx u \in E$. □

The given replacement condition in the definition is sufficient to reason about equality and function symbols as well, i.e. it holds the following congruence for any model E, terms $s_1, \ldots, s_n$ and $t_1, \ldots, t_n$ and function symbol f in $\mathcal{L}$:

$$\begin{array}{ll} \text{if} & +s_1 \approx t_1 \in E \text{ and } \ldots \text{ and } +s_n \approx t_n \in E \\ \text{then} & +f(s_1, \ldots, s_n) \approx f(t_1, \ldots, t_n) \in E \end{array}$$

We have $+f(s_1, \ldots, s_n) \approx f(s_1, \ldots, s_n) \in E$ by reflexivity, i.e. in prefix notation that is $+\approx (f(s_1, \ldots, s_n), f(s_1, \ldots, s_n)) \in E$ and by assumption we have $+s_1 \approx t_1 \in E$ and $\ldots$ and $+s_n \approx t_n \in E$. Thus it follows by definition $+\approx (f(s_1, \ldots, s_n), f(t_1, \ldots, t_n)) \in E$, which is $+f(s_1, \ldots, s_n) \approx f(t_1, \ldots, t_n) \in E$ in infix notation.

Because $\approx$ is an equivalence relation on the domain $U(\mathcal{L})$ of the model E that is $\approx$-closed, it partitions its domain into disjoint equivalence classes. We denote the equivalence class containing the closed term t with $\langle\langle t\rangle\rangle$. Formally,

$$\langle\langle t\rangle\rangle = \{u \in U(\mathcal{L}) \mid +t \approx u \in E\}$$

Lemma 5.3.2
For terms t and u and a model E, $\langle\langle t\rangle\rangle = \langle\langle u\rangle\rangle$ if and only if $+t \approx u \in E$.

Proof

($\rightarrow$) $+t \approx t \in E$ by reflexivity, thus $t \in \langle\langle t\rangle\rangle$; by assumption $\langle\langle t\rangle\rangle = \langle\langle u\rangle\rangle$ it follows that $t \in \langle\langle u\rangle\rangle$; thus $+u \approx t \in E$ and by symmetry $+t \approx u \in E$.

($\leftarrow$) Let $v \in \langle\langle t\rangle\rangle$ then $+t \approx v \in E$ and by symmetry $+v \approx t \in E$; it follows by assumption $+t \approx u \in E$ and transitivity that $+v \approx u \in E$ and by symmetry $+u \approx v \in E$, i.e. $v \in \langle\langle u\rangle\rangle$; thus $\langle\langle t\rangle\rangle \subseteq \langle\langle u\rangle\rangle$. It follows similarly that $\langle\langle u\rangle\rangle \subseteq \langle\langle t\rangle\rangle$; hence $\langle\langle t\rangle\rangle = \langle\langle u\rangle\rangle$. □

Let $U'(\mathcal{L})$ be the set of all equivalence classes over $\approx$, i.e.

$$U'(\mathcal{L}) = \{\langle\langle u\rangle\rangle \mid u \in U(\mathcal{L})\}$$

We take $U'(\mathcal{L})$ to be the domain of a new model E'. Next, we define a new interpretation for the model E' by relating the new interpretation $[\![.]\!]$ to the already established interpretation $[.]$. First, we consider constant and function symbols.

Definition 5.3.4
Ground terms in $\mathcal{L}$ are interpreted as follows, where $[\![.]\!]$ is the new interpretation relation.

> $[\![c]\!] = \langle\langle c \rangle\rangle$, for any constant symbol c.
>
> $[\![f]\!](\langle\langle t_1 \rangle\rangle, ..., \langle\langle t_n \rangle\rangle) = \langle\langle f(t_1, \ldots, t_n) \rangle\rangle$ for some function symbol f and terms $t_1, \ldots, t_n$.

Recall that for any interpretation I it holds $(f(t_1, \ldots, t_n))^I = f^I((t_1)^I, ..., (t_n)^I)$ and, in particular, $[\![f(t_1, \ldots, t_n)]\!] = [\![f]\!]([\![t_1]\!], ..., [\![t_n]\!])$.

We need to check whether these definitions are well-chosen because the behaviour of $[\![f]\!]$ on the class $\langle\langle t_i \rangle\rangle$ of ground terms depends on t_i, a member of the class. We show: For ground terms $t_1, \ldots, t_n$ and $u_1, \ldots, u_n$, if $\langle\langle t_1 \rangle\rangle = \langle\langle u_1 \rangle\rangle$ and ... and $\langle\langle t_n \rangle\rangle = \langle\langle u_n \rangle\rangle$ then $\langle\langle f(t_1, \ldots, t_n) \rangle\rangle = \langle\langle f(u_1, \ldots, u_n) \rangle\rangle$. This follows because $\langle\langle f(t_1, \ldots, t_n) \rangle\rangle = [\![f]\!](\langle\langle t_1 \rangle\rangle, ..., \langle\langle t_n \rangle\rangle)$; using the assumptions we get $[\![f]\!](\langle\langle u_1 \rangle\rangle, ..., \langle\langle u_n \rangle\rangle)$ which is equal to $\langle\langle f(u_1, \ldots, u_n) \rangle\rangle$.

Lemma 5.3.3
For a closed term t of $\mathcal{L}$ it holds that $[\![t]\!] = \langle\langle [t] \rangle\rangle$.

Proof
We use induction over the structure of t to show this.

Base case.
Consider the term t is a constant, i.e. $t = c$: $[\![t]\!] = [\![c]\!] = \langle\langle c \rangle\rangle = \langle\langle [c] \rangle\rangle = \langle\langle [t] \rangle\rangle$

Induction step. Assume it holds for ground terms $t_1, \ldots, t_n$. We show it also holds for terms $t = f(t_1, \ldots, t_n)$: $[\![t]\!] = [\![f(t_1, \ldots, t_n)]\!] = [\![f]\!]([\![t_1]\!], \ldots, [\![t_n]\!]) = [\![f]\!](\langle\langle t_1 \rangle\rangle, \ldots, \langle\langle t_n \rangle\rangle) = \langle\langle f(t_1, \ldots, t_n) \rangle\rangle = \langle\langle t \rangle\rangle$ □

This implies that the model we construct is canonical, i.e. that the member $\langle\langle t \rangle\rangle$ of the domain $U'(\mathcal{L})$ will have the closed term t as a name.

Next, we consider relation symbols. We define

Definition 5.3.5
For a relation symbol α and terms $t_1, \ldots, t_n$ it holds

$$+\alpha(\langle\langle t_1 \rangle\rangle, ..., \langle\langle t_n \rangle\rangle) \in E' \text{ iff } +\alpha(t_1, \ldots, t_n) \in E$$
$$-\alpha(\langle\langle t_1 \rangle\rangle, ..., \langle\langle t_n \rangle\rangle) \in E' \text{ iff } -\alpha(t_1, \ldots, t_n) \in E$$

In particular, it holds $+\langle\langle t_1\rangle\rangle \approx \langle\langle t_2\rangle\rangle \in E'$ iff $+t_1 \approx t_2 \in E$. Thus, the model E' we construct is normal because $+t_1 \approx t_2 \in E$ iff $\langle\langle t_1\rangle\rangle = \langle\langle t_2\rangle\rangle$, i.e. the symbol $\approx$ is interpreted as equality.

Again, we need to demonstrate that the definition is well-chosen because the satisfaction of a relation α over equivalence classes depends on its satisfaction over particular members. Thus, for ground terms $t_1, \ldots, t_n$ and $u_1, \ldots, u_n$, if $\langle\langle t_1\rangle\rangle = \langle\langle u_1\rangle\rangle$ and ... and $\langle\langle t_n\rangle\rangle = \langle\langle u_n\rangle\rangle$ then $+\alpha(t_1, \ldots, t_n) \in E$ iff $+\alpha(u_1, \ldots, u_n) \in E$. This holds because $+\alpha(t_1, \ldots, t_n) \in E$ iff $+\alpha(\langle\langle t_1\rangle\rangle, ..., \langle\langle t_n\rangle\rangle) \in E'$ by definition; using the assumptions it follows $+\alpha(\langle\langle u_1\rangle\rangle, ..., \langle\langle u_n\rangle\rangle) \in E'$ and by definition $+\alpha(u_1, \ldots, u_n) \in E$. A similar property can be established for negative objects, too.

Given is $A : Var \to U(\mathcal{L})$, the assignment in a model E. We introduce $A' : Var \to U'(\mathcal{L})$ the assignment in E' such that for a variable x it holds $x^{A'} = \langle\langle x^A\rangle\rangle$. Then it follows

Lemma 5.3.4
For a term t of $\mathcal{L}$, not necessarily closed, it holds that $[\![t]\!]^{A'} = \langle\langle [t]^A\rangle\rangle$.

Proof
We use induction over the structure of t to show this.

Base cases. Consider the term t is a constant, i.e. $t = c$: $[\![t]\!]^{A'} = [\![c]\!]^{A'} = [\![c]\!] = \langle\langle c\rangle\rangle = \langle\langle [c]^A\rangle\rangle = \langle\langle [t]^A\rangle\rangle$, or a variable, i.e. $t = x$: $[\![t]\!]^{A'} = [\![x]\!]^{A'} = x^{A'} = \langle\langle x^A\rangle\rangle = \langle\langle [t]^A\rangle\rangle$

Induction step. Assume it holds for terms $t_1, \ldots, t_n$. We show it also holds for terms $t = f(t_1, \ldots, t_n)$: $[\![t]\!]^{A'} = [\![f(t_1, \ldots, t_n)]\!]^{A'} = [\![f]\!]([\![t_1]\!]^{A'}, \ldots, [\![t_n]\!]^{A'}) = [\![f]\!](\langle\langle [t_1]^A\rangle\rangle, \ldots, \langle\langle [t_n]^A\rangle\rangle) = \langle\langle f([t_1]^A, \ldots, [t_n]^A)\rangle\rangle = \langle\langle [f(t_1, \ldots, t_n)]^A\rangle\rangle = \langle\langle [t]^A\rangle\rangle$
□

Finally, we need to define the variants of the weak and strong satisfaction relations. Basically, they are similar to the standard definitions. The major difference occurs in the atomic case:

$$
\begin{array}{lll}
(E', A') \vDash_s \alpha(t_1, \ldots, t_n) & \text{iff} & +\alpha([\![t_1]\!]^{A'}, \ldots, [\![t_n]\!]^{A'}) \in E' \\
(E', A') \vDash_s \neg\, \alpha(t_1, \ldots, t_n) & \text{iff} & -\alpha([\![t_1]\!]^{A'}, \ldots, [\![t_n]\!]^{A'}) \in E' \\
(E', A') \vDash_w \alpha(t_1, \ldots, t_n) & \text{iff} & +\alpha([\![t_1]\!]^{A'}, \ldots, [\![t_n]\!]^{A'}) \in E' \\
(E', A') \vDash_w \neg\, \alpha(t_1, \ldots, t_n) & \text{iff} & -\alpha([\![t_1]\!]^{A'}, \ldots, [\![t_n]\!]^{A'}) \in E'
\end{array}
$$

Lemma 5.3.5
For any formula φ and any assignment A in an $\approx$-closed model E it holds

$$
\begin{array}{l}
(E, A) \vDash_s \varphi \text{ iff } (E', A') \vDash_s \varphi \\
(E, A) \vDash_w \varphi \text{ iff } (E', A') \vDash_w \varphi
\end{array}
$$

i.e. it holds for every formula that it is satisfiable in a model E if and only if it is also satisfiable in a normal model E'.

Proof
We use induction over the structure of φ to show this.

- Base cases.
  (a) Let $\varphi = \alpha(t_1, \ldots, t_n)$, $t_1, \ldots, t_n$ terms. Then $(E', A') \models_s \alpha(t_1, \ldots, t_n)$ iff $+\alpha([\![t_1]\!]^{A'}, \ldots, [\![t_n]\!]^{A'}) \in E'$ iff $+\alpha(\langle\langle [t_1]^A \rangle\rangle, \ldots, \langle\langle [t_n]^A \rangle\rangle) \in E'$ iff $+\alpha([t_1]^A, \ldots, [t_n]^A) \in E$ iff $(E, A) \models_s \alpha(t_1, \ldots, t_n)$. The case for the weak satisfaction relation follows similarly.
  (b) Let $\varphi = \neg\, \alpha(t_1, \ldots, t_n)$. Then it follows similarly as for $\varphi = \alpha(t_1, \ldots, t_n)$ but using negative objects $-\alpha$ instead of positive objects $+\alpha$.

- Induction step.

  Suppose it holds for formulae ϕ, ψ and ρ. We show that it also holds for more complicated formulae.

  The propositional cases are straightforward.

  ($\wedge$) For example: $(E', A') \models_s \phi \wedge \psi$ iff $(E', A') \models_s \phi$ and $(E', A') \models_s \psi$, it follows by the induction hypothesis $(E, A) \models_s \phi$ and $(E, A) \models_s \psi$ iff $(E, A) \models_s \phi \wedge \psi$. The case for weak satisfaction follows similarly.

  ($\vee_w$) The disjunctive case needs to be treated separately, because strong and weak satisfaction are defined differently for disjunctive formulae. First, the weak satisfaction relation: $(E', A') \models_w \phi \vee \psi$ iff $(E', A') \models_w \phi$ or $(E', A') \models_w \psi$, by hypothesis $(E, A) \models_w \phi$ or $(E, A) \models_w \psi$ iff $(E, A) \models_w \phi \vee \psi$.

  ($\vee_s$) Strong satisfaction for disjunction is defined for literals only. Hence, for literals $\alpha_1, \ldots, \alpha_n$, $(E', A') \models_s \alpha_1 \vee \ldots \vee \alpha_n$ iff $[[(E', A') \models_s \alpha_1$ or $\ldots$ or $(E', A') \models_s \alpha_n]$ and $\forall\, i$ s.t. $1 \leq i \leq n$ $[(E', A') \models_s \neg\, \alpha_i$ implies $(E', A') \models_s \otimes(\alpha_1 \vee \ldots \vee \alpha_n, \alpha_i)]]$.
  By base case $[[(E, A) \models_s \alpha_1$ or $\ldots$ or $(E, A) \models_s \alpha_n]$ and $\forall\, i$ s.t. $1 \leq i \leq n$ $[(E, A) \models_s \neg\, \alpha_i$ implies $(E, A) \models_s \otimes(\alpha_1 \vee \ldots \vee \alpha_n, \alpha_i)]]$ and by definition of strong satisfaction $(E, A) \models_s \alpha_1 \vee \ldots \vee \alpha_n$.

  The other propositional cases follow similarly.

  We consider one of the quantifier cases (the others follow similarly).

  ($\exists_{\rightarrow}$) Suppose $(E, A) \models_s (\exists\, x.\phi) \vee \psi$. Then for some x-variant B of A, $(E, B) \models_s \phi \vee \psi$. By the induction hypothesis, $(E', B') \models_s \phi \vee \psi$. But B' is an x-variant of A', and so $(E', A') \models_s (\exists\, x.\phi) \vee \psi$. Similar for weak satisfaction.

  ($\exists_{\leftarrow}$) Suppose $(E', A') \models_s (\exists\, x.\phi) \vee \psi$. Then for some x-variant V of A', $(E', V) \models_s \phi \vee \psi$. Define an assignment B in E as follows: On variables other than x, B agrees with A, and on x, x^A is some arbitrary member of x^V (x^V is a member of $U'(\mathcal{L})$, hence it is an equivalence class and thus we can choose any member). Then B is an x-variant of A, and it also easy to see that $B' = V$. Then $(E', B') \models_s \phi \vee \psi$. so by induction hypothesis $(E, B) \models_s \phi \vee \psi$, and therefore $(E, A) \models_s (\exists x.\phi) \vee \psi$. Similarly for weak satisfaction. $\square$

5.4 Equality Tableau Rules

The aim of this chapter is to develop a proof procedure incorporating reasoning about equalities. Basically, it is sufficient to add the equality rules to the set of assumptions. However, we can also incorporate equality rules explicitly into the tableau method. Adding equality to the semantic tableau for classical logic has been discussed, for example, by (Reeves, 1987), (Fitting, 1996) and (Beckert, 1997).

Definition 5.4.1 (Tableau Equality Rules)
The following are the EQ-rules for QC semantic tableau, where s and t are terms, f is a function symbol and α is a literal.

Reflexivity: $$\frac{}{t \approx t}$$

Replacement: $$\frac{\begin{array}{c}\alpha(s)\\ s \approx t\end{array}}{\alpha(t)}$$

Inequality rules:

$$\frac{\neg\,(f(s_1, \ldots, s_n) \approx f(t_1, \ldots, t_n))}{\neg\,(s_1 \approx t_1) \mid \ldots \mid \neg\,(s_n \approx t_n)} \qquad \frac{\begin{array}{c}\alpha(s_1, \ldots, s_n)\\ \neg\,\alpha(t_1, \ldots, t_n)\end{array}}{\neg\,(s_1 \approx t_1) \mid \ldots \mid \neg\,(s_n \approx t_n)}$$

Note, the tableau rule $\dfrac{s \approx t}{f(s) \approx f(t)}$ is implicitly given due to the reflexivity and replacement rules, i.e. by reflexivity we have $f(s) \approx f(s)$ and by assumption $s \approx t$ thus it follows by replacement $f(s) \approx f(t)$.

In a simplified notation, the tableau U-Rules are given by

$$\frac{(t \approx t)^*}{\text{closed}} \qquad \frac{(\alpha(t))^*}{(\alpha(s))^* \mid (s \approx t)^*} \qquad \frac{(f(s) \approx f(t))^*}{(s \approx t)^*} \qquad \frac{(\neg\,(s \approx t))^*}{(\alpha(s))^* \mid (\neg\,\alpha(t))^*}$$

but they are dismissable because each can be simulated by the EQ-rules for the QC semantic tableau as introduced above. First, if we derived $(t \approx t)^*$ in some branch then we can close that branch by using the reflexivity rule to add $t \approx t$ to the end of it. After applying the second rule we need to establish $\alpha(s)$ and $s \approx t$ to close each branch. However, given both we can apply the replacement rule to derive $\alpha(t)$ which closes the branch with $(\alpha(t))^*$ in it. Applying the function U-rule results in showing $s \approx t$ to close it. However, given this and reflexivity we obtain $f(s) \approx f(t)$ which would close the branch, too. Finally, the last rule can be simulated using the inequality rule for relation symbols. Consequently we do not require the use of the equality U-rules.

We illustrate the use of the tableau rules with a couple of examples. The following reasoning tree shows an example of how to use equality and function symbols. We show $\{\forall\, x, y.(x \approx y \wedge f(y) \approx g(y))\} \vdash_{Q_\approx} \forall\, x, y.(h(f(x)) \approx h(g(y)))$.

$$
\begin{array}{c}
\forall\, x, y.(x \approx y \wedge f(y) \approx g(y)), (\forall\, x, y.(h(f(x)) \approx h(g(y))))^* \\
| \\
a \approx b, f(b) \approx g(b) \\
| \\
f(a) \approx g(b) \\
| \\
h(f(a)) \approx h(f(a)) \\
| \\
h(f(a)) \approx h(g(b)) \\
| \\
(h(f(a) \approx h(g(b)))^* \\
| \\
\text{closed}
\end{array}
$$

Next, we use equality, function symbols and predicates. To construct the tree below we apply symmetry and transitivity. We already established the validity of these rules on the semantic level but will not repeat this argument here. However, using both properties of equality shortens the proof considerably. We show $\{\forall\, x, y.(f(x) \approx g(y) \Rightarrow p(x, y)), f(a) \approx c, g(b) \approx c\} \vdash_{Q_\approx} p(a, b)$.

$$
\begin{array}{c}
\forall\, x, y.(f(x) \approx g(y) \Rightarrow p(x, y)), f(a) \approx c, g(b) \approx c, (p(a, b))^* \\
| \\
f(a) \approx g(b) \Rightarrow p(a, b) \\
| \\
\neg\, (f(a) \approx g(b)) \vee p(a, b) \\
\diagup \qquad \diagdown \\
\begin{array}{cc}
\begin{array}{c}
(f(a) \approx g(b))^* \\
| \\
f(a) \approx c, g(b) \approx c \\
| \\
f(a) \approx g(b) \\
| \\
\text{closed}
\end{array}
&
\begin{array}{c}
p(a, b) \\
| \\
\text{closed} \\
\\
\\
\\
\\
\end{array}
\end{array}
\end{array}
$$

5.5 Soundness and Completeness

We need to establish the link between the QC tableau method and the QCL semantics. We need to show that we can only prove with the QC semantic tableau method what is satisfiable by QCL, i.e. soundness, and that we can prove everything that is satisfiable, i.e. completeness. Hunter showed that a set of assumptions Δ implies a conclusion φ by QCL ($\Delta \vDash_Q \varphi$), if and only if a QC tableau for Δ and conclusion φ is closed ($\Delta \vdash_Q \varphi$). We extend this proof to QCL with equality.

Theorem 5.5.1
For any set of formulae $\Delta \subseteq \mathcal{L}$ and any formula $\varphi \in \mathcal{L}$, a quasi-classical tableau with equality for Δ and φ is closed if and only if $\Delta \vDash_{Q\approx} \varphi$.

The basic idea of the proof relies on the fact that a tableau method is sound and complete if each tableau rule is sound and complete. Hunter already uses this principle thus we have little to change from the case without equality to the case with equality.

Soundness of the Tableau Rules

Basically, we need to show that the application of a tableau rule or equality rule to a tableau that is satisfiable in a normal model will produce another tableau that is satisfiable in the same normal model.

Lemma 5.5.1 (Soundness S-rules)
Each tableau rule given in definition 4.4.11 and definition 5.4.1 is sound in the following sense: If $\phi \in \mathcal{L}^*$ is a formula above the line, and $\varphi \in \mathcal{L}^*$ is a formula below the line, and E' is a normal model such that $E' \vDash_s \phi$, then $E' \vDash_s \varphi$.

Proof
According to (Hunter, 2001), the tableau rules in definition 4.4.11 are sound in the sense that if $\phi \in \mathcal{L}^*$ is a formula above the line, and $\varphi \in \mathcal{L}^*$ is a formula below the line, and E is a model such that $E \vDash_s \phi$, then $E \vDash_s \varphi$. Because $E' \vDash_s \phi$ iff $E \vDash_s \phi$ and $E \vDash_s \varphi$ iff $E' \vDash_s \varphi$ it follows that the tableau rules in definition 4.4.11 are sound in the above sense.

The EQ reflexivity rule is sound because $t \approx t$ is the formula below the line and according to definition 5.3.1 we consider only those models such that for all E, A, $(E, A) \vDash_s t \approx t$, i.e. $(E', A') \vDash_s t \approx t$. The EQ replacement rule is sound because according to definition 5.3.1 for all E, A, if $(E, A) \vDash_s \alpha(s)$ and $(E, A) \vDash_s s \approx t$ then $(E, A) \vDash_s \alpha(t)$ and using lemma 5.3.5 it follows if $(E', A') \vDash_s \alpha(s)$ and $(E', A') \vDash_s s \approx t$ then $(E', A') \vDash_s \alpha(t)$ for all E', A'. Similarly for the inequality rules. $\square$

Lemma 5.5.2 (Soundness U-rules)
Each tableau rule given in definition 4.4.12 is sound in the following sense: If $\phi \in \mathcal{L}^*$ is a formula above the line, and $\varphi \in \mathcal{L}^*$ is a formula below the line, and E' is a normal model such that $E' \models_w \phi$, then $E' \models_w \varphi$.

Proof
This follows from (Hunter, 2001), i.e. each tableau rule given in definition 4.4.12 is sound in the following sense: If $\phi \in \mathcal{L}^*$ is a formula above the line, and $\varphi \in \mathcal{L}^*$ is a formula below the line, and E is a model such that $E \models_w \phi$, then $E \models_w \varphi$; and from $E' \models_w \phi$ iff $E \models_w \phi$ and $E \models_w \varphi$ iff $E' \models_w \varphi$. There are no equality U-rules thus we are done. $\square$

Completeness of the Tableau Rules

Lemma 5.5.3 (Completeness S-rules)
The set of tableau rules given in definitions 4.4.11 and 5.4.1 is complete in the following sense: If $\phi \in \mathcal{L}^*$ is a formula in a branch of a QC semantic tableau, and there is a pair (E', A') such that $(E', A') \models_s \phi$, and according to definitions 4.4.7 and 5.3.1 there is a derivation of the form $(E', A') \models_s \phi$ implies $(E', A') \models_s \varphi$, then φ can be obtained as a formula in the branch by using the S-rules in definition 4.4.11 or the equality rules in definition 5.4.1.

Proof
The completeness of the S-rules follows from (Hunter, 2001) and lemma 5.3.5. It remains to be shown the completeness of the EQ-rules. The EQ rules for the stong satisfaction relation are captured in definition 5.3.1. According to definition 5.3.1 always $(E, A) \models_s t \approx t$, i.e. $(E', A') \models_s t \approx t$ which can be obtained by the reflexivity rule. Given $\alpha(s)$ and $s \approx t$ in a branch, according to definition 5.3.1 there is a derivation $(E, A) \models_s \alpha(s)$ and $(E, A) \models_s s \approx t$ implies $(E', A') \models_s \alpha(t)$ which can be obtained using the replacement rule. Similarly for the inequality rules. $\square$

Lemma 5.5.4 (Completeness U-rules)
The set of tableau rules given in definition 4.4.12 and 5.4.1 is complete in the following sense: If $\phi \in \mathcal{L}^*$ is a formula in a branch of a QC semantic tableau, and there is a pair (E', A') such that $(E', A') \models_w \phi$, and according to definitions 4.4.8 and 5.3.2 there is a derivation of the form $(E', A') \models_w \phi$ implies $(E', A') \models_w \varphi$, then φ can be obtained as a formula in the branch by using the U-rules in definition 4.4.12 or the equality rules in definition 5.4.1.

Proof
The completeness of the U-rules follows from (Hunter, 2001) and lemma 5.3.5. It remains to be shown the completeness of the EQ-rules. This follows basically from the earlier discussion that each EQ U-rule can be modelled using the EQ-rules. $\square$

Given the soundness and completeness of each of the tableau rules it is easy to show that the tableau method is sound and complete, i.e. for any set of formulae Δ and any formula φ there is a QC semantic tableau with equality for Δ and φ that is closed if and only if there is no model E such that $E \models_s \Delta$ and $E \models_w \varphi^*$.

This, however, has essentially been proved by (Hunter, 2001). According to the above lemmata each application of the S-rules, U-rules and EQ-rules is sound and complete. Consider a particular Δ and φ. There is a QC tableau with equality for Δ and φ that is closed iff every branch of the tableau with root $\Delta \cup \{\varphi^*\}$ is closed iff every branch of the tableau with root $\Delta \cup \{\varphi^*\}$ contains ϕ and ϕ^* for some ground literal ϕ iff there is no model for each branch of the tableau with root $\Delta \cup \{\varphi^*\}$ iff there is no model E such that $E \models_s \Delta$ and $E \models_w \varphi^*$.

5.6 The One-Point Rule

The notion of equality allows us to introduce or eliminate the existential quantifier. If a variable is found to be bound by an existential quantifier and it is identical to some given term, then we can replace all instances of the variable by that term and remove the existential quantifier. Consider the predicate $\exists\, x.(p(x) \wedge x \approx t)$. This states that there is a value for x for which the predicate $p(x) \wedge x \approx t$ holds. Obviously, t itself is a reasonable candidate for replacing x. The one-point rule in standard predicate logic expresses the following equivalence:

$$\exists\, x.(p(x) \wedge x \approx t) \equiv p(t) \;\; \text{[provided } x \text{ is not free in } t\text{]}$$

We are interested in preserving this rule in QCL, i.e. we want

$$\exists\, x.(p(x) \wedge x \approx t) \equiv_Q p(t)$$

under the same provision. This means, that the class of the strong models and the class of the weak models of the left and right hand side of this equivalence must be equal. We found that bi-directional QC derivability is a necessary condition to hold. It is, however, easy to see that for any formula p it is the case: $\exists\, x.(p(x) \wedge x \approx t) \vdash\dashv_Q p(t)$

$$\begin{array}{c}
\exists\, x.(p(x) \wedge x \approx t), (p(t))^* \\
| \\
p(t) \wedge t \approx t, (p(t))^* \\
| \\
p(t), t \approx t, (p(t))^* \\
| \\
\text{closed}
\end{array}
\qquad\qquad
\begin{array}{c}
p(t), (\exists\, x.(p(x) \wedge x \approx t))^* \\
| \\
p(t), (p(t) \wedge t \approx t)^* \\
\swarrow \qquad \searrow \\
\begin{array}{cc}
(p(t))^* & (t \approx t)^* \\
| & | \\
\text{closed} & t \approx t \\
 & | \\
 & \text{closed}
\end{array}
\end{array}$$

The strong model classes of both sides are equal if for every strong model of $\exists\, x.(p(x) \wedge x \approx t)$ there is an equivalent strong model for $p(t)$, i.e. if for every model E it holds $E \vDash_s \exists\, x.(p(x) \wedge x \approx t)$ iff $E \vDash_s p(t)$. The case for the weak satisfaction relation follows similarly.

$$
\begin{array}{ll}
 & E \vDash_s p(t) \\
\text{iff } \{\text{for all assignments } A\} & \\
 & (E, A) \vDash_s p(t) \\
\text{iff } \{\text{Consider normal models}\} & \\
 & (E', A') \vDash_s p(t) \\
\text{iff } \{\text{By Reflexivity}\} & \\
 & (E', A') \vDash_s p(t) \text{ and } (E', A') \vDash_s t \approx t \\
\text{iff } \{\text{Definition}\} & \\
 & (E', A') \vDash_s p(t) \wedge t \approx t \\
\text{iff } \{B' \text{ is x-variant assignment of } A'\} & \\
 & (E', B') \vDash_s \exists\, x.(p(x) \wedge x \approx t) \\
\text{iff} & \\
 & (E, B) \vDash_s \exists\, x.(p(x) \wedge x \approx t) \\
\text{iff } \{\text{for all assignments } B\} & \\
 & E \vDash_s \exists\, x.(p(x) \wedge x \approx t)
\end{array}
$$

5.7 Discussion and Summary

Adding equality to a paraconsistent logic has previously been considered by (Batens and De Clercq, 1999) and (da Costa, 2000). Basically, both approaches are similar to ours by being based on adding reflexivity and the replacement principle.

We find, however, that equivalence classes can be trivialised in the presence of inconsistency. For example, under the assumption $1 \approx 2$ all numbers collapse into one equivalence class, i.e. all numbers are provably equal. This problem can be contributed to the richness of reasoning with equality, in particular to functionality. (Mortensen, 1995, p. 12f) notes:

> Fortunately or unfortunately, the methods and results in this book [(Mortensen, 1995)] indicate that the 'essence' of mathematics is deeper than paraconsistentists have thought [...].
>
> [...] classical mathematics, interested in functionality, concentrated on the consistent subtheory [...]

> [...] it is not true that there are no interactions between functionality and inconsistency or incompleteness. [...] this can lead to interesting insights about functionality;

(Mortensen, 1995) suggests a controlled relaxation of functionality to avoid such trivialities and (Vermeir, 2001) investigates a new axiomatisation of inconsistent arithmetic by means of inconsistency-adaptive logics (see (Batens, 1999) and (Batens, 2000), for example). The latter approach, however, abandons the property of monotonicity which we identified as desirable.

Surely, this issue of inconsistency and arithmetic needs to be further investigated. Note, however, given such inconsistency between numbers does not necessarily mean that the given theory is trivialised too. For example, $1 \approx 2 \wedge \alpha$, for some formula α, does not imply that we can infer $\neg\, \alpha$ using QCL with equality.

In this chapter we introduced the notion of equality to the semantics of QCL. We showed that extra tableau rules to reason about equality are sound and complete with respect to the given semantics. Given equality we established the validity of the one-point rule, a commonly used rule to introduce and remove existential quantification. We will use QCL with equality in the next chapter to reason about formal specifications written in the Z notation.

Chapter 6

Formal Reasoning about Inconsistent Z Specifications using Quasi-Classical Logic

The aim of this chapter is to discuss what formal support can be given to the process of analysing and refining Z specifications in a context that explicitly allows and recognises inconsistencies. This work is part of the wider area of research on living with inconsistencies, rather than eradicating them. We discussed in Chapter 4 that logicians have developed a range of logics to continue to reason in the presence of inconsistencies and we introduced in particular one representative of such paraconsistent logics, namely Hunter's quasi-classical logic (QCL). Here we apply QCL to analyse inconsistent Z schemas. Quasi-classical logic allows us to derive less, but more "useful", information, in the presence of inconsistency. Consequently, inconsistent Z specifications can be analysed in more depth than at present.

Part of the analysis of a Z operation is the calculation of the precondition. In the presence of an inconsistency, however, information about the intended application of the operation may be lost. It is our aim to regain this information. We introduce a new classification of precondition areas, based on the notions of definedness, overdefinedness and undefinedness. We discuss an option for determining these areas which is based on quasi-classical reasoning.

Refinement is the process of developing abstract specifications into more concrete ones. This is a major development tool for formal specifications. Here, we consider the refinement of inconsistent operation schemas. Given an inconsistent predicate in an operation, any other predicate replacing it is a valid refinement. This, however, allows a wide range of non-intuitive refinements. We claim that inconsistent operations carry information that should be preserved during refinement, like consistent operations do. We develop a refinement method based on quasi-classical reasoning to account for this.

6.1 Introduction

The purpose of this chapter is to discuss how to reason in the presence of inconsistencies in a formal setting. Although this might sound strange, specifications, especially large ones, are often inconsistent at some level. Inconsistencies range from contradictory descriptions of the system at hand to contradictions specified in the operations. A significant proportion of the specification analysis process is then devoted to detecting and eliminating such inconsistencies, because, classically (and intuitively), inconsistencies in specifications are regarded as undesirable.

6.1.1 Motivation

Those involved in large scale software engineering in practice treat inconsistencies as a fact of life. They occur frequently in large projects and need to be tolerated (possibly for some time) and managed, rather than eradicated immediately. This has led to a considerable amount of research on the development of tools and techniques for living with inconsistencies (Ghezzi and Nuseibeh, 1998; Ghezzi and Nuseibeh, 1999), (Balzer, 1991), (Schwanke and Kaiser, 1988), and handling inconsistencies (Finkelstein et al., 1994), (Hunter and Nuseibeh, 1998). The general aim of such work is to provide practical support for deciding if, when, and how to remove inconsistencies, and to possibly reason in the presence of inconsistencies.

Although the techniques and tools developed for this approach have had a certain amount of success they have, however, mainly focused on informal and semi-formal specification techniques. There has been recent work on more formal approaches (Hunter and Nuseibeh, 1997) but these have largely concentrated on purely logical issues, not connecting them to current specification languages. We are interested in seeing what support we can give for the process of living with inconsistencies in a specification notation, namely Z.

Our purpose here is to explore the issue of handling inconsistencies in Z, especially those present in operations. The general aim is, in the presence of inconsistency, not to immediately derive falsehood, but to rather allow further, intermediate, reasoning on other aspects of the state, operation, or specification. This should enable us to infer more useful conclusions from inconsistent Z schemas or specifications. One particular aspect is how tolerating inconsistencies can benefit the development process from abstract to concrete specifications.

6.1.2 The Use of Quasi-Classical Logic

In classical predicate logic, on which Z is based, inconsistent information results in triviality, because everything can be inferred from it. This, in turn, renders the information useless, when, in fact, there may be further valid inferences we wish to make. However, there are several ways of handling inconsistent information. One is to divide the pieces of information into (possibly maximal) consistent subsets (Rescher and Manor, 1970), another is paraconsistent reasoning. The latter allows the derivation of only non-trivial inferences from inconsistent information, i.e. not everything can be inferred.

One representative of paraconsistent logics is quasi-classical logic, developed by (Besnard and Hunter, 1995). We introduced the semantics and proof theory of QCL with Equality in the previous chapter. The key to QCL is that it allows only the derivation of information already present in a given theory, even though that theory might be inconsistent. This feature is what we need to analyse inconsistent Z operations. QCL is not so much aimed at reasoning about the truth in the real world but about handling beliefs. This seems to be compliant with the idea of formal specification where we gather requirements of a system yet to be built.

The main advantage of QCL, in comparison with many other paraconsistent logics, is that the logical connectives behave classically. Therefore, we believe that QCL is more suitable for our application to Z, because specifiers and analysts will already be familiar with the notation and meaning of the connectives.

6.1.3 Hypothesis

In this chapter we show that quasi-classical logic enables us to analyse inconsistent operations specified in the formal notation Z. QCL allows us to infer less but more useful information in the presence of inconsistencies. We understand the term "useful" with respect to the problem of triviality arising from inconsistency, i.e. everything is derivable. In comparison to standard predicate logic, QCL restricts the amount of information inferable from inconsistent premises.

Furthermore, quasi-classical logic is a tool to direct the process of refinement of inconsistent operation schemas such that fewer but more useful refinements remain. In standard Z, an inconsistent predicate in an operation can be refined by any other predicate. For example, we present an inconsistent operation to add a user to a library but refine it by an operation removing a user. QCL allows us to distinguish between some forms of unwanted refinements and desired refinements.

Quasi-classical logic proves helpful for both tasks. However, we found that QCL itself needs to be further developed to suit this particular application within the notation Z. We already reported some of the necessary extensions, like equality and logical equivalence, in the previous chapter. Here, we identify further areas to guide the development of QCL. In particular, QCL and its application to set theory come to mind.

6.1.4 Scope

In this work, we only consider the issue of local inconsistency. A schema can have an inconsistent, i.e. unsatisfiable, predicate. If such a schema is an operation schema, then the operation may not be applicable at all, or only parts of the operation are applicable. This is due to the fact that contradictions in an operation only restrict the precondition of that operation which characterises where the operation is feasible. In the case of the schema describing the state of the system, the entire part of the system governed by that state is not implementable. These kinds of errors are local in the sense that the specification of other components of the system may still be meaningful (although it is usually assumed implicitly, in a state and operation specification that at least one possible (initial) value of the state should exist).

In contrast, global inconsistencies are more serious, because they make an entire specification unsatisfiable. They occur if some axiom schema, generic schema, or constraint is unsatisfiable. Furthermore, they can arise due to a combination of different paragraphs of a specification, each being consistent. However, set declarations, abbreviations, and schema definitions cannot introduce global inconsistency. In this work we do not consider global inconsistencies though we believe that our work could contribute to the research on analysing globally inconsistent specifications, too.

There is another issue related to inconsistency. (Henson and Reeves, 2000) investigate the logic of Z. Their intent is to define Z based on proof theory. As part of their research, (Henson, 1998) reported that a previous development of the logic of Z, as published by (Nicholls, 1995), was inconsistent. We do not investigate the consistency of Z but the consistency of specifications written in Z, and in particular of their operations.

6.1.5 Outline

This chapter is structured as follows. First, we present a small example of a library system specified using the Z notation. We introduce an inconsistency to use it as an illustration throughout this chapter. Next, we use quasi-classical logic to infer some properties of a part of this specification. We also show, that QCL allows fewer inferences than standard predicate logic. In Section 6.4, we apply QCL to the process of calculating the precondition of inconsistent operation schemas. It was here, that we found that we need QCL to possess a notion of logical equivalence as introduced in Chapter 4. Given the notion of a quasi-classical precondition, we turn to the refinement process of inconsistent operations in Section 6.5. Following the notions of standard refinement, we establish the principles of quasi-classical applicability and QC correctness. We summarize this chapter in Section 6.6.

6.2 An Inconsistent Library Specification in Z

The following example presents a specification of a simple library system. We have been inspired by some of our students who developed a similar system (including the inconsistency) in their 2002 exam on Software Engineering.

Our library consists of users who are allowed to borrow books. The sets $NAME$ of user's names, and $BOOK$, of books, are taken as given; their structure is of no concern for this detail of specification.

$$[NAME, BOOK]$$

The state of the library is modelled by the schema *Library*. The *Library* schema uses a partial function *borrowed* to record the books borrowed by a user. The set *users* contains the names of the people who joined the library.

Library

$$\begin{array}{l} users : \mathbb{P}\, NAME \\ borrowed : NAME \rightarrow\!\!\!\!\!\!+\;\, \mathbb{P}\, BOOK \\ \hline users = \operatorname{dom} borrowed \end{array}$$

Initially, there are no members of the library and, therefore, no books are borrowed.

InitLibrary

$$\begin{array}{l} Library' \\ \hline borrowed' = \varnothing \\ users' = \varnothing \end{array}$$

A sensible condition to impose on the state schema *Library* is that it allows at least one initial state. We specified such an initial state by the schema *InitLibrary*. We use Z/EVES to show that *InitLibrary* is indeed an initial state of *Library*.

```
=> try \exists Library' @ InitLibrary;
=> prove by reduce;
```

Proving gives ...

true

Next, we specify the operation *AddUser* to register a new user, given a name. To register, the user must not be a member of the library. The record of books borrowed remains unchanged.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{AddUser}} \\
\hline
\Delta \textit{Library} \\
\textit{name}? : \textit{NAME} \\
\hline
\textit{name}? \notin \textit{users} \\
\textit{users}' = \textit{users} \cup \{\textit{name}?\} \\
\textit{borrowed}' = \textit{borrowed} \\
\hline
\end{array}
$$

Operation schemas can be analysed in different ways. It is common to determine the precondition of the operation to find those states where the operation is applicable. We use Z/EVES as a starting point for this calculation.

```
=> try \pre AddUser;
=> prove by reduce;
```

Proving gives ...

$$
\begin{aligned}
&\textit{borrowed} \in \mathbb{P}(\textit{NAME} \times \mathbb{P}\, \textit{BOOK}) \\
\wedge\ &\textit{borrowed} \in \textit{NAME} \nrightarrow \mathbb{P}\, \textit{BOOK} \\
\wedge\ &\textit{users} = \operatorname{dom} \textit{borrowed} \\
\wedge\ &\textit{name}? \in \textit{NAME} \\
\wedge\ &\neg\ \textit{name}? \in \operatorname{dom} \textit{borrowed} \\
\wedge\ &\operatorname{dom} \textit{borrowed} = \{\textit{name}?\} \cup \operatorname{dom} \textit{borrowed}
\end{aligned}
$$

which, in turn, simplifies to

$$\operatorname{pre} \textit{AddUser} = [\textit{Library}, \textit{name}? : \textit{NAME} \mid \textit{false}]$$

We find, the operation *AddUser* is never applicable. This suggests an inconsistency in the specification. Therefore, we can use *AddUser* as one example for the work we present in the next sections.

Furthermore, we specify the operation of removing a user from the library system. The user to be removed must be registered but is not allowed to have any books on loan. We report the outcome of the operation in case the operation does not succeed. Therefore, we introduce the type

$$\textit{Report} ::= \textit{success} \mid \textit{failure}$$

before turning to the actual operation

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{RemoveUser}} \\
\hline
\Delta Library \\
name? : NAME \\
out! : Report \\
\hline
(name? \notin users \land borrowed' = borrowed \land out! = failure) \lor \\
(name? \in users \land name? \notin \operatorname{dom} borrowed \land \\
\quad users' = users \setminus \{name?\} \land borrowed' = borrowed \land \\
\quad out! = success) \\
\hline
\end{array}
$$

Using Z/EVES, we also determine the precondition of the operation *RemoveUser*, to identify those states where the operation is applicable.

```
=> try \pre RemoveUser;
=> prove by reduce;
```

Proving gives ...

$$
\begin{array}{l}
\quad borrowed \in \mathbb{P}(NAME \times \mathbb{P}\, BOOK) \\
\land\ borrowed \in NAME \pfun \mathbb{P}\, BOOK \\
\land\ users = \operatorname{dom} borrowed \\
\land\ name? \in NAME \\
\land\ \neg\ name? \in \operatorname{dom} borrowed
\end{array}
$$

i.e.

$$\operatorname{pre} RemoveUser = [Library, name? : NAME \mid name? \notin \operatorname{dom} borrowed]$$

We know, the operation *RemoveUser* is not correctly specified because we expected to cover all cases. However, the calculated precondition of *RemoveUser* may actually distract from finding the real problem, because the given precondition is designed to restrict the remove operation to those users who returned all books. Actually, the condition arose from the predicate including $out! = failure$ and not from the predicate $out! = success$ which one might have assumed.

This specification is small enough to look for the mistakes by inspecting all the schemas involved. However, consider a larger system with several schemas included. Inspection becomes a laborious task. Below we introduce mechanisms to support the analysis of such inconsistent specifications. Also, we introduce an approach to refining such inconsistent schema preserving the intended application. The given specification is used as an example to guide our development and to demonstrate and validate our results.

6.3 Investigating Inconsistent Z Specifications

One of the distinguishing features of formal methods is the ability to formally investigate specifications. Formal reasoning enables us to infer new properties or to check whether a set of properties holds for a given specification. Such properties may be demanded in the informal requirements for the specification, or they may be identified as key points about the specification.

Investigating an inconsistent specification is a challenge, because, in classical predicate logic, a contradiction enables the reasoner to infer any property. We claim, that this is not very helpful in the process of analysing inconsistent specifications. We introduced quasi-classical logic as a logic that deals with this problem of triviality differently. In QCL not every property can be inferred from an inconsistency. Therefore, QCL is more suitable to derive more useful information about an inconsistent specification.

For example, we introduced the operation *AddUser* to describe the task of adding a new member to the library. This operation should result in an increase of the number of members, i.e.

$$AddUser \vdash_{Q_\approx} name? \notin users \Rightarrow \#users' > \#users$$

and indeed we can show this

$$\begin{array}{c}
name? \notin users \Rightarrow \#(users \cap \{name?\}) = 0, \#\{name?\} = 1, \\
users = \operatorname{dom} borrowed, user' = \operatorname{dom} borrowed', \\
name? \notin users, users' = users \cup \{name?\}, borrowed' = borrowed, \\
(name? \notin users \Rightarrow \#users' > \#users)^* \\
| \\
(\neg\, (name? \notin users))^*, (\#users' > \#users)^* \\
| \\
\#users' = \#(users \cup \{name?\}) \\
| \\
\#users' = \#users + \#\{name?\} - \#(users \cap \{name?\}) \\
\begin{array}{cc}
\diagup & \diagdown \\
\neg\, (name? \notin users) & \#(users \cap \{name?\}) = 0 \\
| & | \\
\text{closed} & \#users' = \#users + \#\{name?\} - 0 \\
 & | \\
 & \#users' = \#users + 1 \\
 & | \\
 & \#users' > \#users \\
 & | \\
 & \text{closed}
\end{array}
\end{array}$$

We introduced the predicates $name? \notin users \Rightarrow \#(users \cap \{name?\} = 0)$ and $\#\{name?\} = 1$ as extra assumptions. Both predicates are derived from the mathematical toolkit of Z. Often, such assumptions are not made explicit and proofs in Z are, therefore, semi-formal. Actually, our proof is only semi-formal, too. For example, we did not introduce the laws about the length of sets nor that the value of a number increases through addition. As such, we follow the Z "tradition" and apply obvious replacements without introducing them explicitly.

Due to the information provided in *AddUser* we are also able to show that the amount of users of this library system remains unchanged, i.e.

$$AddUser \vdash_{Q_{\approx}} \#users' = \#users$$

which is validated by the following proof tree

$$\begin{array}{c} users = \operatorname{dom} borrowed, user' = \operatorname{dom} borrowed', \\ name? \notin users, users' = users \cup \{name?\}, borrowed' = borrowed, \\ (\#users' = \#users)^{*} \\ | \\ \operatorname{dom} borrowed' = \operatorname{dom} borrowed \\ | \\ users' = users \\ | \\ \#users' = \#users \end{array}$$

The advantage of quasi-classical logic over classical predicate logic becomes apparent when we try to prove that adding a new member could actually reduce the number of users of the library. Using standard logic we would be able to infer this statement but not when we use quasi-classical logic, i.e.

$$AddUser \nvdash_{Q_{\approx}} \#users' < \#users$$

Apart from the operation *AddUser* we introduced the operation *RemoveUser*. Using QCL and its proof theory we also establish the following properties.

1. $RemoveUser \vdash_{Q_{\approx}} name? \notin users \Rightarrow users' = users$
2. $RemoveUser \vdash_{Q_{\approx}} name? \notin users \Rightarrow borrowed' = borrowed$
3. $RemoveUser \vdash_{Q_{\approx}} name? \notin users \Rightarrow out! = failure$
4. $RemoveUser \vdash_{Q_{\approx}} name? \notin users \Rightarrow \#users' < \#users$
5. $RemoveUser \vdash_{Q_{\approx}} name? \in users \Rightarrow \#users' < \#users$
6. $RemoveUser \nvdash_{Q_{\approx}} name? \in users \Rightarrow \#users' > \#users$
7. $RemoveUser \nvdash_{Q_{\approx}} name? \in users \Rightarrow out! = failure$

Examples (1)-(3) establish some facts about the classically applicable case. Example (4) shows, the inconsistency in the operation also allows to infer that the number of users can be reduced even if the user is not a member of the library. The same holds if the user is a member of the library, which we intended. The examples (6) and (7), however, demonstrate that not everything is inferable from an inconsistency. Using QCL we cannot establish those "undesired" facts. Standard predicate logic, however, verifies those inferences. We use Z/EVES to demonstrate this.

```
=> try RemoveUser \implies
          (name? \in users \implies \# users' > \# users);
=> prove by reduce;
```

Proving gives ...

true

```
=> try RemoveUser \implies
          (name?\in users\implies out!=failure);
=> prove by reduce;
```

Proving gives ...

true

We promised that QCL will help us to infer less but more useful information. The above examples demonstrates the value of this approach. Using QCL enables the reasoner to validate only information which is present in a specification, even if it is inconsistent, but no more. Next, we look at further issues of reasoning about formal specifications. First, we investigate quasi-classical preconditions of inconsistent specifications. Afterwards, we turn to the problem of refinement.

6.4 Quasi-Classical Preconditions of Inconsistent Z Specifications

(Woodcock and Davies, 1996) write: "The precondition of an operation schema describes the set of states for which the outcome of the operation is properly defined." In standard Z, this means that the outcome of the operation needs to be defined and must not be overdefined, i.e. inconsistent. Overdefinedness and undefinedness are, in standard Z, inseparable. However, when using alternative forms of reasoning, undefinedness and overdefinedness can be distinguished.

We believe that from the developer's point of view undefinedness and overdefinedness should be treated differently. In the one case, the developer had no intention to specify the effect of an operation, therefore it was left undefined. In the other case, a specification mistake or an unforeseen interaction of parts of the specification rendered the operation inapplicable. Being able to formally separate these situations will help to analyse the specification more deeply and to develop it further in a more directed way.

The aim of this section is to investigate the effect of calculating the preconditions of possible inconsistent operation schemas using quasi-classical logic. We demonstrate that QCL is able to separate the undefined part of an operation from the overdefined. We also investigate QCL itself by applying it to such tasks. We find that QCL needs to be further developed to be fully suitable for our needs.

6.4.1 The Quasi-Classical Precondition

The precondition of an operation describes all the initial states in which the operation is defined. To us, an operation is defined if it is consistently defined or possibly overdefined. Given an operation schema Op we write

$$\mathrm{pre}_Q\ Op$$

to denote the quasi-classical precondition of Op. This is another schema obtained by hiding all the components from Op that correspond to the after state of the operation including any outputs. If the state of the system is modelled by a schema S, and $outs!$ is the list of outputs associated with the operation, then the QC precondition of Op on a state schema S is defined by

$$\mathrm{pre}_Q\ Op = \exists\, S', outs! \bullet Op$$

At first, this definition seems identical to the standard definition of the precondition. However, we now consider QCL as the background logic. Therefore, inconsistencies do not evaluate to false and the notion of logical equivalence is changed, too. Thus, the classical and quasi-classical precondition of an operation are different in their effect.

The QC precondition of the operation schema $AddUser$, which describes the effect of adding a new member to the library, is given by

$$\mathrm{pre}_Q\ AddUser = \exists\, Library' \bullet AddUser$$

Using the standard rules of quantification and schema expansion this results in

$$
\begin{array}{|l}
\hline
\textit{PreAddUser} \\
\textit{Library} \\
\textit{name?} : \textit{NAME} \\
\hline
\exists\, \textit{Library}' \bullet \\
\qquad \textit{name?} \notin \textit{users} \land \\
\qquad \textit{users}' = \textit{users} \cup \{\textit{name?}\} \land \\
\qquad \textit{borrowed}' = \textit{borrowed} \\
\hline
\end{array}
$$

This schema describes the QC precondition of *AddUser*. However, this schema is over-complicated to simply identify the conditions under which *AddUser* is applicable. Next, we investigate how QCL can be used to simplify this schema to give a neater but logically equivalent statement.

6.4.2 Simplifying Quasi-Classical Preconditions

To simplify a precondition schema we need to perform a series of equivalence preserving steps to reduce the complexity of the given predicate. However, not only do we need each step to preserve equivalence but we need transitivity of this process, too. Otherwise, the resulting formula might not be logically equivalent to the starting one.

The problem of simplifying quasi-classical preconditions made us aware of the fact, that the issue of logical equivalence has not been covered by the published research on quasi-classical logic. One reason could be that logical equivalence is a simple property in QCL. We do not think so. Logical equivalence in QCL is more complicated due to the two satisfaction relations involved. Therefore, for example, bi-directional inference is not a valid notion of equivalence, because transitivity fails. In Chapter 4, we summarised our work on logical equivalence in QCL. Given $\equiv_Q$ to denote equivalence preserving steps in a proof, we simplify the precondition of *AddUser*.

$$
\begin{array}{ll}
 & \mathrm{pre}_Q\ \textit{AddUser} \\
\equiv_Q & \{\text{Definition of } \mathrm{pre}_Q\} \\
 & \exists\, \textit{Library}' \bullet \textit{AddUser} \\
\equiv_Q & \{\text{Schema Expansion and Quantification}\} \\
 & [\textit{Library};\ \textit{name?} : \textit{NAME} \mid \\
 & \qquad \exists\, \textit{users}' : \mathbb{P}\,\textit{NAME};\ \textit{borrowed}' : \textit{NAME} \mathrel{+\!\!\!\rightarrow} \mathbb{P}\,\textit{BOOK} \mid \\
 & \qquad\qquad \textit{users}' = \mathrm{dom}\,\textit{borrowed}' \bullet \textit{name?} \notin \textit{users} \land \\
 & \qquad\qquad\qquad \textit{users}' = \textit{users} \cup \{\textit{name?}\} \land \textit{borrowed}' = \textit{borrowed}]
\end{array}
$$

The One-Point Rule

According to the semantics of QCL we can eliminate an existential quantifier if there is an assignment of a fixed value to the bound variable. This step can formally be expressed by a derivation law, the so called one-point rule:

$$\exists x \bullet p(x) \land x = t \equiv_Q p(t) \;\; \text{[provided } x \text{ is not free in } t]$$

In the context of Z specifications we need to consider the typing information as well. Therefore, the precise one-point rule is slightly more complicated:

$$\exists x : T \bullet p(x) \land x = t \equiv_Q t \in T \land p(t)$$

Given the one-point rule, we further simplify the QC precondition of $AddUser$.

$$\begin{array}{l}
\quad [Library;\ name? : NAME \mid \\
\qquad \exists\, users' : \mathbb{P}\, NAME;\ borrowed' : NAME \pfun \mathbb{P}\, BOOK \mid \\
\qquad\quad users' = \operatorname{dom} borrowed' \bullet name? \notin users \land \\
\qquad\qquad users' = users \cup \{name?\} \land borrowed' = borrowed] \\
\equiv_Q \{\text{One-point rule on } borrowed'\} \\
\quad [Library;\ name? : NAME \mid \\
\qquad \exists\, users' : \mathbb{P}\, NAME \bullet borrowed \in NAME \pfun \mathbb{P}\, BOOK \land \\
\qquad\quad users' = \operatorname{dom} borrowed \land name? \notin users \land \\
\qquad\qquad users' = users \cup \{name?\}] \\
\equiv_Q \{\text{One-point rule on } users'\} \\
\quad [Library;\ name? : NAME \mid \\
\qquad users \in \mathbb{P}\, NAME \land borrowed \in NAME \pfun \mathbb{P}\, BOOK \land \\
\qquad\quad users \cup \{name?\} = \operatorname{dom} borrowed \land name? \notin users] \\
\equiv_Q \{\text{Type information provided in Library}\} \\
\quad [Library;\ name? : NAME \mid \\
\qquad users \cup \{name?\} = \operatorname{dom} borrowed \land name? \notin users] \\
\equiv_Q \{\text{Replacement of Equals, Symmetry}\} \\
\quad [Library;\ name? : NAME \mid \\
\qquad users = users \cup \{name?\} \land name? \notin users] \\
\equiv_Q \{\text{Replacement of Equals}\} \\
\quad [Library;\ name? : NAME \mid \\
\qquad users = users \cup \{name?\} \land name? \notin users \cup \{name?\}] \\
\equiv_Q \{\text{Set theory}\} \\
\quad [Library;\ name? : NAME \mid \\
\qquad users = users \cup \{name?\} \land name? \notin users \land name? \notin \{name?\}]
\end{array}$$

We could continue substituting the definition of *users* accordingly but we do not derive anything new. Therefore, the quasi-classical precondition schema of *AddUser* is

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{Pre}_Q \textit{AddUser}} \\ \hline
\textit{Library} \\
\textit{name}? : \textit{NAME} \\ \hline
\textit{name}? \notin \textit{users} \\
\textit{name}? \notin \{\textit{name}?\} \\
\textit{users} = \textit{users} \cup \{\textit{name}?\} \\ \hline
\end{array}$$

We interpret this QC precondition in the following way. The operation *AddUser* was designed to perform a task when a user *name*? is not a member of the set of *users*. However, there is an inconsistency present, which forces the constraint that the set of *users* must not change after adding a new user *name*?. This, however, is only possible, if *name*? is not a member of the set containing *name*?, which is clearly violating a basic set theoretic axiom. We believe that this quasi-classical precondition is more insightful than the classical precondition

$$\mathrm{pre}\,\textit{AddUser} = [\textit{Library};\ \textit{name}? : \textit{NAME} \mid \mathrm{false}]$$

The operation *AddUser* is defined for the case that $\textit{name}? \notin \textit{users}$. Unfortunately, it is also overdefined. We discuss below some advantages of calculating both the classical and quasi-classical precondition. Later, when considering refinement, we extend this work even further.

6.4.3 Using Classical and Quasi-Classical Preconditions

With the introduction of the quasi-classical precondition of an operation we have established a second notion of a precondition besides the standard notion as introduced in Chapter 2. We now investigate whether using both notions together gives some advantages to the process of analysing formal specifications.

Overdefinedness

Earlier we introduced the schema *RemoveUser* to specify the operation of removing a user from the library. We intended that an error message occurs if we try to remove a user who is not a member of the library. Furthermore, the operation of removing a user is only guaranteed if the user has no books on loan. We established the standard precondition as

$$\mathrm{pre}\,\textit{RemoveUser} = [\textit{Library}, \textit{name}? : \textit{NAME} \mid \textit{name}? \notin \mathrm{dom}\,\textit{borrowed}]$$

We also established the notion of a quasi-classical precondition which we now apply to the schema *RemoveUser*. We present the simplification steps in detail to demonstrate the approach on a second example.

$$
\begin{array}{l}
\quad \mathrm{pre}_Q\ RemoveUser \\
\equiv_Q \{\text{Definition of } \mathrm{pre}_Q\} \\
\quad \exists\, users', borrowed', out! \bullet Library, name? : NAME, out! : Report \mid \\
\qquad users' = \mathrm{dom}\, borrowed' \wedge \\
\qquad ((name? \notin users \wedge out! = failure \wedge borrowed' = borrowed) \vee \\
\qquad\ (name? \in users \wedge name? \notin \mathrm{dom}\, borrowed \wedge \\
\qquad\qquad users' = users \setminus \{name?\} \wedge borrowed' = borrowed \wedge \\
\qquad\qquad out! = success)) \\
\equiv_Q \{\text{OPR on } borrowed'\} \\
\quad \exists\, users', out! \bullet Library, name? : NAME, out! : Report \mid \\
\qquad users' = \mathrm{dom}\, borrowed \wedge \\
\qquad ((name? \notin users \wedge out! = failure) \vee \\
\qquad\ (name? \in users \wedge name? \notin \mathrm{dom}\, borrowed \wedge \\
\qquad\qquad users' = users \setminus \{name?\} \wedge out! = success)) \\
\equiv_Q \{\text{Distribution of Conjunction}\} \\
\quad \exists\, users', out! \bullet Library, name? : NAME, out! : Report \mid \\
\qquad (users' = \mathrm{dom}\, borrowed \wedge name? \notin users \wedge out! = failure) \vee \\
\qquad (users' = \mathrm{dom}\, borrowed \wedge name? \in users \wedge \\
\qquad\qquad name? \notin \mathrm{dom}\, borrowed \wedge users' = users \setminus \{name?\} \wedge \\
\qquad\qquad out! = success) \\
\equiv_Q \{\text{Distribution of Existential Quantification}\} \\
\quad Library, name? : NAME \mid \\
\quad\ \exists\, users', out! \bullet out! : Report \wedge \\
\qquad (users' = \mathrm{dom}\, borrowed \wedge name? \notin users \wedge out! = failure) \vee \\
\quad\ \exists\, users', out! \bullet out! : Report \wedge \\
\qquad (users' = \mathrm{dom}\, borrowed \wedge name? \in users \wedge \\
\qquad\qquad name? \notin \mathrm{dom}\, borrowed \wedge users' = users \setminus \{name?\} \wedge \\
\qquad\qquad out! = success) \\
\equiv_Q \{\text{OPR on } out!\ (2\text{x})\} \\
\quad Library, name? : NAME \mid \\
\quad\ \exists\, users' \bullet (users' = \mathrm{dom}\, borrowed \wedge name? \notin users) \vee \\
\quad\ \exists\, users' \bullet (users' = \mathrm{dom}\, borrowed \wedge name? \in users \wedge
\end{array}
$$

$$
\begin{array}{l}
\qquad\qquad\qquad name? \notin \mathrm{dom}\, borrowed \land users' = users \setminus \{name?\}) \\
\equiv_Q \{\text{OPR on } users' \text{ (2x)}\} \\
\quad Library, name? : NAME \mid (name? \notin users) \lor \\
\quad\quad (name? \in users \land name? \notin \mathrm{dom}\, borrowed \land \\
\quad\quad\quad \mathrm{dom}\, borrowed = users \setminus \{name?\}) \\
\equiv_Q \{\text{Replacement of Equals}\} \\
\quad Library, name? : NAME \mid (name? \notin users) \lor \\
\quad\quad (name? \in users \land name? \notin users \land users = users \setminus \{name?\})
\end{array}
$$

The absorption laws do not hold in QCL. Therefore, we cannot simplify the predicate to $name \notin users$. We could apply replacement to yield $name? \in users \setminus \{name?\} \land name? \notin users \land users = users \setminus \{name?\}$. This, however, does not deliver any new insight, nor is it an intuitive simplification of the above predicate. Therefore, we decided to stop the simplification process.

The quasi-classical precondition identifies both the defined and the overdefined area of applicability of an operation. The classical precondition shows only the defined area. Both together could help us to separate the overdefined area. For example, the operation *RemoveUser* is defined for the case that $name? \notin users$ and overdefined for $name? \in users \land name? \notin users \land users = users \setminus \{name?\}$. The overdefined area is of particular interest, because it contains the inconsistency. However, we have not formally determined the overdefined area yet.

Given both the standard and QC precondition we should be able to derive the condition where the operation is overdefined. Unfortunately, the following problems arise. On the one hand we cannot use classical logic, otherwise the inconsistency in the overdefined predicate would allow us to derive the predicate false. On the other hand, QCL does not have a notion of true or false, both of which could be the classical precondition. Therefore, separating the overdefined predicate formally remains an open problem for now.

Operation Consistency

In standard Z, we cannot formally distinguish between the operation *RemoveUser* and the following schema

$$
\begin{array}{|l}
\hline
RemoveUser_x \\
\Xi Library \\
name? : NAME \\
out! : Report \\
\hline
name? \notin users \land borrowed' = borrowed \land out! = failure \\
\hline
\end{array}
$$

i.e.

```
=> try RemoveUser \iff RemoveUser\_x;
=> prove by reduce;
```

Proving gives ...

$$true$$

The preconditions of both schemas *RemoveUser* and $RemoveUser_x$ are equivalent and so are their postconditions. How do we know that *RemoveUser* is actually inconsistent, apart from the fact that it does not perform the task it was designed for? How does an analyst know what the operation was meant for, apart from the informal text that should describe the meaning of the formal schemas? Standard predicate logic does not help to solve this problem satisfactorily but combining it with quasi-classical logic we hope to provide an answer.

Calculating both types of precondition of an operation could enables us to decide formally whether an operation is consistent. Given the classical and QC precondition of an operation Op we define operation consistency $cons(Op)$ as

$$\text{Operation consistency: } cons(Op) \text{ iff } \operatorname{pre} Op \equiv_{?} \operatorname{pre}_Q Op$$

i.e. an operation is consistent if both its classical and QC precondition are equivalent. Informally, we find that the operations *AddUser* and *RemoveUser* are both inconsistent but the operation $RemoveUser_x$, for example, is consistent.

There is, however, a major problem. We have not specified the equivalence relation and, therefore, we cannot compare the classical and the QC precondition. For example, the classical precondition might contain the predicates true or false which are not comparable to any predicate from QCL. Furthermore, QCL uses two different satisfaction relations. However, which of the two could be used to define an appropriate equivalence relation? This issue remains for further research.

6.5 Refinement of Inconsistent Z Specifications

So far we investigated how quasi-classical logic can support the process of reasoning about formal specifications, in particular about inconsistent specifications. The aim of a specification is to capture the essentials of a system as abstractly as possible in order to focus on the essential properties of the system. Such an abstract specification is not directly implementable, because it is not close enough to a computer program. However, we can develop a succession of more concrete specifications leading us towards an implementation. This process of development is called refinement.

(Derrick and Boiten, 2001) describe the intuition behind refinement as the

> **Principle of Substitutivity:** it is acceptable to replace one program by another, *provided* it is impossible for a user of the programs to observe that the substitution has taken place. If a program can be acceptably substituted by another, then the second program is said to be a *refinement* of the first.

To (Woodcock and Davies, 1996), refinement is all about improving specifications. It involves the removal of non-determinism, or uncertainty. Inconsistency is a form of uncertainty. Hence, refinement is also about removing inconsistency. In standard Z, the process of removing inconsistencies as a refinement step is rather unrestricted leading to some possible refinements which we intuitively reject. In this section we investigate how quasi-classical logic can be applied to the process of refining formal specifications written in the Z notation.

6.5.1 Two Refinement Examples

Earlier in this chapter we introduced a simple library specification. One of the specified operations is *AddUser* aimed at admitting a new member to the library. Unfortunately, the operation *AddUser* is inconsistent. This leaves us with a wide choice of possible refinements. For example, the following two schemas are both standard refinements of *AddUser*. We believe, however, that at least one of these two should intuitively be rejected.

$$
\begin{array}{|l}
\hline
\textit{AddUser\_R1} \\
\Delta \textit{Library} \\
\textit{name?} : \textit{NAME} \\
\hline
\textit{name?} \in \textit{users} \\
\textit{users}' = \textit{users} \setminus \{\textit{name?}\} \\
\textit{borrowed}' = \\
\qquad \{\textit{name?}\} \lhd \textit{borrowed} \\
\hline
\end{array}
\qquad
\begin{array}{|l}
\hline
\textit{AddUser\_R2} \\
\Delta \textit{Library} \\
\textit{name?} : \textit{NAME} \\
\hline
\textit{name?} \notin \textit{users} \\
\textit{users}' = \textit{users} \cup \{\textit{name?}\} \\
\textit{borrowed}' = \\
\qquad \textit{borrowed} \cup \{\textit{name?} \mapsto \varnothing\} \\
\hline
\end{array}
$$

The schema *AddUser_R1* describes the operation of removing a member from the library. This certainly was not the intention when specifying the operation *AddUser*. Therefore, we claim, that *AddUser_R1* should not be a valid refinement of *AddUser*. The operation schema *AddUser_R2* describes the operation of adding a new member to the library. It removes the inconsistency by assigning an empty set of books to the user. To us, this operation looks more like a valid refinement of *AddUser*. Below, we find out whether *AddUser_R2* can be shown to be a quasi-classical refinement of *AddUser*.

6.5.2 Classical Refinement Conditions

Before we go into detail of quasi-classical refinement, i.e. refinement using quasi-classical logic, we briefly re-cap the formal notion of standard refinement. To keep our illustration simple we choose to concentrate on the refinement of operations on the same state.

A refinement has to ensure that a more concrete operation is still applicable when the abstract operation was. Applicability relates to the preconditions of the operation. In refinement, we can weaken the precondition but not strengthen it. Furthermore, refinement needs to ensure that all the properties of the abstract specification are preserved. This means, that whenever the abstract operation was applicable in state S but the concrete operation was applied, relating the state S to an after state S', then the abstract operation also relates S to S'.

Formally, we say that an operation COp refines an operation AOp, denoted $AOp \sqsubseteq COp$, when it fulfills the following two conditions:

1. Applicability: $\mathrm{pre}\, AOp \vdash \mathrm{pre}\ COp$
2. Correctness: $\mathrm{pre}\, AOp \wedge COp \vdash AOp$

The schemas $AddUser\_R1$ as well as $AddUser\_R2$ are both standard refinements of the schema $AddUser$, i.e. $AddUser \sqsubseteq AddUser\_R1$ and $AddUser \sqsubseteq AddUser\_R2$. It is easy to see that the left-hand side of the consequence evaluates to *false* because, as we recall from the last section, $\mathrm{pre}\, AddUser = [Library, name? : NAME \mid \mathrm{false}]$. Next, we investigate a notion of quasi-classical refinement.

6.5.3 Quasi-Classical Applicability

We saw earlier that the quasi-classical precondition of an operation determines those states for which an operation is defined or overdefined. Consider the idea that overdefined is a special case of being defined. Then we can use the standard notion of applicability, i.e. that the concrete operation must be defined or overdefined on those states where the abstract operation was defined or overdefined on. We express this notion formally by

Quasi-Classical Applicability: $\mathrm{pre}_Q\, AOp \vdash_Q \mathrm{pre}_Q\ COp$

This notion of quasi-classical applicability allows weakening of QC preconditions but not strengthening.

Example

Given quasi-classical applicability, we are now able to show that the operation schema $\mathit{AddUser\_R1}$ is not a valid refinement of the schema $\mathit{AddUser}$. First, we need the quasi-classical precondition of $\mathit{AddUser\_R1}$. This is the same as the classical precondition, i.e.

$$\mathrm{pre}_Q\, \mathit{AddUser\_R1} = [\mathit{Library}, \mathit{name?} : \mathit{NAME} \mid \mathit{name?} \in \mathit{users}]$$

Then, we show that the proof tree for $\mathrm{pre}_Q\, \mathit{AddUser} \vdash_Q \mathrm{pre}_Q\, \mathit{AddUser\_R1}$ does not close:

$$\begin{array}{c}
\mathit{users} = \mathrm{dom}\, \mathit{borrowed}, \mathit{name?} \notin \mathit{users}, \\
\mathit{name?} \notin \{\mathit{name?}\}, \mathit{users} = \mathit{users} \cup \{\mathit{name?}\}, \\
(\mathit{users} = \mathrm{dom}\, \mathit{borrowed} \land \mathit{name?} \in \mathit{users})^* \\
\begin{array}{cc}
(\mathit{users} = \mathrm{dom}\, \mathit{borrowed})^* & (\mathit{name?} \in \mathit{users})^* \\
| & | \\
\text{closed} & \text{not possible to close}
\end{array}
\end{array}$$

Hence, $\mathrm{pre}_Q\, \mathit{AddUser} \nvdash_Q \mathrm{pre}_Q\, \mathit{AddUser\_R1}$ and, therefore, $\mathit{AddUser\_R1}$ is not a valid refinement. The operation schema $\mathit{AddUser\_R2}$, however, is quasi-classically applicable. The quasi-classical precondition of $\mathit{AddUser\_R2}$ is

$$\mathrm{pre}_Q\, \mathit{AddUser\_R2} = [\mathit{Library}, \mathit{name?} : \mathit{NAME} \mid \mathit{name?} \notin \mathit{users}]$$

and the proof for $\mathrm{pre}_Q\, \mathit{AddUser} \vdash_Q \mathrm{pre}_Q\, \mathit{AddUser\_R2}$ succeeds:

$$\begin{array}{c}
\mathit{users} = \mathrm{dom}\, \mathit{borrowed}, \mathit{name?} \notin \mathit{users}, \\
\mathit{name?} \notin \{\mathit{name?}\}, \mathit{users} = \mathit{users} \cup \{\mathit{name?}\}, \\
(\mathit{users} = \mathrm{dom}\, \mathit{borrowed} \land \mathit{name?} \notin \mathit{users})^* \\
\begin{array}{cc}
(\mathit{users} = \mathrm{dom}\, \mathit{borrowed})^* & (\mathit{name?} \notin \mathit{users})^* \\
| & | \\
\text{closed} & \text{closed}
\end{array}
\end{array}$$

Properties of QC Applicability

QC applicability extends the standard notion of applicability. It is sound with respect to standard applicability because whenever QC applicability holds, standard applicability must hold, too. This follows directly from the properties of QCL. Consequently, QC applicability fails if a consistent operation was made inconsistent, as this is not permitted by standard applicability either.

The converse, however, is not true. For example, consider the operations $\mathit{AddUser\_R1}$ and $\mathit{AddUser\_R2}$ both standard refinements of $\mathit{AddUser}$. Therefore, standard applicability holds for both operations but we showed that QC applicability failed for $AddUser\_R1$. The question is when does QC applicability reject a refinement that is valid according to the standard notion, i.e. could QC applicability be too restrictive?

We do not think that QC applicability is too restrictive, i.e. it does not reject any refinement of consistent operations that standard applicability would accept. Unfortunately, we lack meta-theoretical results about QCL to verify this formally. QC applicability does not validate all refinements of inconsistent operations, because inconsistency arises from overdefinedness, i.e. an inconsistent predicate provides too much information. As in standard refinement this information needs to be incorporated into the concrete operation.

6.5.4 Quasi-Classical Correctness

Once we established applicability we need to verify the correctness of an operation. A concrete operation behaves correctly with respect to an abstract operation if an observer cannot distinguish the outcome of the concrete operation and abstract operation, provided they are both applied on the same domain. We introduced the formal definition of standard correctness earlier.

Establishing QC Correctness using the Classical Law and QC Inference

QC applicability is very similar to standard applicability. Basically, we changed the inference relation to use QC entailment rather than standard entailment. This effects also the notion of a precondition which we discussed separately. It seems natural to investigate the impact of using a similar method for deriving QC correctness, i.e. to change the inference system. We define

$$\text{Quasi-Classical Correctness: } \text{pre}_Q\, AOp \wedge COp \vdash_Q AOp$$

Unfortunately, it is not as simple as that. We introduce the following two operation schemas *AbsExample* and *ConExample* for illustrative purpose.

$$\begin{array}{|l}
\multicolumn{1}{l}{AbsExample} \\
\hline
n? : \mathbb{Z} \\
x! : \mathbb{Z} \\
\hline
n? = 1 \wedge x! = 0 \\
\hline
\end{array}
\qquad
\begin{array}{|l}
\multicolumn{1}{l}{ConExample} \\
\hline
n? : \mathbb{Z} \\
x! : \mathbb{Z} \\
\hline
(n? = 1 \wedge x! = 0) \vee \\
(n? = 2 \wedge x! = 1) \\
\hline
\end{array}$$

The schema *ConExample* is intuitively, and according to the standard refinement rules, a valid refinement of *AbsExample*, because the operation has not been changed if the given number is one. Only the precondition has been weakened to consider also the case of the number two. We are interested in QC correctness to hold for consistent operations if standard correctness holds. Therefore, we should be able to establish the QC correctness condition, i.e. we need to show

$$\text{pre}_Q\, AbsExample \wedge ConExample \vdash_Q AbsExample$$

with $\text{pre}_Q\, AbsExample = [n? : \mathbb{Z} \mid n? = 1]$. Using the tableau method we construct the following proof tree

$$\begin{array}{c}
n? = 1, \\
(n? = 1 \wedge x! = 0) \vee (n? = 2 \wedge x! = 1), \\
(n? = 1 \wedge x! = 0)^* \\
| \\
n? = 1 \vee n? = 2, n? = 1 \vee x! = 1, x! = 0 \vee n? = 2, x! = 0 \vee x! = 1
\end{array}$$

$$\begin{array}{ccc}
(n? = 1)^* & \qquad & (x! = 0)^* \\
| & & \begin{array}{cc} x! = 0 & n? = 2 \\ | & | \\ \text{closed} & \vdots \end{array} \\
\text{closed} & &
\end{array}$$

This tree remains open and the proof fails, because the branch containing $n? = 2$ cannot be closed. Alternatively, we could have chosen $x! = 0 \vee x! = 1$ but, equally, the tree could not be closed.

The Problem of the Classical Approach with respect to QCL

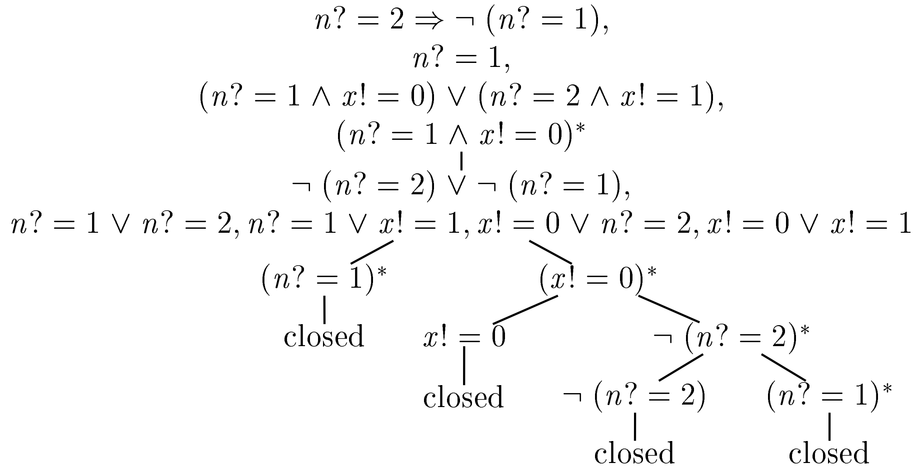
In classical logic, pre AOp restricts the applicability of COp to those cases where AOp was applicable, too. This restriction is achieved by controlled use of inconsistencies, i.e. the part of the precondition of COp that is not the precondition of AOp is reduced to false. For example,

$$\begin{aligned}
& \quad \text{pre}\, AbsExample \wedge ConExample \\
\equiv & \; \{\} \\
& \quad n? = 1 \wedge ((n? = 1 \wedge x! = 1) \vee (n? = 2 \wedge x! = 2)) \\
\equiv & \; \{\} \\
& \quad (n? = 1 \wedge n? = 1 \wedge x! = 1) \vee (n? = 1 \wedge n? = 2 \wedge x! = 2) \\
\equiv & \; \{\} \\
& \quad (n? = 1 \wedge x! = 1) \vee \text{false} \\
\equiv & \; \{\} \\
& \quad (n? = 1 \wedge x! = 1)
\end{aligned}$$

The result is the part of $ConExample$ that is applicable if the precondition of $AbsExample$ holds. This derivation used the information that $n? = 1 \wedge n? = 2$ is inconsistent. In QCL, however, we cannot use such restrictions because inconsistencies are tolerated.

Three Possible Solutions

One solution is to incorporate explicitly the information about such inconsistencies. For example, including the assumption $n? = 2 \Rightarrow \neg\,(n? = 1)$ would enable us to complete the correctness proof.



Somehow it seems not satisfactory to explicitly add side conditions to the correctness proof each time. For example, the use of automated theorem provers would be restricted. Therefore, we would prefer a more general approach to overcome the problem. We could imagine to combine the preconditions of the abstract and concrete operation such that they always provide the necessary proof conditions.

We observe, for example, that the predicate $n? = 2$ is the part of the precondition of the concrete operation that has been introduced by weakening the precondition of the abstract operation. Basically, weakening of the precondition can be expressed as a disjunction of the abstract precondition and some predicate p, i.e. $\text{pre}_Q\, COp \equiv \text{pre}_Q\, AOp \vee p$. Then, we would need to isolate the predicate p and we could add $(p)^*$ to the tree by a disjunction of p with the conclusion AOp, i.e. we derive the following correctness condition

$$\text{pre}_Q\, AOp \wedge COp \vdash_Q AOp \vee p$$

This condition expresses that we can either show AOp or the weakening of the precondition. Such an approach seems not to violate the classical correctness condition but to extend it. The problem, however, remains to extract the weakening precondition predicate p.

We could also try to generalise the idea of adding the inconsistency assumption explicitly. For example, the predicate $n? = 2 \Rightarrow \neg\,(n? = 1)$ is an implicit conjunct in the more general statement

$$
\begin{aligned}
& \mathrm{pre}_Q\ \mathit{ConExample} \Rightarrow \neg\ \mathrm{pre}_Q\ \mathit{AbsExample} \\
\equiv & \{\} \\
& (n? = 1 \lor n? = 2) \Rightarrow \neg\ (n? = 1) \\
\equiv & \{\} \\
& (\neg\ (n? = 1) \land \neg\ (n? = 2)) \lor \neg\ (n? = 1) \\
\equiv & \{\} \\
& \neg\ (n? = 1) \land (\neg\ (n? = 1) \lor \neg\ (n? = 2)) \\
\equiv & \{\} \\
& \neg\ (n? = 1) \land ((n? = 2) \Rightarrow \neg\ (n? = 1))
\end{aligned}
$$

Using that $\mathrm{pre}_Q\ COp \equiv \mathrm{pre}_Q\ AOp \lor p$, this generalises to $\neg\ \mathrm{pre}_Q\ AOp \land (p \Rightarrow \neg\ \mathrm{pre}_Q\ AOp)$. The question now is whether $\neg\ \mathrm{pre}_Q\ AOp$ can interfere with the completeness or soundness of the proof. Assuming our reasoning above is valid, the correctness proof condition for refinement would become

$$(\mathrm{pre}_Q\ COp \Rightarrow \neg\ \mathrm{pre}_Q\ AOp) \land \mathrm{pre}_Q\ AOp \land COp \vdash_Q AOp$$

It is cumbersome and, furthermore, in terms of classical logic not possible to validate. The antecedent evaluates, using classical logic, to false. In QCL, however, $\mathrm{pre}\, AOp$ and $\neg\ \mathrm{pre}\, AOp$ are two different entities. We admit that this solution seems not entirely satisfactory either. Therefore, research on the correctness condition needs to be continued.

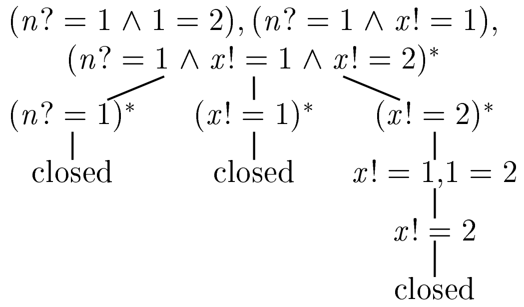
Properties of QC Correctness

Despite the problems above we try to investigate QC correctness further. We established that it is not possible to introduce inconsistencies during refinement, because applicability would fail. Removing inconsistencies by choosing one of the possible cases is, however, allowed by applicability. Now we want to find out whether this is also valid with respect to correctness.

We introduce the following simple example of two operations. The abstract operation is clearly inconsistent and the concrete one is not. Furthermore, the concrete operation is meant to return one of the possible two results from the abstract operation. We would assume that such refinement fails, because we weakened the postcondition.

$$
\begin{array}{|l}
\hline \mathit{PrimAbsExample} \\
n? : \mathbb{Z} \\
x! : \mathbb{Z} \\
\hline
n? = 1 \land x! = 1 \land x! = 2 \\
\hline
\end{array}
\qquad
\begin{array}{|l}
\hline \mathit{PrimConExample} \\
n? : \mathbb{Z} \\
x! : \mathbb{Z} \\
\hline
n? = 1 \land x! = 1 \\
\hline
\end{array}
$$

The precondition of *PrimAbsExample* is $n? = 1 \wedge 1 = 2$ and it is easy to show that QC applicability of *PrimConExample* holds. We also find that we succeed in closing the proof tree for the correctness condition of this example.

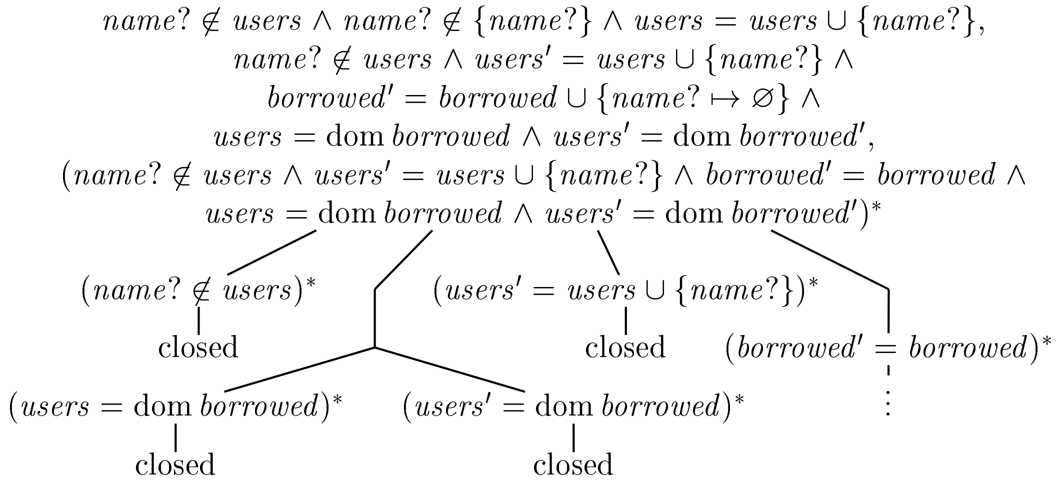


The example demonstrates that QC correctness is not enough to prevent removing inconsistencies from an operation schema. Some might consider this result as positive, although it is against the idea of refinement. Removing inconsistencies can be regarded as desired but a little alteration of the above example would show that this also allows "trivial" refinements, e.g. the operation *PrimConExample* could return $x! = 3$, which is not what we intended.

The next question is whether the removal of inconsistencies can always be verified. Unfortunately, this is not the case. In the last section, we established the applicability of the operation schema *AddUser_R2* with respect to the abstract operation *AddUser*. We are left to verify the correctness of this schema. We use the correctness condition

$$\mathrm{pre}_Q\ \mathit{AddUser} \wedge \mathit{AddUser\_R2} \vdash_Q \mathit{AddUser}$$

which results in the following proof tree



This tree cannot be closed and, therefore, the proof fails. Here, the result is as expected, because we weakened the postcondition of *AddUser* by removing the inconsistency. According to the standard rules of refinement, a postcondition can only be strengthened. In this case, QC refinement followed the idea of “living with inconsistencies” rather than eradicating them.

Both examples share the common property that an inconsistency was removed from the postcondition of the operation. However, for one example, such a refinement step is correct, for the other not. Both examples did not relate to the earlier problem of restricting the applicability of the concrete operation to those states where the abstract operation is applicable. Therefore, we have discovered another problem of QC correctness that needs to be added to the list of future work.

Correctness with respect to Invariant Properties

(Jacky, 1997, p. 247) writes: “An abstract model has some properties of the thing it models, but not all of them. A design is more concrete than a specification. A design is correct if it has all the properties of the specification; it usually has some additional properties as well.”

We need to relax this claim slightly in the presence of inconsistency. An operation is inconsistent if it contains properties that contradict each other. Insisting on the fact that a concrete specification possesses all the properties of the abstract specification would prevent us from removing inconsistencies. Therefore we rephrase the above statement: A design is correct if it has all the desired properties of the specification.

This approach involves identifying all the required properties of the operation and then to verify that the concrete operation possesses all those properties. This assumes that these selected properties have been validated with respect to the abstract operation. As such, we do not prove the correctness of refinement but the correctness of the new specification.

6.5.5 Quasi-Classical Operation Refinement

The aim of quasi-classical operation refinement is to extend the standard refinement notation by restricting the possible refinements of inconsistent operations. We introduced two examples to clarify this idea. We also introduced QC applicability and QC correctness. Now we combine all refinement rules to propose an extended notion of refinement.

We showed that QC applicability implies standard applicability. Therefore, the first refinement condition is

1. QC Applicability: $\mathrm{pre}_Q\, AOp \vdash_Q \mathrm{pre}_Q\, COp$

QC applicability, in particular, restricts the refinements to those that respect the intended application domain. Next, we need to verify that the concrete operation does at least what the abstract operation was designed for. We were not able to develop a suitable QC correctness condition. Therefore we use the standard condition which is

2. Correctness: $\mathrm{pre}\, AOp \wedge COp \vdash AOp$

The standard correctness condition still allows refinements which we would intuitively reject. As we showed earlier, it allows one to replace inconsistent outcomes of an operation by one which is not related to the contradiction. Therefore, we suggest to check those properties that the refined system should obey again, using QCL to deal with inconsistencies appropriately.

These refinement conditions extend the standard approach because QC applicability extends standard applicability. However, in QCL the transitivity of inferences fails, in particular, in the presence of inconsistencies. This implies that the above conditions do not facilitate stepwise refinement, at least with respect to applicability.

6.6 Summary

In this chapter we applied quasi-classical logic to analyse, especially inconsistent, operations specified using the Z notation. QCL proved valuable to infer properties of inconsistent operations, in particular not to infer "useless" properties.

Then we calculated the quasi-classical precondition of an operation, defined as existential quantification over the after state's variables and outputs. To simplify the precondition we had to adapt QCL and to develop a notion of logical equivalence which we presented in Chapter 4. Like in standard Z, the one-point rule plays a central role in the simplification process. It also provided a benchmark for the development of QCL with equality.

Next, we turned to QC refinement. We presented the notions of QC applicability and QC correctness. However, only QC applicability proved valuable for now, because we could not fully establish a notion of QC correctness. Therefore, QC refinement has been defined using the QC applicability and the standard correctness condition.

We have not only investigated inconsistency handling in Z using quasi-classical logic but QCL itself. QCL has only been recently developed and, therefore, only a limited amount of applications of QCL exist. Furthermore, as far as we are aware, QCL has only been used to reason about specifications written in predicate logic. The application of QCL to the process of reasoning about formal specifications written in a language richer than first-order predicate logic is new. As such, we discovered several problems that need to be addressed while developing QCL further.

Chapter 7

Un(der)definedness in Z: Guards, Preconditions and Refinement

In the common Z specification style operations are, in general, partial relations. The domains of these partial operations are traditionally called preconditions, and there are two interpretations of the result of applying an operation outside its domain. In the traditional interpretation anything may result whereas in the alternative, guarded, interpretation the operation is blocked outside its precondition.

In fact these two interpretations can be combined, and this allows representation of both refusals and underspecification in the same model. In this chapter, we explore this issue and we extend existing work in this area. To do so we adopt a non-standard three-valued interpretation of an operation by introducing a third truth value. This value corresponds to a situation where we don't care what effect the operation has, i.e. the guard holds but we may be outside the precondition.

In this chapter, we develop a schema representation based on such a three-valued interpretation. We extend in particular the work by (Fischer, 1998) by allowing arbitrary predicates in the guard. We demonstrate the advantage of this approach by means of a small example. Furthermore, we classify regions of before states based on the familiar concepts of precondition and guard. We extend these notions to the "impossible" and the "undefined" region.

Using the three-valued interpretation leads to a simple and intuitive semantics for operation refinement, where refinement means reduction of undefinedness or reduction of non-determinism. In this approach, both weakening of the precondition as well as strengthening of the guard is possible. We also show that this notion of refinement extends the standard Z refinement for both the precondition and the guarded interpretation.

7.1 Introduction

In the states-and-operations (abstract data type) specification style in Z, operations are, in general, partial relations. The domains of these partial relations are traditionally called preconditions. Depending on which context the abstract data types are used in, there are two interpretations of the result of applying an operation outside its domain.

In the traditional interpretation, presented, for example, by (Spivey, 1992), anything may happen outside the precondition, including divergence; in the blocking, also called guarded, interpretation the operation is not possible. The latter interpretation is the common one when modelling reactive systems or combining Z with process algebra, and also in Object-Z. (Strulo, 1995) calls it the 'firing condition' and (Josephs, 1991) calls it the 'enabling condition' interpretation.

It has been observed that it is often convenient to use a combination of the guarded and precondition interpretation to allow both modelling of refusals and underspecification. One way of doing this is by having explicit guards as introduced by (Abrial, 1996) in the B-Method or by (Fischer, 1998) for CSP-OZ.

7.1.1 Hypothesis

In this chapter, we generalise existing work on combining the guarded and the precondition interpretation by allowing arbitrary predicates in the guards. Furthermore, we give a model of refinement, refining both guard and precondition. We previously presented the main concepts of this work in (Miarka et al., 2000).

Our inspiration comes from a non-standard semantics of operations, viz. an interpretation in three-valued logic. The third logical truth value, denoted $\perp$, stands for the idea that we "don't care" about the outcome of an operation. We do occasionally refer to "undefinedness", although this should be distinguished from the kind of undefinedness discussed by (Valentine, 1998) and solved by VDM's third logic value. Using a three-valued logic leads to a simple and intuitive notion of (operation) refinement, where refinement is reduction of undefinedness or reduction of non-determinism (or both). It would even allow an alternative definition of refinement which preserves "required non-determinism" as discussed by (Lano et al., 1997) and (Steen et al., 1997).

However, such an interpretation of operations requires a more expressive notation than normal operations with explicit guards. In such notation, we take the operation to be false (impossible) outside its guard, and undefined where the guard holds but not the precondition. This allows us to state that, for certain before states, any after state "is undefined", but not that some after states are undefined, and others possible or impossible. We will define a syntax which is sufficiently expressive for this semantics, and define operation refinement rules for this which generalise the traditional ones.

7.1.2 Outline

The remainder of this chapter is structured as follows. In Section 7.2, we demonstrate by means of two examples, normalisation and a simple money transaction system, that a combination of the traditional and blocking interpretations is sometimes required. Then, in Section 7.3, we define a schema notation including both guards and effect schemas. Based on that we define regions of operation behaviour, i.e. whether an operation is inside or outside the guard, or inside or outside the precondition. These regions can also be related to a three-valued interpretation, which we present in Section 7.4. Using such a three-valued interpretation leads to a simple and intuitive notion of refinement that generalises standard operation refinement. We introduce the rules in Section 7.5 and show their compatibility to the standard ones. We discuss some related work in Section 7.6 and conclude with a short summary in Section 7.7. In Chapter 8 we develop a schema calculus for the guarded precondition schema notation we present here.

7.2 Guards and Preconditions in Z

The precondition of an operation characterises all the states and inputs to which the operation can be applied such that there is an after state and output which are related to the states and inputs by the operation, i.e. it characterises "before" states. However, there are two different points of view on how to interpret such a precondition. On the one hand, it can be read to be a guard, i.e. the operation will not be executed if the precondition is false. On the other hand, the interpretation may be that the operation can be executed at any time but the result of it is only guaranteed if the precondition is true. In our opinion both interpretations can coexist and sometimes should. We illustrate our point of view with the following two examples.

7.2.1 Normalisation in Z

Normalisation is the process of rewriting a schema such that all the constraint information appears in the predicate part. For example, the natural numbers are not a basic type of Z but constrained integers[1]. Therefore, a schema declaration referring to naturals can be normalised to use integers and a constraint on the predicate, e.g.

[1]This is the case in Spivey's de facto standard (Spivey, 1992); in the current draft standard (ISO/IEC 13568, 2002) even $\mathbb{Z}$ is a true subset of another type $\mathbb{A}$ ("arithmos").

$$\begin{array}{|l} \multicolumn{1}{l}{\textit{Schema}} \\ \hline a, a' : \mathbb{N} \\ \hline (a')^2 \leq a < (a'+1)^2 \\ \hline \end{array} \qquad \begin{array}{|l} \multicolumn{1}{l}{\textit{Normalised\_Schema}} \\ \hline a, a' : \mathbb{Z} \\ \hline a \in \mathbb{N} \land a' \in \mathbb{N} \land \\ \qquad (a')^2 \leq a < (a'+1)^2 \\ \hline \end{array}$$

However, somehow the interpretation may change through that process. As the operation *Schema* is defined on natural numbers, it appears unreasonable to even consider applying it on negative integers, so the blocking interpretation appears quite reasonable for this area. However, the normalised schema is formally equivalent to *Schema* but is interpreted in the precondition approach as being fully undefined on integers. This means, that the specifier needs to know about normalisation, i.e. which sets are proper types and which are proper subsets of a type, which might not always be the case and somehow should not be necessary in the first place. This example also shows that normalisation is more guard, rather than precondition, related and that we might want to deal with it accordingly.

7.2.2 A Money Transfer System

Consider the following example of a simple money transaction system. It allows to transfer a positive amount of money to a person's bank account. Therefore, we need a set of bank account holders

$$[PID]$$

Each bank account is characterised by its holder and the amount of money in it. Of course, we allow negative amounts in the account as well. On the other hand, not every person in the above set has to have a bank account, therefore, a collection of accounts is a partial function. Furthermore, *total* is a derived state component which calculates the amount of money in our bank by taking the sum of the money in all accounts.

$$\begin{array}{|l} \multicolumn{1}{l}{\textit{Bank}} \\ \hline \textit{account} : PID \rightarrow\!\!\!\!\!+ \mathbb{Z} \\ \textit{total} : \mathbb{Z} \\ \hline \textit{total} = \textit{makesum account} \\ \hline \end{array}$$

with the function

$$
\begin{array}{l}
[X] \\
\hline
makesum : (X \leftrightarrow \mathbb{Z}) \rightarrow \mathbb{Z} \\
\hline
\forall x : X;\ y : \mathbb{Z};\ z : (X \leftrightarrow \mathbb{Z}) \bullet \\
\quad makesum\ \varnothing = 0 \land \\
\quad makesum\ \{(x, y)\} = y \land \\
\quad makesum\ (\{(x, y)\} \cup z) = y + makesum\ (z \setminus \{(x, y)\})
\end{array}
$$

to calculate the total sum of all the accounts.

We describe a transaction that will transfer a given amount of money to someone's bank account. Clearly the amount transfered has to be positive, because we do not want to be able to decrease someone else's account.

$$
\begin{array}{l}
\textit{Transfer} \\
\hline
\Delta Bank \\
a? : \mathbb{Z} \\
p? : PID \\
\hline
a? \geq 0 \\
p? \in \mathrm{dom}(account) \\
account' = account \oplus \{p? \mapsto account(p?) + a?\}
\end{array}
$$

Below we analyse this small example and point out weaknesses in both the guarded and precondition interpretation.

7.2.3 Classical Precondition and Guarded Interpretation

We determine the classical precondition of the operation schema *Transfer* using Z/EVES. We find that in the above example two conditions have to be fulfilled for a transfer to be successful. On the one hand, the amount must be positive and on the other hand the receiving person must have an account.

```
=> try \pre Transfer;
=> prove by reduce;
```

Proving gives ...

$$
\begin{array}{l}
\quad account \in PID \rightarrowtail\!\!\!\!\!+ \mathbb{Z} \\
\land\ total = makesum\ account \\
\land\ a? \in \mathbb{Z} \\
\land\ p? \in PID \\
\land\ p? \in \mathrm{dom}\ account \\
\land\ a? \geq 0
\end{array}
$$

which is equal to the following schema:

$$
\begin{array}{|l}
\hline
\textit{PreTransfer} \\
\hline
\textit{Bank} \\
a? : \mathbb{Z} \\
p? : \textit{PID} \\
\hline
p? \in \operatorname{dom} \textit{account} \wedge a? \geq 0 \\
\hline
\end{array}
$$

But what happens if we try to apply the operation outside of these conditions? There are two possible interpretations: the precondition interpretation, allowing the operation, and the guarded interpretation, preventing it. A related issue is refinement, the development from a specification towards a more concrete representation. How do both interpretations deal with it?

In the standard Z interpretation a precondition represents the set of states where the operation is defined, i.e. guaranteed to produce the specified result. Outside the precondition the operation is considered to be undefined which means that the operation can do anything including non-termination ("divergence"). Therefore, refinement can, apart from reduction of non-determinism, weaken a precondition, allowing one to widen the scope of the operation and thereby reduce the area of undefinedness.

Other specification languages, like Object-Z (Smith, 2000) treat the precondition differently. There the precondition is considered as a guard, blocking the operation if the precondition is not fulfilled. Such an interpretation is occasionally used in Z as well, for example, when modelling reactive systems, as reported by (Josephs, 1991) and (Strulo, 1995). Refinement of guards is treated differently. In Object-Z, for example, one is not allowed to change the guard. Other approaches, notably the one presented by (Lano et al., 1997), where preconditions and guards are combined, allow strengthening of guards, i.e. the reduction of the applicability of the operation. They also allow to weaken any precondition. However, the precondition is the upper bound for strengthening the guard and the guard is the lower bound for weakening the preconditions.

7.2.4 Refinement

In the precondition interpretation, the following two refinements of the operation schema *Transfer* would be possible, each of them weakening one of the constraints of the precondition of *Transfer*. First, we could allow the creation of an account if the recipient of the transfer does not have one:

$$
\begin{array}{|l}
\hline
\textit{Transfer\_R1} \\
\Delta Bank \\
a? : \mathbb{Z} \\
p? : PID \\
\hline
a? \geq 0 \\
p? \notin \mathrm{dom}(account) \Rightarrow account' = account \oplus \{p? \mapsto a?\} \\
p? \in \mathrm{dom}(account) \Rightarrow account' = account \oplus \{p? \mapsto account(p?) + a?\} \\
\hline
\end{array}
$$

The given amount will be put into the newly created account. This appears a sensible refinement, however, in the guarded interpretation it would be forbidden.

The guarded interpretation rightly forbids the more dangerous refinement

$$
\begin{array}{|l}
\hline
\textit{Transfer\_R2} \\
\Delta Bank \\
a? : \mathbb{Z} \\
p? : PID \\
\hline
p? \in \mathrm{dom}(account) \\
account' = account \oplus \{p? \mapsto account(p?) + a?\} \\
\hline
\end{array}
$$

which, by removing the requirement that $a? \geq 0$, suddenly allows withdrawal of someone else's money. In the precondition interpretation this is still a valid refinement, though. We verify the applicability and correctness conditions by using Z/EVES

```
=> try \pre Transfer \implies \pre Transfer\_R2;
=> prove by reduce;
```

Proving gives ...

true

```
=> try \pre Transfer \land Transfer\_R2 \implies Transfer;
=> prove by reduce;
```

Proving gives ...

true

Apparently, the two predicates in *PreTransfer* have a different status: $a? \geq 0$ is more like a guard, whereas $p? \in \mathrm{dom}(account)$ is more like a precondition. This example shows that each interpretation alone is not always sufficient. Therefore, we propose to have both guards and preconditions in the same specification.

7.2.5 Combining Guards and Preconditions

The idea to combine guards and preconditions is not new. For example, (Fischer, 1997; Fischer, 1998) provides a solution to this problem by using an “enabled” schema to denote the guard and an “effect” schema for the standard operation schema with its precondition interpretation. Using this approach the *Transfer* operation in our example evolves to

$$
\begin{array}{l}
\textit{F\_Transfer} \\
\hline
\begin{array}{|l} \texttt{enable\_}\textit{Transfer} \\ \hline a? : \mathbb{Z} \\ \hline a? \geq 0 \\ \hline \end{array}
\qquad
\begin{array}{|l} \texttt{effect\_}\textit{Transfer} \\ \hline \Delta \textit{Bank} \\ a? : \mathbb{Z} \\ p? : \textit{PID} \\ \hline p? \in \operatorname{dom}(\textit{account}) \\ \textit{account}' = \textit{account} \oplus \\ \qquad \{p? \mapsto \textit{account}(p?) + a?\} \\ \hline \end{array} \\
\hline
\end{array}
$$

where `enable` refers to the guard of the operation and `effect` to the effect of the operation. Now the operation *F_Transfer* is blocked whenever $a?$ is negative. However, the update of someone’s account is only guaranteed if the account already exists. In case it does not, divergence may occur.

With this notation we are able to develop refinement rules which deal with the guards and preconditions in an appropriate fashion. Such refinement rules would allow one to weaken the precondition of *F_Transfer* (i.e. `effect_`*Transfer*), reduce any non-determinism in the specification, and potentially strengthen the guard (i.e. `enable_`*Transfer*). With these rules in place we are able to weaken the precondition $p? \in \operatorname{dom}(account)$ provided we do preserve the guard $a? \geq 0$.

However, according to (Fischer, 1998) the guard “must contain unprimed state variables only”. Unfortunately, this would still allow undesired refinements, as the after state is completely unconstrained for before states satisfying the guard but not the precondition. Sensible restrictions like

$$\{p?\} \lhd account' = \{p?\} \lhd account$$
and $total' = total + a?$

which express that no one else’s account changes and that the total amount of money cannot exceed the previous amount plus the newly added, cannot be imposed. Adding this restriction to `effect_`*Transfer* would have no effect, because it can be derived from `effect_`*Transfer* already. However, for states currently outside the precondition but within the guard, we have no way of imposing this as a postcondition.

7.3 The Encoding of Un(der)definedness in Z

Incorporating both guards and preconditions for operations enables a particular way of specifying un(der)definedness in Z. Basically, an operation can be blocked by the guard. However, if not blocked it leaves two possibilities, either its result is guaranteed, i.e. applied within its precondition, or its result is un(der)defined.

In this section we introduce the syntax to describe an operation in terms of guards and preconditions. We then use this characterisation to define the different regions of definition that an operation can have. The operation syntax we introduce splits an operation into two parts consisting of its guard and its effect in a way similar to that described in Section 7.2.5.

7.3.1 A Schema Representation of Un(der)definedness

An operation is defined as a triple $(Decl\_Op, gd\_Op, do\_Op)$, where $Decl$ denotes the declaration part of the operation, gd the guard of the operation and do the effect of the operation itself. It is depicted by the following schema:

Op

- $Decl\_Op$
  - $Decl$
- gd_Op
  - $pred_{gd}$
- do_Op
  - $pred_{do}$

where $Decl\_Op$ is implicitly included in gd_Op and do_Op. Note, that this is different to (Miarka et al., 2000) where we put the declaration in the gd- and do-part separately. However, this way should ease the writing of schemas by not duplicating information. Often, we use the abbreviation (gd_Op, do_Op) assuming the declaration to be included where necessary.

The following axiom ensures that the only relevant part of the do-part of the operation is that which lies within the guard, i.e. that it can only be applied if the guard is fulfilled.

Axiom 1 $\forall P, Q \bullet (P, Q) \equiv (P, P \wedge Q)$

In any formal interpretation we should ensure that this Axiom holds. Hence, we can, where necessary, restrict our considerations to specifications where the guard is included in the definition of the effect.

Theorem 7.3.1
For every schema, there is an equivalent one, such that the effect implies the guard, i.e. that the guard is embedded within the effect.

$$\forall P, Q \;\; \exists P', Q' \bullet (P, Q) \equiv (P', Q') \land Q' \Rightarrow P'$$

Proof
The required P' and Q' are given by P and $P \land Q$, respectively:

$$\begin{aligned}
& \forall P, Q \;\; \exists P', Q' \bullet (P, Q) \equiv (P', Q') \land Q' \Rightarrow P' \land \\
& \qquad\qquad\qquad\qquad\qquad\qquad P' = P \land Q' = P \land Q \\
\equiv\ & \{\text{One-point rule (twice)}\} \\
& \forall P, Q \;\; \bullet (P, Q) \equiv (P, P \land Q) \land (P \land Q) \Rightarrow P \\
\equiv\ & \{\text{Predicate Calculus}\} \\
& \forall P, Q \;\; \bullet (P, Q) \equiv (P, P \land Q)
\end{aligned}$$

which is valid by Axiom 1. □

Theorem 7.3.2
A formally weaker version of (P, Q) is obtained by replacing Q with $P \Rightarrow Q$, i.e.

$$\forall P, Q \bullet (P, Q) \equiv (P, P \Rightarrow Q)$$

Proof

$$\begin{aligned}
& (P, Q) \\
\equiv\ & \{\text{Axiom 1}\} \\
& (P, P \land Q) \\
\equiv\ & \{\text{Predicate Calculus}\} \\
& (P, P \land (P \Rightarrow Q)) \\
\equiv\ & \{\text{Axiom 1}\} \\
& (P, P \Rightarrow Q)
\end{aligned}$$

□

7.3.2 Normalisation revisited

Given the above schema notation for expressing guards and preconditions we can express normalisation differently.

Normalised_Schema

Decl_Normalised_Schema

$a, a' : \mathbb{Z}$

gd_Normalised_Schema

$a \in \mathbb{N} \land a' \in \mathbb{N}$

do_Normalised_Schema

$(a')^2 \leq a < (a' + 1)^2$

Now the operation is blocked, if a is not a natural number, which is probably more like the intended interpretation of $a : \mathbb{N}$. Following this intuition, we define normalisation of guarded precondition schemas such that any type constraints which are implicit have to be made explicit and will become part of the guard.

7.3.3 The Money Transfer System revisited

The previously discussed operation *Transfer* with the desired extension of the guard can now be expressed as

$$
\begin{array}{l}
\textit{Transfer2} \\
\hline
\quad \begin{array}{|l}
\textit{Decl\_Transfer2} \\
\hline
\Delta \textit{Bank} \\
a? : \mathbb{Z} \\
p? : \textit{PID} \\
\hline
\end{array} \\
\quad \begin{array}{|l}
\textit{gd\_Transfer2} \\
\hline
a? \geq 0 \\
\textit{total}' = \textit{total} + a? \\
\{p?\} \lhd \textit{account}' = \\
\qquad \{p?\} \lhd \textit{account} \\
\hline
\end{array}
\quad \begin{array}{|l}
\textit{do\_Transfer2} \\
\hline
p? \in \mathrm{dom}(\textit{account}) \\
\textit{account}' = \textit{account} \oplus \\
\qquad \{p? \mapsto \textit{account}(p?) + a?\} \\
\hline
\end{array} \\
\hline
\end{array}
$$

Having primed state variables in the guard causes the guard not to be executable, because we cannot test the after state beforehand. However, we may consider specifications that contain undefined areas as not implementable anyway, because some refinement is still missing. For refinement rules which remove undefinedness see Section 7.5. Primed state variables in the guard do not limit implementations in general, they just give us more expressiveness.

7.3.4 Regions of Before States

Using such a notation of guarded precondition schema, we can describe (at least) three different possibilities for a particular pair of before/after states:

1. *gd_Op* holds and *do_Op* holds: the states belong to the operation.
2. *gd_Op* holds but *do_Op* does not hold: the states may or may not belong to the operation, we don't care.
3. *gd_Op* does not hold: we do not wish the states to belong to the operation. (Note, that this makes *do_Op* for this pair of states redundant information.)

Based on this description, we can define a number of regions of before states that are of interest.

Impossible. The impossible region is the set of states where the operation is blocked, i.e. it is always going to fail.

$$\text{impo}(Op) \mathrel{\widehat{=}} [S, ins? \mid \neg \exists S', outs! \bullet gd\_Op]$$

Analysing our example, we identify that the operation *Transfer*2 is always rejected when the amount $a?$ is negative, i.e.

$$\text{impo}(Transfer2) = [Bank, a? : \mathbb{Z}, p? : PID \mid a? < 0].$$

Precondition. The precondition region is the area where the operation is possible and well defined. It is defined by

$$\text{pre}(Op) \mathrel{\widehat{=}} [S, ins? \mid \exists S', outs! \bullet gd\_Op \wedge do\_Op]$$

Observe that this is consistent with our convention of Op denoting $gd\_Op \wedge do\_Op$. Then this results in the following precondition for our example:

$$\text{pre}(Transfer2) = [Bank, a? : \mathbb{Z}, p? : PID \mid p? \in \text{dom}(account) \wedge a? \geq 0].$$

Guard. The guarded region is simply the complement to the impossible region, i.e. it is the area where the blocking predicate holds.

$$\text{guard}(Op) \mathrel{\widehat{=}} [S, ins? \mid \exists S', outs! \bullet gd\_Op]$$

This, however, is the same as calculating the precondition of the guarded part of the operation, i.e. $\text{guard}(Op) = \text{pre}(gd\_Op)$. Then it holds for our example

$$\text{guard}(Transfer2) = \text{pre}(gd\_Transfer2) = [Bank, a? : \mathbb{Z}, p? : PID \mid a? \geq 0].$$

Here it is clear that our approach is strictly more expressive than Fischer's: $\text{guard}(Op)$ contains an abstraction of the information in our approach, whereas in his $pre(\texttt{enable}) = \texttt{enable}$. In *Transfer*2 the guard is $a? \geq 0$, loosing the information that any widening of the precondition should respect $\{p?\} \lhd account' = \{p?\} \lhd account$ and $total' = total + a?$.

Undefined. Given the regions defined by guard and precondition we could define the "completely undefined" region as the difference between guard and precondition. This would be

$$\text{undef}(Op) \mathrel{\widehat{=}} [S, ins? \mid \exists S', outs! \bullet gd\_Op \wedge (\neg \exists S' \bullet gd\_Op \wedge do\_Op)]$$

In the initial *Transfer* operation it is

$$\text{undef}(Transfer) = [Bank, a? : \mathbb{Z}, p? : PID \mid a? \geq 0 \wedge p? \notin \text{dom}(account)]$$

whereas in *Transfer*2 this region is empty.

In the next chapter, we develop formally a schema calculus for guarded precondition schemas. We introduce existential quantification and review our work on calculating the precondition, the guard, and the other regions.

7.4 Three Valued Interpretation

In the last section we defined several regions according to pairs of before/after states. We distinguished three different possibilities: First, the region where *gd_Op* does not hold, i.e. where the operation should be impossible. Second, the region where both *gd_Op* and *do_Op* hold, i.e. where after states belong to the operation. Third, the remaining region where *gd_Op* holds but *do_Op* does not hold. In that case the outcome of the operation is undefined. These three regions are depicted in Figure 7.1 and can be naturally described using a set of three truth values $\{\mathbf{f}, \mathbf{t}, \perp\}$ respectively.

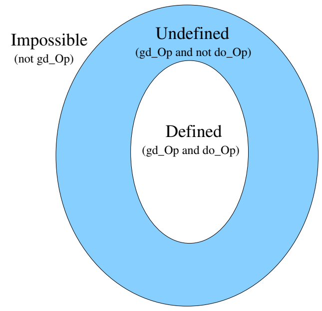


Figure 7.1: Combining Guard and Precondition

7.4.1 Semantical Description of the Regions

Formally, we define the transition from pairs of schemas to a three-valued logic via a mapping function *val* that returns the appropriate truth value related to the schema. Given a boolean-like type

$$bool3 ::= \mathbf{t} \mid \mathbf{f} \mid \perp$$

we define the three-valued interpretation of an operation $Op = (P, Q)$ on state S as follows:

$$\begin{aligned}\mathrm{val}(Op) = &\{x;\ y \mid (x, y) \in \mathrm{rel}(P \wedge Q) \bullet (x, y) \mapsto \mathbf{t}\} \\ &\cup \{x;\ y \mid (x, y) \notin \mathrm{rel}(P) \bullet (x, y) \mapsto \mathbf{f}\} \\ &\cup \{x;\ y \mid (x, y) \in \mathrm{rel}(P \wedge \neg Q) \bullet (x, y) \mapsto \perp\}\end{aligned}$$

where $\mathrm{rel}(Op)$ is a binary relation between bindings of type S and bindings of type S', i.e. $\mathrm{rel}(Op) = \{Op \bullet \theta S \mapsto \theta S'\}$.

We show that the given Axiom 1 also holds in this three-valued interpretation. This is the case, if and only if it maps to the same truth values in either case of using pairs (P, Q) or $(P, P \wedge Q)$.

Proof sketch:

$$\mathrm{val}(P, Q) = \left\{ \begin{array}{lllcll} \mathbf{t} & \text{iff} & P \wedge Q & = & P \wedge (P \wedge Q) & \text{iff } \mathbf{t} \\ \mathbf{f} & \text{iff} & \neg\, P & = & \neg\, P & \text{iff } \mathbf{f} \\ \perp & \text{iff} & P \wedge \neg\, Q & = & P \wedge \neg\, (P \wedge Q) & \text{iff} \perp \end{array} \right\} = \mathrm{val}(P, P \wedge Q)$$

7.4.2 Depicting Before and After States

We use a table style notation to depict the relation of before states and after states of an operation by means of the possible outcome, i.e. by val(Op). For example, given an operation

Filter

> *Decl_Filter*
> $a? : \mathbb{Z}$
> $b! : \mathbb{Z}$
>
> *gd_Filter*
> $a? > 0$
>
> *do_Filter*
> $isEven(a?) \wedge b! \leq a?$

which takes only a positive number as input and returns any number less or equal to it if the given number is even. Then the table representation is

$a?\ ^{b!}$	...	-1	0	1	2	3	4	5	...
⋮									
-1		**f**	**f**	**f**	**f**	**f**	**f**	**f**	
0		**f**	**f**	**f**	**f**	**f**	**f**	**f**	
1		$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	
2		**t**	**t**	**t**	**t**	**f**	**f**	**f**	
3		$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	
4		**t**	**t**	**t**	**t**	**t**	**t**	**f**	
5		$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	
⋮									

Table 7.1: Before and After States Relations

7.4.3 Meaning of Refinement

Operation refinement is defined as removal of undefinedness as well as non-determinism. Taking our three-valued interpretation and the above representation, we can explain refinement intuitively as replacing multiple $\perp$ in a row by

t provided it enlarges the precondition region or by replacing any $\perp$ by **f** which in turn may reduce the guarded region. Furthermore, we can replace multiple **t** in a line by **f** (as long as one **t** remains) to reduce non-determinism. Note, the latter step does not change either the precondition nor the guarded region.

We consider the *Filter* operation from above to clarify the presented notion of refinement. Therefore, we introduce a possible refinement *C_Filter*.

C_Filter

Decl_C_Filter

$a? : \mathbb{Z}$
$b! : \mathbb{Z}$

gd_C_Filter

$a? > 0$
$b! < a?$

do_C_Filter

$isEven(a?)$
$b! = a?/2$

The following refinement took place. First, we ensure that $b!$ is always less than $a?$. This is done by strengthening the guard and corresponds to changing $\perp$ to **f** for all cases where $b! \geq a?$. Note, that this refinement step also strengthens the postcondition of *Filter* in some cases. Second, we remove non-determinism by providing a more concrete representation of the output in case that $a?$ is even. This is done by replacing multiple **t** by **f**. Weakening of the precondition did not take place but we may define an output for the case that $a?$ is an odd number in another refinement step. However, the result will always be bound by the newly introduced predicate in the guard. The outcome of this refinement step is illustrated in the following table.

$a?^{b!}$	…	-1	0	1	2	3	4	5	…
⋮									
-1		**f**	**f**	**f**	**f**	**f**	**f**	**f**	
0		**f**	**f**	**f**	**f**	**f**	**f**	**f**	
1		$\perp$	$\perp$	**f**	**f**	**f**	**f**	**f**	
2		**f**	**f**	**t**	**f**	**f**	**f**	**f**	
3		$\perp$	$\perp$	$\perp$	$\perp$	**f**	**f**	**f**	
4		**f**	**f**	**f**	**t**	**f**	**f**	**f**	
5		$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	**f**	
⋮									

Table 7.2: Before and After States Relations after Refinement

7.5 Operation Refinement

In this work, we restrict ourselves to operation refinement. Our work is intended to generalise the standard approach of refinement. In this section, we first present our generalised rules of refinement which we then apply to the *Transfer* example. Finally, we show that our new refinement conditions indeed generalise both the guarded and the preconditioned approach.

7.5.1 Rules for Operation Refinement

Given an abstract operation $AOp = (gd\_AOp, do\_AOp)$ and a concrete operation $COp = (gd\_COp, do\_COp)$ both over the same state *State* with input $x? : X$ and output $y! : Y$, then COp refines AOp, denoted $AOp \sqsubseteq COp$, if and only if applicability (1) and correctness (2) hold:

$$(1)\ \forall State;\ x? : X \bullet \mathrm{pre}\, AOp \vdash \mathrm{pre}\, COp$$
$$(2)\ \forall State;\ State';\ x? : X;\ y! : Y \bullet \mathrm{pre}\, AOp \land COp \vdash AOp$$

The first condition allows to weaken the precondition and the second condition ensures that the refined operation does at least what the abstract operation did.

Additionally, we allow strengthening of guards but not weakening:

$$(3)\ \forall State;\ State';\ x? : X;\ y! : Y \bullet gd\_COp \vdash gd\_AOp$$

Conditions (1) and (3) together ensure that the precondition is the upper bound for strengthening the guard and that the guard is the lower bound for weakening the precondition.

We observe that the correctness rule can be formally weakened using (3):

$$\begin{aligned}
& \mathrm{pre}\, AOp \land COp \Rightarrow AOp \\
\equiv\ & \{\text{Definition of Op}\} \\
& \mathrm{pre}(gd_AOp \land do_AOp) \land gd_COp \land do_COp \Rightarrow gd_AOp \land do_AOp \\
\equiv\ & \{\text{Using } gd_COp \Rightarrow gd_AOp\} \\
& \mathrm{pre}(gd_AOp \land do_AOp) \land gd_COp \land do_COp \Rightarrow do_AOp \\
\equiv\ & \{\text{Definition of Op}\} \\
& \mathrm{pre}\, AOp \land COp \Rightarrow do_AOp
\end{aligned}$$

However, it turns out nicely that the shape of the standard refinement rules is preserved when we use the introduced abbreviation.

7.5.2 Refinement of the Money Transfer System

We introduced in Section 7.2.2 a simple money transaction system that allows to put money into the account of an existing customer. We showed via an example that using only the guarded or precondition interpretation limits the expressiveness, and also perhaps allows unintended refinement. In our combined approach we solved these problems. Therefore, we are now able to express the following refinement of the *Transfer*2 operation:

$C\_Transfer2$

$Decl\_C\_Transfer2$
$$\Delta Bank$$
$$a? : \mathbb{Z}$$
$$p? : \mathbb{Z}$$

$gd\_C\_Transfer2$
$$a? \geq 0$$
$$total' = total + a?$$
$$\{p?\} \ntriangleleft account' = \{p?\} \ntriangleleft account$$

$do\_C\_Transfer2$
$$p? \notin \mathrm{dom}(account) \Rightarrow account' = account \oplus \{p? \mapsto a?\}$$
$$p? \in \mathrm{dom}(account) \Rightarrow account' = account \oplus \{p? \mapsto account(p?) + a?\}$$

First, we strengthened the guard *gd_Transfer*2. Now, the money to be transfered has to be positive and we are not permitted to change another person's bank account, no matter what future refinement will do to the precondition. Second, we also refined the *do_Transfer*2 operation. We weakened the precondition of *Transfer*2 to handle the case that the receiving user does not have an account. In this case we allow the creation of a new bank account which will have the amount $a?$ as initial input.

7.5.3 Generalisation of Traditional Refinement Rules

Our concept of refinement is a valid generalisation of the traditional operation refinement rules in both the guarded and the preconditioned approach. Taking $gd\_Op = \mathrm{pre}\, Op$ and $do\_Op = Op$ or $gd\_Op = \mathrm{true}$ and $do\_Op = Op$, respectively, we show that our refinement rules reduce to the traditional ones.

Guarded Approach

In the guarded interpretation the guard is the precondition of the operation. Therefore, we use $gd\_Op = \mathrm{pre}\, Op$ and $do\_Op = Op$.

Let $Op_1 = (gd\_Op_1, do\_Op_1) = (\text{pre}\,AOp, AOp)$ and $Op_2 = (gd\_Op_2, do\_Op_2) = (\text{pre}\,COp, COp)$. We show that for this choice of Op_1, Op_2 it holds $Op_1 \sqsubseteq Op_2 \equiv AOp \sqsubseteq COp$ in the guarded approach.

(1) Applicability.

$$\begin{aligned}
&\text{pre}\,Op_1 \vdash \text{pre}\,Op_2\\
\equiv\ &\{Op = (gd_Op \wedge do_Op)\}\\
&\text{pre}(gd_AOp \wedge do_AOp) \vdash \text{pre}(gd_COp \wedge do_COp)\\
\equiv\ &\{gd_Op = \text{pre}\,Op \text{ and } do_Op = Op\}\\
&\text{pre}(\text{pre}\,AOp \wedge AOp) \vdash \text{pre}(\text{pre}\,COp \wedge COp)\\
\equiv\ &\{\text{Simplification: pre}\,Op \wedge Op \equiv Op\}\\
&\text{pre}\,AOp \vdash \text{pre}\,COp
\end{aligned}$$

(2) Correctness.

$$\begin{aligned}
&\text{pre}\,Op_1 \wedge Op_2 \vdash Op_1\\
\equiv\ &\{Op = (gd_Op \wedge do_Op)\}\\
&\quad \text{pre}(gd_AOp \wedge do_AOp) \wedge (gd_COp \wedge do_COp)\\
&\vdash\ (gd_AOp \wedge do_AOp)\\
\equiv\ &\{gd_Op = \text{pre}\,Op \text{ and } do_Op = Op\}\\
&\text{pre}(\text{pre}\,AOp \wedge AOp) \wedge (\text{pre}\,COp \wedge COp) \vdash (\text{pre}\,AOp \wedge AOp)\\
\equiv\ &\{\text{Simplification: pre}\,Op \wedge Op \equiv Op\}\\
&\text{pre}\,AOp \wedge COp \vdash AOp
\end{aligned}$$

(3) Strengthening.

$$\begin{aligned}
&gd_Op_2 \vdash gd_Op_1\\
\equiv\ &\{gd_Op_1 = \text{pre}\,AOp,\ gd_Op_2 = \text{pre}\,COp\}\\
&\text{pre}\,COp \vdash \text{pre}\,AOp
\end{aligned}$$

Applicability and strengthening together result in the fact that $\text{pre}\,COp = \text{pre}\,AOp$, i.e. the standard condition in Object-Z that a guard cannot be strengthened nor weakened. The correctness rule is as in standard refinement as well.

Precondition Approach

To show that our approach is a generalisation of the precondition approach, we consider that the guard of the operation is the weakest possible, i.e. $gd\_Op = \text{true}$.

Then our notation coincides with the standard Z notation, where $do\_Op = Op$. Using the fact that we consider $Op = gd\_Op \wedge do\_Op$ it is easy to show that applicability (1) and correctness (2) hold. The rule for strengthening (3) evaluates to $\forall\, State;\ State';\ x? : X;\ y! : Y \bullet$ true which means there is no strengthening at all. Therefore, in the case of no guards our refinement rules are equivalent to the standard ones.

7.5.4 Refinement Rules for Required Non-Determinism

A different interpretation is possible for the operations in three-valued logic that we have described. Various authors, like (Lano et al., 1997) and (Steen et al., 1997) have argued that for behavioural specifications, the traditional identification of non-determinism with implementation freedom is unsatisfactory. They would like the opportunity to specify *required* non-determinism, which implies a need for additional specification operators to express implementation freedom. Refinement rules should then remove implementation freedom but not non-determinism. (Steen et al., 1997) describe such a calculus, obtained by adding a disjunction operator to LOTOS.

We could introduce a similar calculus for Z by reinterpreting the three-valued operations described above. As before, when the operation evaluates to **f** for a particular before and after state, it denotes an impossibility. However, the collection of after states that are related by **t** to a particular before state represents *required* non-determinism. As a consequence, none of these **t** values may be removed in refinement. Finally, the collection of after states that are related by $\perp$ to a particular before state represent an implementation choice, i.e. at least one of those after states will need to be related by **t** in a final refinement.

As a consequence, expressed in terms of the tabular representation used before, refinement rules for required non-determinism and disjunctive specification are:

- if a line contains a single $\perp$, it is equivalent to **t** (required choice from a singleton set);
- if a line contains multiple occurrences of $\perp$, some but not all of them may be changed to **f** (reducing possibility of choice);
- any $\perp$ may be changed to **t** (in particular, an implementation choice between several after states may be refined to a non-deterministic choice between some of them).

This approach generalises only the guarded approach – the precondition just characterises those before states for which possible after states have been determined already. It also prevents some undesired interaction between removing undefinedness and increasing determinism.

7.6 Related Work

Dealing with un(der)definedness in Z explicitly has been an issue for a while. It came up when researchers, like (Josephs, 1991), tried to use Z for specifying concurrent systems and it became apparent that one might need guards and preconditions together.

7.6.1 Strulo's Work on Firing Conditions

(Strulo, 1995) attempts to unify both the precondition and the guarded interpretation to model passive and active behaviour in Z accordingly. He developed the idea of classifying a schema according to its function and to use at one time the guarded interpretation and another time the precondition interpretation. Note, that Strulo uses the term firing condition rather than guard.

An operation is described by a single state schema, plus a label indicating whether the operation is either active or passive. A distinction is made between active operations being impossible or divergent, by interpreting before states which allow all possible after states as divergent. This encoding extends the guarded approach to preconditions in Z. Refinement in Strulo's work is subtle as "the conditions for refinement depend on the identification of active and passive behaviours".

The characterisation of an "unconstrained" operation, whose predicate interpretation is universally true, as divergent is somewhat artificial. For example, given an operation over a singleton state, the classification into unconstrained and interesting region contradict. An operation over a singleton state is either *true* or *false*, but not one or the other at some time, i.e. there is no interesting region but only an empty or an unconstrained. However, such an operation is clearly not divergent, so it should not be in the unconstrained area but in the interesting region. This is a contradiction, showing that Strulo's classification is not always sufficient.

7.6.2 The (R, A)-Calculus by Doornbos

The (R, A)-calculus by (Doornbos, 1994) separates well-definedness of an operation from its effect, in an abstract setting of binary relations and sets. An operation (R, A) consists of a set A essentially representing its precondition, and a relation R specifying its effect. This is substantially different from having a relation with an explicit *guard*, in particular it allows the specification of "miracles". The fragment of the calculus satisfying $A \subseteq \mathrm{dom}\, R$ (i.e., the "law" of the excluded miracle), is generalised by our calculus, viz. $(gd\_Op, do\_Op) \mathrel{\widehat{=}} (R, A \lhd R)$. Doornbos also draws a parallel between the (R, A) calculus and weakest (liberal) preconditions which suggests a similar exercise would be possible for our calculus.

7.6.3 Hehner and Hoare's Predicative Approach to Programming

In (Hehner, 1993; Hehner, 1999; Hoare and He Jifeng, 1998) the authors consider a specification to be a predicate of the form $P \Rightarrow Q$ meaning that if P is satisfied, then the computation terminates and satisfies Q. A specification S is refined by a specification T if all computations satisfying T also satisfy S, i.e. the reverse implication $S \Leftarrow T$ ($T \sqsubseteq S$). This allows weakening of the precondition P as well as strenghtening of the postcondition Q.

Within this approach, the predicate $guard \wedge (pre \Rightarrow post)$ in a schema body would express nearly the desired effect under the guarding interpretation of Z schemas. In this interpretation, a false *guard* causes the specification to be false, i.e. impossible, and a false precondition *pre* leads to the specification being true, which in turn allows any output.

However, the advantage of our approach with two schemas *gd* and *do* is a certain independence of the guard and precondition. Even when the precondition is false, not every output is permitted: it is still restricted by the guard.

7.7 Summary

In this chapter we presented the idea of using a three-valued interpretation of operations to combine and extend the guarded and precondition approaches. Using this non-standard interpretation we were able to present a simple and intuitive notion of operation refinement, which generalizes the traditional refinement relations.

A full theory of refinement would also include a notion of data refinement. However, when the retrieve relation is a two-valued predicate the extension becomes obvious. It remains, however, unclear what might be represented by a three-valued retrieve relation.

In our interpretation of pairs of schemas (gd_Op, do_Op) we identified only three regions. Clearly, we could further distinguish the areas $\neg\, gd\_Op \wedge \neg\, do\_Op$ and $\neg\, gd\_Op \wedge do\_Op$. The latter area might be regarded as representing "miracles" or inconsistency. Detecting and managing inconsistency between the guarded and the preconditioned region is another of our topics for future research, possibly based on the work presented in Chapter 6.

Chapter 8

A Schema Calculus for Un(der)definedness in Z

In the states-and-operations style in the specification language Z, un(der)definedness is not normally made explicit. However, in the last chapter we showed that it is possible to adapt Z schemas such that both guards and preconditions are represented at the same time, and thus enabling the specification of un(der)definedness. We call such schemas guarded precondition schemas.

Schemas are central building blocks in standard Z and it is possible to perform a variety of operations with and on them. In the last chapter, we presented the semantics for guarded precondition schemas based on a non-standard three-valued interpretation of an operation. We introduced a third truth value to correspond to a situation where we don't care what effect the operation has. In this chapter, we use this three-valued interpretation to develop a schema calculus for guarded precondition schemas.

Our approach is based on three-valued truth tables for the common logical operators, i.e. negation, conjunction, disjunction, and entailment. These truth tables guide the development process of the corresponding schema operators. We demonstrate the validity of the definitions by proving several laws from standard predicate logic. However, we also find that some laws do not hold. This is not surprising as we do not deal with two- but three-valued logic. Furthermore, we find that entailment and implication are decoupled but schema entailment can be defined using standard implication.

Schema quantification is also an important part of standard Z. We, too, present a notion of schema quantification for guarded precondition schema and apply it to schema hiding, projection, composition and precondition calculation. Given this calculus we revise the regions of operation applicability, as introduced in the last chapter. We also revise operation refinement using the new schema calculus.

8.1 Introduction

Schemas are central building blocks in standard Z and it is possible to perform a variety of operations with and on them. In the last chapter, we presented the semantics for guarded precondition schemas based on a non-standard three-valued interpretation of an operation. We introduced a third truth value to correspond to a situation where we don't care what effect the operation has. In this chapter, we use this three-valued interpretation to develop a schema calculus for guarded precondition schemas.

8.1.1 Motivation

In Chapter 7 we introduced a new schema representation to combine both guarded and precondition interpretation of Z schemas. We demonstrated the use of our notation by means of an example. We introduced the concepts of the regions of applicability of an operation and operation refinement rules for such guarded precondition schemas. However, we did not present mechanisms to combine such schemas.

The schema calculus is used to structure and compose descriptions. This allows to divide up the information content of a specification into manageable pieces. In particular, this enables re-usability of common components. Of course, while developing a new schema representation we do not want to loose the advantages of the standard Z notation, i.e. we need a schema calculus for guarded precondition schemas as well.

In standard Z, the existential quantification over the after states and output variables of an operation schema enables the calculation of the precondition of that operation. The result is a schema containing the predicate that needs to hold to guarantee the outcome of an operation. Furthermore, quantification, schema implication and schema conjunction are used in standard Z to formalise the notion of refinement. Surely, we want to be able to perform precondition calculation and refinement, too.

(Fischer, 1998) introduced a schema notation based on `enable` and `effect` schemas to capture guards and preconditions. His research was aimed at combining Object-Z and CSP specifications. While it inspired our work in guarded precondition schemas it did not provide a schema calculus.

(Strulo, 1995), too, works on unifying both the precondition and the guarded interpretation. His aim is to model passive and active behaviour in Z accordingly. Strulo decided to classify a schema according to its function and to use at one time the guarded interpretation and another time the precondition interpretation. To combine and to reason about schemas he uses the schema calculus from standard Z. Obviously this is not possible in our approach as we develop a new schema representation which is more expressive than Strulo's.

8.1.2 Hypothesis

In the last this chapter, we developed our new schema representation based on a three-valued interpretation. We propose to extend this work by using the same interpretation to develop a schema calculus for guarded precondition schemas. We show that it is possible to define the schema operators based on the given valuation function, mapping the schema representation to three distinct truth values, and three-valued truth tables. We then extend the calculus to enable quantification of schemas variables.

By developing this calculus we demonstrate that our guarded precondition schemas can be used to construct complex specifications. We already introduced the regions of applicability of an operation. The schema operators can be used to formally determine these regions. Also, the schema calculus is sufficient to enable the specifier to verify the refinement of an abstract operation by a concrete operation.

8.1.3 Outline

Here, we develop a schema calculus for guarded precondition schemas. We briefly re-cap the notion of a guarded precondition schema in Section 8.2 and we illustrate its use by presenting an example of a heat control system. We present the main part of this chapter in Section 8.3 which consists of the development of the schema calculus itself. Based on the standard schema operators, we introduce the schema operators for the guarded precondition schemas. We also prove several laws for the schema operators, to validate the correctness of our definitions. Furthermore, we show that some laws of two-valued predicate logic do not hold within our calculus. Next, in Section 8.5, we revise the notions of schema applicability and, finally, in Section 8.6 we look at operation refinement again, using the newly developed schema calculus.

8.2 Un(der)definedness in Z: Guarded Precondition Schemas

Incorporating both guards and preconditions for operations enables a particular way of specifying un(der)definedness in Z. Basically, an operation can be blocked by the guard. However, if not blocked it leaves two possibilities, either its result is guaranteed, i.e. applied within its precondition, or its result is un(der)defined.

8.2.1 A Schema Representation of Un(der)definedness

An operation is defined as a triple $(Decl\_Op, gd\_Op, do\_Op)$, where $Decl$ denotes the declaration part of the operation, gd the guard of the operation and do the effect of the operation itself. It is depicted by the following schema:

Op

- $Decl\_Op$
  - $Decl$
- gd_Op
  - $pred_{gd}$
- do_Op
  - $pred_{do}$

where $Decl\_Op$ is implicitly included in gd_Op and do_Op. Often, we will use the abbreviation (gd_Op, do_Op) assuming the declaration to be included where necessary.

8.2.2 Example: A Heat Control System

Here we give an example specification using guarded precondition schemas to illustrate the concept and use of guarded precondition schemas. We develop a heat control system which turns a fan on or off according to a given temperature. The fan has to run when the temperature is above a maximum and it is off when the temperature is below a given minimum. However, between these two boundaries it can be on or off. We define a boolean like type $FanType$ to capture the two possible operation modes of a fan.

$$FanType ::= On \mid Off$$

The state of our system is only concerned about the status of the fan.

$System$

$fan : FanType$

We use an axiomatic definition for the temperature range such that both values constrain the entire specification.

$$heat\_max, heat\_min : \mathbb{Z}$$

$$heat\_max = 65$$
$$heat\_min = 45$$

The maximum temperature is set to be 65 degrees Celsius and the minimum temperature to be 45 degrees. These are average values for the operation of some computer processor fans. Initially, the fan will be on, for safety reasons.

InitFan

$System'$

$fan' = On$

The fan can be turned on if the current temperature, given by the input *heat*?, is above the minimum temperature and if the fan is not running. However, it must be turned on if the temperature is above the maximum allowed temperature.

On

Decl

$\Delta System$

$heat? : \mathbb{Z}$

gd_On

$heat? > heat\_min$

$fan = Off$

do_On

$heat? \geq heat\_max$

$fan' = On$

The *Off* operation is specified similar to the *On* operation, being allowed if the temperature is below maximum but being certainly applied if it is below minimum.

Off

Decl

$\Delta System$

$heat? : \mathbb{Z}$

gd_Off

$heat? < heat\_max$

$fan = On$

do_Off

$heat? \leq heat\_min$

$fan' = Off$

8.2.3 Schemas using *true* and *false*.

In this chapter, we use two special schemas, denoted TRUE and FALSE. The schema TRUE can always be applied and the outcome of its application is unconstrained. Therefore, its representation is given by the pair $(true, true)$. Contrary to TRUE, the schema FALSE is never applicable, i.e. it is always blocked, hence its guard is *false*. According to Axiom 1, the *do*-part is irrelevant in such cases. However, for practical use we define it to be *false*, too, i.e. FALSE = $(false, false)$.

Schemas using *false* in the guard are not applicable, they do not allow any operation and cannot be weakened in refinement. However, they may come in handy to add constraints to the *do*-part using schema disjunction. If the *do*-part is *false* but not the guard then it is possible to perform an operation though no outcome is defined. However, during refinement this operation may become defined.

The schema $(gd\_Op, true)$ is mostly used to add constraints to the guard via schema conjunction. Otherwise, any outcome is possible as long as the guard permits the operation. Finally, we turn to the schema $(true, do\_Op)$. Due to the guard being *true*, such an operation is always applicable, i.e. it is never blocked, though its result can be both undefined or well-defined. However, this is the same situation that occurs in standard Z with the precondition interpretation. Therefore, it is possible to embed standard Z schemas into guarded precondition schemas using the following three steps: first, move its declarations into the declaration part, second, let the guard be *true* and, third, let the *do*-part be equivalent to the predicate of the standard schema, i.e.

$$S \mathrel{\widehat{=}} \left[\, Decl \mid pred \,\right] \equiv S = (Decl, true, pred)$$

8.3 A Schema Calculus for Guarded Precondition Schemas

In this section, we develop a schema calculus for the guarded precondition schemas. We consider the main Z schema operators: negation, conjunction, disjunction, quantification, hiding, projection, and sequential composition. An overview of the standard definitions of these operators can be found in Chapter 2 as well as in (Woodcock and Davies, 1996) and (Potter et al., 1991).

We also show that this calculus obeys several laws of predicate logic. This is necessary since we are not dealing with standard predicate logic anymore but with an encoding of three-valued logic with two two-valued predicates. This is illustrated by the fact that some classical laws, like the law of excluded middle, do not hold. However, we are able to use two-valued predicate logic and its laws whenever we are dealing with classical predicates, which are contained in both the *gd*- and *do*-part of an operation.

8.3.1 Three-Valued Truth Tables

In the previous section we introduced a schema representation that allows the specification of un(der)definedness in Z. We used two predicates, one, the guard, to describe that the operation is permitted and another one, the *do*-part to describe that the operation is also defined. Both together capture that the operation is well-defined. We are also able to express that the operation is possible but not defined, i.e. it is undefined. Finally, the negated guard is used to express that the operation is forbidden, i.e. impossible. These three cases can be described using a set of three truth values $\{\mathbf{t}, \perp, \mathbf{f}\}$ respectively, where $\perp$ is often called "bottom".

We defined the transition from pairs of schemas to a three-valued logic via a mapping function *val* that returns the appropriate truth value relating to the schema. Given a boolean-like type

$$bool3 ::= \mathbf{t} \mid \mathbf{f} \mid \perp$$

we also defined the three-valued interpretation of an operation $Op = (P, Q)$ as follows:

$$\begin{aligned}\mathrm{val}(Op) = &\{x;\ y \mid (x, y) \in \mathrm{rel}(P \wedge Q) \bullet (x, y) \mapsto \mathbf{t}\} \\ &\cup \{x;\ y \mid (x, y) \notin \mathrm{rel}(P) \bullet (x, y) \mapsto \mathbf{f}\} \\ &\cup \{x;\ y \mid (x, y) \in \mathrm{rel}(P \wedge \neg Q) \bullet (x, y) \mapsto \perp\}\end{aligned}$$

where $\mathrm{rel}(Op) = \{Op \bullet \theta State \mapsto \theta State'\}$.

Given the three truth values we introduce the following truth tables which are the three-valued fragment with $\perp$ of (Damásio and Pereira, 1998), as well as those derived from (Herre and Pearce, 1992). These tables define the propositional fragment of a logic we need for this work:

p	¬ p
t	**f**
f	**t**
$\perp$	$\perp$

p∧q	**t**	**f**	$\perp$
t	**t**	**f**	$\perp$
f	**f**	**f**	**f**
$\perp$	$\perp$	**f**	$\perp$

p∨q	**t**	**f**	$\perp$
t	**t**	**t**	**t**
f	**t**	**f**	$\perp$
$\perp$	**t**	$\perp$	$\perp$

p→q	**t**	**f**	$\perp$
t	**t**	**f**	**f**
f	**t**	**t**	**t**
$\perp$	**t**	**t**	**t**

Table 8.1: Three-Valued Truth Tables

These truth tables will guide us in the development of the schema calculus, i.e. the schema operators will be defined with respect to these three-valued connectives.

8.3.2 Schema Inclusion and Schema Decoration

Both schema inclusion and schema decoration follow the standard Z conventions. They allow us to hide some details of a schema and to focus on the relationship of the relevant variables, leaving implicit the invariant properties of a system. Of course, those properties can be made explicit again by expanding the schemas.

Schema Inclusion. Schema inclusion is one of the simplest schema operations. It allows to use the name of a schema amongst the declarations of another schema. Like in standard Z, the effect of inclusion of a schema U amongst the declarations of a schema V is that the declarations of U are included in those of V, the predicates of the guard of U are included in the guard of V and the predicates of the *do*-part of U are appended to the *do*-part of V. Note, that no type clashed must occur if the schemas are fully expanded. For example, the schemas *On* and *Off* of the heat control system that we specified earlier include the schemas *System* and *System'*.

Schema Decoration. The rules of schema decoration are similar to those in standard Z. In particular, the use of primed schema names follows the standard convention, i.e. the effect is that the decoration is applied to all the variables in the declaration of the decorated schema both within the declaration and the predicate parts of the schema. For example, within the schema *InitFan*, the schema *System* is decorated with a prime and, therefore, the same applies to the variable *fan*.

A further notational convention is the use of Δ and Ξ in front of a schema name. For any schema U, ΔU is defined as

$$\Delta U \mathrel{\widehat{=}} [U;\ U']$$

i.e. it contains all the variables and predicates declared in the schema U together with another set of primed declarations and predicates corresponding to the definitions in the schema U.

Sometimes, an operation does not cause any change of a particular state U but the operation requires some information provided by that state. Then we use ΞU, as defined by

$$\Xi U \mathrel{\widehat{=}} [\Delta U \mid \theta U = \theta U']$$

to express that no changes to the variables declared in U occur.

8.3.3 Schema Negation

Negation is a function that changes truth but not knowledge. Since we are dealing with undefinedness the latter is important. An operation can be forbidden, undefined, or well-defined. According to the truth table for negation, the negation of forbidden is well-defined, and vice versa, leaving the undefined region to remain undefined. Since the guard being *false* defines the forbidden, or impossible, region, it has to define the well-defined region after negation. It is just the opposite for the defined area, i.e. it should become impossible. Therefore, whenever there was an effect, it will be forbidden after negation. This intuition leads to the following definition of schema negation.

We define schema negation to preserve the declaration of the operation but to swap and to negate its predicates, i.e. given the schema U

U

$Decl\_U$

$a : A$
$b : B$

gd_U	do_U
P	Q

The negation $W = \neg\ U$ is:

W

$Decl\_W$

$a : A$
$b : B$

gd_W	do_W
$\neg\ (P \wedge Q)$	$\neg\ P$

The appearance of P in the new guard can be explained from Axiom 1. However, if originally $Q \Rightarrow P$, then the predicate in the guard is equivalent to $\neg\ Q$, in which case negation can be written as $\neg\ (P, Q) = (\neg\ Q, \neg\ P)$.

To derive the above definition of negation we used the following reasoning process: if $\neg\ P$ stands for "the operation is not applicable", i.e. *false* then it should be *true* after negation, i.e. it should be applicable and defined, hence $\neg\ P$ in the *do*-part. Furthermore, an operation is defined if $P \wedge Q$ holds, which should in turn become *false* after negation, i.e. operations inside it should be blocked, or in other words, operations outside it should be allowed, hence $\neg\ (P \wedge Q)$ in the guard.

However, we need to assume that the given schema U is normalised. As in standard Z, a syntactic form of a schema negation can only be given on the assumption that the negated schema was normalised first. Thus, we assume that schemas are normalised whenever schema negation is applied in any inference in the following sections and subsections.

The Double Negation Law. The first law we show to hold is the double negation law, i.e. that $\neg\neg\, U = U$ holds:

$$\begin{aligned}
&\neg\neg\, U\\
\equiv\ &\{\text{Definition of } U\}\\
&\neg\neg\,(P, Q)\\
\equiv\ &\{\text{Schema Negation}\}\\
&\neg\,(\neg\,(P \wedge Q), \neg\, P)\\
\equiv\ &\{\text{Schema Negation}\}\\
&(\neg\,(\neg\,(P \wedge Q) \wedge \neg\, P), \neg\neg\,(P \wedge Q))\\
\equiv\ &\{\text{Classical de Morgan Law}\}\\
&(\neg\neg\,((P \wedge Q) \vee P), \neg\neg\,(P \wedge Q))\\
\equiv\ &\{\text{Classical Double Negation (twice)}\}\\
&((P \wedge Q) \vee P, P \wedge Q)\\
\equiv\ &\{\text{Classical Absorption Law}\}\\
&(P, P \wedge Q)\\
\equiv\ &\{\text{Axiom 1, Definition of } U\}\\
&U
\end{aligned}$$

We can simplify schema negation and subsequently the proof of the double negation law when $Q \Rightarrow P$ holds:

$$\begin{aligned}
&\neg\neg\, U\\
\equiv\ &\{\text{Definition of } U\}\\
&\neg\neg\,(P, Q)\\
\equiv\ &\{\text{Schema Negation}\}\\
&\neg\,(\neg\, Q, \neg\, P)\\
\equiv\ &\{\text{Schema Negation}\}\\
&(\neg\neg\, P, \neg\neg\, Q)\\
\equiv\ &\{\text{Classical Double Negation Law}\}\\
&(P, Q)\\
\equiv\ &\{\text{Definition of } U\}\\
&U
\end{aligned}$$

It may seem trivial but nevertheless, the following law is rather useful in many proofs: $\neg\, \mathit{TRUE} = \mathit{FALSE}$

$$\neg\, \mathit{TRUE}$$

$$
\begin{aligned}
&\equiv \{\text{Definition of TRUE}\}\\
&\quad \neg\,(\text{true}, \text{true})\\
&\equiv \{\text{Schema Negation}\}\\
&\quad (\text{false}, \text{false})\\
&\equiv \{\text{Definition of FALSE}\}\\
&\quad \mathit{FALSE}
\end{aligned}
$$

8.3.4 Schema Conjunction

According to the truth table of three-valued logic, conjunction is *true* if both its arguments are *true*, i.e. the conjunction of two schemas should be in the defined region in case both schemas are in their defined region, too. Furthermore, conjunction results in *false*, i.e. the operation is outside the guard, if either of the schemas involved in the conjunction is outside their guards, i.e. it is *true* if both are inside the guard.

For example, given:

U
- $Decl\_U$: $a : A$, $b : B$
- gd_U: P
- do_U: Q

V
- $Decl\_V$: $a : A$, $d : D$
- gd_V: S
- do_V: T

then their conjunction is given by the schema $W = U \wedge V$:

W
- $Decl\_W$: $a : A$, $b : B$, $d : D$
- gd_W: $P \wedge S$
- do_W: $P \wedge Q \wedge S \wedge T$

Following from the Axiom 1 the following simplification is always correct:

W

$Decl\_W$

$a : A$
$b : B$
$d : D$

gd_W

$P \wedge S$

do_W

$Q \wedge T$

Note, for every variable declared in both schemas we define its common type to be the intersection of both given types. Thus, just like for standard Z schema conjunction, names declared in both schemas from incompatible sets will lead to a type error.

Here, and throughout this chapter, we prove several laws for working with schema operators. We already showed that the double negation law holds. Now we turn to some principal laws for schema conjunction.

Idempotent Law for Conjunction: $U \wedge U = U$

Applying conjunction to two identical schemas results in nothing but the schema itself.

$$\begin{aligned}
& U \wedge U \\
\equiv\ & \{\text{Definition of } U\} \\
& (P, Q) \wedge (P, Q) \\
\equiv\ & \{\text{Schema Conjunction, Axiom 1}\} \\
& (P \wedge P, Q \wedge Q) \\
\equiv\ & \{\text{Classical Idempotency of Conjunction}\} \\
& (P, Q) \\
\equiv\ & \{\text{Definition of } U\} \\
& U
\end{aligned}$$

Zero Law for Conjunction: $U \wedge \text{FALSE} = \text{FALSE}$

Using the schema FALSE as an argument for schema conjunction results in the schema FALSE.

$$\begin{aligned}
& U \wedge \text{FALSE} \\
\equiv\ & \{\text{Definition of } U \text{ and FALSE}\} \\
& (P, Q) \wedge (\text{false}, \text{false})
\end{aligned}$$

$$
\begin{aligned}
\equiv\ & \{\text{Schema Conjunction}\}\\
& (P \wedge \text{false}, P \wedge Q \wedge \text{false} \wedge \mathit{false})\\
\equiv\ & \{\text{Classical Zero Law for Conjunction}\}\\
& (\text{false}, \text{false})\\
\equiv\ & \{\text{Definition of FALSE}\}\\
& \text{FALSE}
\end{aligned}
$$

One Law for Conjunction: $U \wedge \text{TRUE} = U$

Complementing the Zero Law is the One Law. A conjunction between a schema U and the schema TRUE is the same as the schema U itself.

$$
\begin{aligned}
& U \wedge \text{TRUE}\\
\equiv\ & \{\text{Definition of } U \text{ and TRUE}\}\\
& (P, Q) \wedge (\text{true}, \text{true})\\
\equiv\ & \{\text{Schema Conjunction, Axiom 1}\}\\
& (P \wedge \text{true}, Q \wedge \text{true})\\
\equiv\ & \{\text{Classical One Law for Conjunction}\}\\
& (P, Q)\\
\equiv\ & \{\text{Definition of } U\}\\
& U
\end{aligned}
$$

All three laws are compression laws, in the sense, that given two schemas their application results in one schema. On the other hand, there are also laws that allow the arguments of schema conjunction to be swapped as well as to change bracketing of conjuncts, i.e. the commutativity and associativity laws. However, they also follow from the commutativity and associativity of the classical conjunction operator. Therefore, we do not prove them in detail.

8.3.5 Schema Disjunction

Basically, there are two possible ways to define schema disjunction. Firstly, it can be done by applying a similar reasoning process as in defining schema conjunction, i.e. inferring it from the three-valued truth table. Secondly, schema disjunction could be calculated using schema conjunction and negation. We will follow the first path and show that the same result would be obtained by using the second method, i.e. we show that the de Morgan laws hold.

For disjunction to be *true*, i.e. defined, either of its arguments must be defined, i.e. $gd \wedge do$ must hold. Disjunction is *false*, i.e. outside its guard, if both schemas are outside their guards, hence it is inside the guard, if either one schema is inside its guard. Therefore, given the schemas U and V from above, their disjunction $W = U \vee V$ is given as:

W

$Decl\_W$

$a : A$
$b : B$
$d : D$

gd_W

$P \vee S$

do_W

$(P \wedge Q) \vee (S \wedge T)$

The type of variables declared in both schemas is the union of the types given to that variable in each of the schemas.

Like for schema conjunction we show idempotency, the Zero Law, as well as the One Law for schema disjunction to hold. Again, we do not prove commutativity and associativity but their proofs are based on both properties holding for classical disjunction.

Idempotent Law for Disjunction: $U \vee U = U$

$$\begin{aligned}
& U \vee U \\
\equiv\ & \{\text{Definition of } U\} \\
& (P, Q) \vee (P, Q) \\
\equiv\ & \{\text{Schema Disjunction}\} \\
& (P \vee P, (P \wedge Q) \vee (P \wedge Q)) \\
\equiv\ & \{\text{Idempotency of Classical Disjunction}\} \\
& (P, P \wedge Q) \\
\equiv\ & \{\text{Axiom 1, Definition of } U\} \\
& U
\end{aligned}$$

Zero Law for Disjunction: $U \vee \text{FALSE} = U$

$$\begin{aligned}
& U \vee \text{FALSE} \\
\equiv\ & \{\text{Definition of } U \text{ and FALSE}\} \\
& (P, Q) \vee (\text{false}, \text{false}) \\
\equiv\ & \{\text{Schema Disjunction}\} \\
& (P \vee \text{false}, (P \wedge Q) \vee (\text{false} \wedge \text{false}))
\end{aligned}$$

$\equiv$ {Classical Zero Law for Conjunction}
$(P \vee \text{false}, (P \wedge Q) \vee \text{false})$
$\equiv$ {Classical Zero Law for Disjunction}
$(P, P \wedge Q)$
$\equiv$ {Axiom 1, Definition of U}
U

One Law for Disjunction: $U \vee \text{TRUE} = \text{TRUE}$

$U \vee \text{TRUE}$
$\equiv$ {Definition of U and TRUE}
$(P, Q) \vee (\text{true}, \text{true})$
$\equiv$ {Schema Disjunction}
$(P \vee \text{true}, (P \wedge Q) \vee (\text{true} \wedge \text{true}))$
$\equiv$ {Classical One Law for Conjunction as well as Disjunction}
$(\text{true}, \text{true})$
$\equiv$ {Definition of TRUE}
TRUE

De Morgan Laws. Classically, disjunction can be defined in terms of negation and conjunction as $(U \vee V) = \neg (\neg U \wedge \neg V)$. For our definition of disjunction to be useful we require it to obey the de Morgan laws, too.

Given the two normalised schemas U and V, then

$\neg (\neg U \wedge \neg V)$
$\equiv$ {Definition of U and V}
$\neg (\neg (P, Q) \wedge \neg (S, T))$
$\equiv$ {Schema Negation}
$\neg ((\neg (P \wedge Q), \neg P) \wedge (\neg (S \wedge T), \neg S))$
$\equiv$ {Schema Conjunction}
$\neg (\neg (P \wedge Q) \wedge \neg (S \wedge T), \neg P \wedge \neg (P \wedge Q) \wedge \neg S \wedge \neg (S \wedge T))$
$\equiv$ {Classical de Morgan Law, Absorption Law}
$\neg (\neg ((P \wedge Q) \vee (S \wedge T)), \neg (P \vee S))$
$\equiv$ {Schema Negation}
$(\neg (\neg ((P \wedge Q) \vee (S \wedge T)) \wedge \neg (P \vee S)), \neg \neg ((P \wedge Q) \vee (S \wedge T)))$
$\equiv$ {Classical de Morgan Law, Classical Double Negation (twice)}
$((P \wedge Q) \vee (S \wedge T) \vee P \vee S, (P \wedge Q) \vee (S \wedge T))$

$$
\begin{aligned}
&\equiv \{\text{Classical Commutativity, Classical Absorption Law}\}\\
&\quad (P \vee S, (P \wedge Q) \vee (S \wedge T))\\
&\equiv \{\text{Definition of Schema Disjunction}\}\\
&\quad (P, Q) \vee (S, T)\\
&\equiv \{\text{Definition of } U \text{ and } V\}\\
&\quad U \vee V
\end{aligned}
$$

Similarly, it can be shown that conjunction can be defined in terms of disjunction and negation. If the given schemas are not normalised then normalisation needs to be added to the above derivation before applying negation, since, as mentioned earlier, it is a necessary condition for schema negation.

Distribution Laws: $U \vee (V \wedge W) = (U \vee V) \wedge (U \vee W)$

The first of the two distribution laws states that disjunction distributes over conjunction:

$$
\begin{aligned}
&\quad U \vee (V \wedge W)\\
&\equiv \{\text{Definition of } U,\ V, \text{ and } W\}\\
&\quad (P, Q) \vee ((S, T) \wedge (X, Y))\\
&\equiv \{\text{Schema Conjunction}\}\\
&\quad (P, Q) \vee (S \wedge X, (S \wedge T) \wedge (X \wedge Y))\\
&\equiv \{\text{Schema Disjunction}\}\\
&\quad (P \vee (S \wedge X), (P \wedge Q) \vee ((S \wedge T) \wedge (X \wedge Y))\\
&\equiv \{\text{Classical Distribution Law for Disjunction}\}\\
&\quad ((P \vee S) \wedge (P \vee X), ((P \wedge Q) \vee (S \wedge T)) \wedge ((P \wedge Q) \vee (X \wedge Y)))\\
&\equiv \{\text{Schema Conjunction}\}\\
&\quad (P \vee S, (P \wedge Q) \vee (S \wedge T)) \wedge (P \vee X, (P \wedge Q) \vee (X \wedge Y))\\
&\equiv \{\text{Schema Disjunction}\}\\
&\quad ((P, Q) \vee (S, T)) \wedge ((P, Q) \vee (X, Y))\\
&\equiv \{\text{Definition of } U,\ V, \text{ and } W\}\\
&\quad (U \vee V) \wedge (U \vee W)
\end{aligned}
$$

Similarly, conjunction distributes over disjunction:

$$U \wedge (V \vee W) = (U \wedge V) \vee (U \wedge W)$$

$$
\begin{aligned}
&\quad U \wedge (V \vee W)\\
&\equiv \{\text{Definition of } U,\ V, \text{ and } W\}
\end{aligned}
$$

$$
\begin{aligned}
& \quad (P, Q) \wedge ((S, T) \vee (X, Y)) \\
\equiv & \ \{\text{Schema Disjunction}\} \\
& \quad (P, Q) \wedge (S \vee X, (S \wedge T) \vee (X \wedge Y)) \\
\equiv & \ \{\text{Schema Conjunction}\} \\
& \quad (P \wedge (S \vee X), (P \wedge Q) \wedge ((S \wedge T) \vee (X \wedge Y))) \\
\equiv & \ \{\text{Classical Distribution Law for Conjunction}\} \\
& \quad ((P \wedge S) \vee (P \wedge X), ((P \wedge Q) \wedge (S \wedge T)) \vee ((P \wedge Q) \wedge (X \wedge Y))) \\
\equiv & \ \{\text{Schema Disjunction}\} \\
& \quad (P \wedge S, (P \wedge Q) \wedge (S \wedge T)) \vee (P \wedge X, (P \wedge Q) \wedge (X \wedge Y)) \\
\equiv & \ \{\text{Schema Conjunction}\} \\
& \quad ((P, Q) \wedge (S, T)) \vee ((P, Q) \wedge (X, Y)) \\
\equiv & \ \{\text{Definition of } U,\ V, \text{ and } W\} \\
& \quad (U \wedge V) \vee (U \wedge W)
\end{aligned}
$$

Absorption Laws: $U \vee (U \wedge V) = U$

Another set of useful laws often used to simplify proofs are the two absorption laws provided here:

$$
\begin{aligned}
& \quad U \vee (U \wedge V) \\
\equiv & \ \{\text{Definition of } U \text{ and } V\} \\
& \quad (P, Q) \vee ((P, Q) \wedge (S, T)) \\
\equiv & \ \{\text{Schema Conjunction}\} \\
& \quad (P, Q) \vee (P \wedge S, (P \wedge Q) \wedge (S \wedge T)) \\
\equiv & \ \{\text{Schema Disjunction}\} \\
& \quad (P \vee (P \wedge S), (P \wedge Q) \vee ((P \wedge Q) \wedge (S \wedge T)) \\
\equiv & \ \{\text{Classical Absorption Law}\} \\
& \quad (P, P \wedge Q) \\
\equiv & \ \{\text{Axiom 1, Definition of } U\} \\
& \quad U
\end{aligned}
$$

Dual to the above is the following law: $U \wedge (U \vee V) = U$

$$
\begin{aligned}
& \quad U \wedge (U \vee V) \\
\equiv & \ \{\text{Definition of } U \text{ and } V\} \\
& \quad (P, Q) \wedge ((P, Q) \vee (S, T)) \\
\equiv & \ \{\text{Schema Disjunction}\} \\
& \quad (P, Q) \wedge (P \vee S, (P \wedge Q) \vee (S \wedge T))
\end{aligned}
$$

$\equiv$ {Schema Conjunction}
$\quad (P \wedge (P \vee S), (P \wedge Q) \wedge ((P \wedge Q) \vee (S \wedge T))$
$\equiv$ {Classical Absorption Law}
$\quad (P, P \wedge Q)$
$\equiv$ {Axiom 1, Definition of U}
$\quad U$

So far, we proved that many laws related to negation, conjunction and disjunction known from classical logic also hold for the newly developed schema representation and, therefore, that they can be used in the schema calculus. We now turn to define quantification and to investigate its laws.

8.3.6 Schema Quantification

Both definitions of universal and existential quantification are analogous to standard Z, i.e. the quantified variable is going to be removed from the schema declaration and will be quantified in the predicate.

Universal Quantification. Since the declaration of the schema is implicitly included in both the *gd*- and the *do*-part of the schema, quantification has to be applied in both sub-schemas as well. Therefore, we define:

$\forall\, a : A \bullet U =$

W
- $Decl\_W$
  - $b : B$
- gd_W
  - $\forall\, a : A \bullet P$
- do_W
  - $\forall\, a : A \bullet (P \wedge Q)$

Existential Quantification. In a similar fashion we define existential quantification, by:

$\exists\, a : A \bullet U =$

W

$Decl\_W$

$b : B$

gd_W

$\exists\, a : A \bullet P$

do_W

$\exists\, a : A \bullet (P \wedge Q)$

Following these definitions, we show that idempotency of quantification, as well as the de Morgan laws hold, too. Later, we apply schema quantification to the notions of hiding and precondition calculation.

Idempotency of Quantification: $\mathcal{Q}a : A \bullet \mathcal{Q}a : A \bullet U = \mathcal{Q}a : A \bullet U$

Quantifying over a bound variable with the same quantifier results in the same schema already provided.

$$
\begin{aligned}
& \mathcal{Q}a : A \bullet \mathcal{Q}a : A \bullet U \\
\equiv\ & \{\text{Definition of } U\} \\
& \mathcal{Q}a : A \bullet \mathcal{Q}a : A \bullet (P, Q) \\
\equiv\ & \{\text{Schema Quantification}\} \\
& \mathcal{Q}a : A \bullet (\mathcal{Q}a : A \bullet P, \mathcal{Q}a : A \bullet (P \wedge Q)) \\
\equiv\ & \{\text{Schema Quantification}\} \\
& (\mathcal{Q}a : A \bullet \mathcal{Q}a : A \bullet P, \mathcal{Q}a : A \bullet \mathcal{Q}a : A \bullet (P \wedge Q)) \\
\equiv\ & \{\text{Idempotency of Classical Quantification}\} \\
& (\mathcal{Q}a : A \bullet P, \mathcal{Q}a : A \bullet (P \wedge Q)) \\
\equiv\ & \{\text{Definition of Schema Quantification}\} \\
& \mathcal{Q}a : A \bullet (P, Q) \\
\equiv\ & \{\text{Definition of } U\} \\
& \mathcal{Q}a : A \bullet U
\end{aligned}
$$

where $\mathcal{Q}$ is either one of the quantifiers $\forall$ or $\exists$.

De Morgan Laws for Quantification. Of course, the quantification operators should respect the de Morgan rules for quantification known from classical logic, and they do, as we show for two cases:

Given the normalised schema U then we show

$$\exists\, a : A \bullet U = \neg\ \forall\, a : A \bullet \neg\ U$$

by the following derivation:

$$
\begin{aligned}
& \neg \forall a : A \bullet \neg U \\
\equiv\ & \{\text{Definition of } U\} \\
& \neg \forall a : A \bullet \neg (P, Q) \\
\equiv\ & \{\text{Schema Negation}\} \\
& \neg \forall a : A \bullet (\neg (P \wedge Q), \neg P) \\
\equiv\ & \{\text{Schema Generalisation}\} \\
& \neg (\forall a : A \bullet \neg (P \wedge Q), \forall a : A \bullet \neg P) \\
\equiv\ & \{\text{Schema Negation}\} \\
& (\neg (\forall a : A \bullet \neg (P \wedge Q) \wedge \forall a : A \bullet \neg P), \neg \forall a : A \bullet \neg (P \wedge Q)) \\
\equiv\ & \{\text{Simplification and Classical de Morgan Law for Quantification}\} \\
& (\neg \forall a : A \bullet (\neg (P \wedge Q) \wedge \neg P), \exists a : A \bullet (P \wedge Q)) \\
\equiv\ & \{\text{Classical de Morgan Law}\} \\
& (\neg \forall a : A \bullet \neg ((P \wedge Q) \vee P), \exists a : A \bullet (P \wedge Q)) \\
\equiv\ & \{\text{Classical de Morgan Law for Quantification}\} \\
& (\exists a : A \bullet (P \wedge Q) \vee P, \exists a : A \bullet (P \wedge Q)) \\
\equiv\ & \{\text{Classical Absorption Law}\} \\
& (\exists a : A \bullet P, \exists a : A \bullet (P \wedge Q)) \\
\equiv\ & \{\text{Schema Particularisation}\} \\
& \exists a : A \bullet (P, Q) \\
\equiv\ & \{\text{Definition of } U\} \\
& \exists a : A \bullet U
\end{aligned}
$$

Given the normalised schema U, we also show that

$$\exists a : A \bullet \neg U = \neg \forall a : A \bullet U$$

by this short inference:

$$
\begin{aligned}
& \exists a : A \bullet \neg U \\
\equiv\ & \{\text{Previous Proof}\} \\
& \neg \forall a : A \bullet \neg \neg U \\
\equiv\ & \{\text{Double Negation Law}\} \\
& \neg \forall a : A \bullet U
\end{aligned}
$$

The above laws show how existential quantification and universal quantification as well as negation are related. Similarly, it can be shown that $\neg \exists a : A \bullet U = \forall a : A \bullet \neg U$ and $\neg \exists a : A \bullet \neg U = \forall a : A \bullet U$ hold, too. In the above proof, we assumed U to be normalised. If U is not normalised, we will have to add normalisation to the proofs, since schema negation is only well-defined for normalised schemas.

8.3.7 Schema Hiding and Projection

Schema hiding and projection are another set of operators. In both cases, the purpose is to localise, i.e. to hide, components, either given as a set or through another schema.

Schema Hiding. Normally, hiding of the variables $(x_1, \ldots, x_n)$ from a schema S, $S \setminus (x_1, \ldots, x_n)$, is defined using existential quantification, i.e. $(\exists\, x_1 : t_1; \ \ldots; \ x_n : t_n \bullet S)$. Therefore, we use the existential quantification introduced above. For example, hiding a from the schema U results in the following: $U \setminus (a) = \exists\, a : A \bullet U$, which is represented by the schema W:

W

Decl_W

$b : B$

gd_W

$\exists\, a : A \bullet P$

do_W

$\exists\, a : A \bullet P \land Q$

Schema Projection. The schema projection operator $U \upharpoonright V$ combines two schemas using conjunction but hiding all components of U except those that are part of V. Both schemas must be type compatible, but V might have extra components not shared by U. Therefore, the resulting schema has the signature of V. Formally, $U \upharpoonright V = (U \land V) \setminus (x_1, \ldots, x_n)$, where $x_1, \ldots, x_n$ are the components of U not shared by V. Therefore, the schema $W = U \upharpoonright V$ is

W

Decl_W

$a : A$
$d : D$

gd_W

$\exists\, c : C \bullet P \land S$

do_W

$\exists\, c : C \bullet (P \land S) \land (Q \land T)$

8.3.8 Schema Composition

One way of combining schemas is to use logical schema operators as introduced before, another is to use composition, i.e. to state that one operation is to be applied after another operation. To use sequential composition it is necessary to ensure that the schema declarations are compatible.

Renaming. In standard Z, schema components can be renamed. Renaming is also called substitution and allows to introduce a different collection of variables with the same pattern of declarations and constraints. We, too, provide a definition of renaming for our calculus. Substituting variables is denoted by $[newvar/oldvar]$ where the old variables will be renamed to new variables. However, renaming will only be applied if the new name is not already present in the schema and only for any free occurrence of the old name. For example, $U[x/a, y/b]$ results in the schema W

W

$Decl\_W$
$$x : A$$
$$y : B$$

gd_W
$$P[x/a, y/b]$$

do_W
$$Q[x/a, y/b]$$

Sequential Composition. Sequential composition is an operation that begins in an initial state of the operation Op_1 and ends in a final state of Op_2. This makes only sense when the final state of Op_1 matches the initial state of Op_2. Given two operation schemas Op_1 and Op_2 both including primed and unprimed copies of a state schema S, then operation composition is the result of applying operation Op_2 to the result of applying Op_1. It is defined by

$$Op_1 \mathbin{;} Op_2 = \exists\, S'' \bullet \\ (\exists\, S' \bullet [Op_1;\ S'' \mid \theta S' = \theta S'']) \land (\exists\, S \bullet [Op_2;\ S'' \mid \theta S = \theta S''])$$

where θ is the operator to construct the set of corresponding bindings. Note, that $\theta S = \theta S' \Leftrightarrow x_1' = x_1 \land \ldots \land x_n' = x_n$.
Sequential composition can be calculated using renaming and hiding, i.e.

$$Op_1 \mathbin{;} Op_2 = (Op_1[x''/x'] \land Op_2[x''/x]) \setminus (x'')$$

when x is the only state variable, which is equivalent to

$Op_1 \mathbin{;} Op_2$

$Decl\_(Op_1 \mathbin{;} Op_2)$
$$x, x' : X$$

$gd\_(Op_1 \mathbin{;} Op_2)$
$$\exists\, x'' \bullet P[x''/x'] \land S[x''/x']$$

$do\_(Op_1 \mathbin{;} Op_2)$
$$\exists\, x'' \bullet (P[x''/x'] \land S[x''/x']) \\ \land (Q[x''/x] \land T[x''/x])$$

Using $Op_1[x''/x'] \land Op_2[x''/x] = (Op_1 \land Op_2)[x''/x', x''/x]$ the above schema can be simplified to

$Op_1 \fatsemi Op_2$

$Decl\_(Op_1 \fatsemi Op_2)$
$$x, x' : X$$

$gd\_(Op_1 \fatsemi Op_2)$
$$\exists\, x'' \bullet (P \land S)[x''/x']$$

$do\_(Op_1 \fatsemi Op_2)$
$$\begin{aligned}&\exists\, x'' \bullet (P \land S)[x''/x'] \\ &\qquad \land (Q \land T)[x''/x]\end{aligned}$$

8.3.9 Further Classical Laws

In this subsection we deal with three laws that do not hold over guarded precondition schemas even though they hold in standard Z. First, the law of excluded middle fails, because we now have a third truth value. Second, the contradiction law fails due to similar reasons. Finally, we show that the definition law, relating implication to negation and disjunction, does not hold either.

The Law of the Excluded Middle: $U \lor \neg\, U = \text{TRUE}$

The law of the excluded middle, adapted to the guarded precondition schema calculus does not hold:

$$\begin{aligned}
& U \lor \neg\, U \\
\equiv\ & \{\text{Definition of } U\} \\
& (P, Q) \lor \neg\, (P, Q) \\
\equiv\ & \{\text{Schema Negation}\} \\
& (P, Q) \lor (\neg\, (P \land Q), \neg\, P) \\
\equiv\ & \{\text{Schema Disjunction}\} \\
& (P \lor \neg\, (P \land Q), (P \land Q) \lor \neg\, P) \\
\equiv\ & \{\text{Classical de Morgan and Commutativity}\} \\
& (P \lor \neg\, P \lor \neg\, Q, (\neg\, P \lor P) \land (\neg\, P \lor Q)) \\
\equiv\ & \{\text{Classical Excluded Middle}\} \\
& (\text{true}, \text{true} \land (\neg\, P \lor Q)) \\
\equiv\ & \{\text{Commutativity and One Law}\} \\
& (\text{true}, \neg\, P \lor Q)
\end{aligned}$$

This operation can always be applied but its result is only defined if $\neg\, P$ holds, i.e. the outcome is *false*, or where Q holds, i.e. the outcome was specified. In case of the undefined area nothing can be said. This conforms to the standard point of view in three-valued logic.

The Contradiction Law: $U \wedge \neg\, U = \text{FALSE}$

The contradiction law, adapted to the guarded precondition schema calculus does not hold:

$$
\begin{aligned}
& U \wedge \neg\, U \\
\equiv\ & \{\text{de Morgan Law, Double Negation Law, Commutativity}\} \\
& \neg\,(U \vee \neg\, U) \\
\equiv\ & \{\text{Law of Excluded Middle}\} \\
& \neg\,(\text{true}, \neg\, P \vee Q) \\
\equiv\ & \{\text{Schema Negation}\} \\
& (\neg\,(\neg\, P \vee Q), \neg\ \text{true}) \\
\equiv\ & \{\text{Classical Predicate Logic}\} \\
& (P \wedge \neg\, Q, \text{false})
\end{aligned}
$$

This resulting operation can be applied exactly in the undefined area of the given operation but no postcondition is specified. This result is somehow surprising because most three-valued logics obey the contradiction law. Paraconsistent logics are a set of logics where the contradiction law does not hold. This raises the question whether our schema calculus is paraconsistent. After introducing entailment in Section 8.3.10 we show, however, that this is not the case.

Definition Law: $U \Rightarrow V = \neg\, U \vee V$

There are two ways of looking at the definition law. On the one hand, it is a law reflecting the relation between negation, disjunction and implication. On the other hand, it is a definition, defining implication in terms of negation and conjunction. We show that $\neg\, U \vee V$ does not reflect the truth table of implication as given in Section 8.3.1. Using the schema calculus we derive:

$$
\begin{aligned}
& \neg\, U \vee V \\
\equiv\ & \{\text{Definition of } U \text{ and } V\} \\
& \neg\,(P, Q) \vee (S, T) \\
\equiv\ & \{\text{Schema Negation}\} \\
& (\neg\,(P \wedge Q), \neg\, P) \vee (S, S \wedge T) \\
\equiv\ & \{\text{Schema Disjunction}\} \\
& (\neg\,(P \wedge Q) \vee S, \neg\, P \vee (S \wedge T))
\end{aligned}
$$

Therefore, schema implication results in non-applicability if U is *true*, i.e. within its defined area and V is *false*, i.e. outside its guard. It is defined, if U is forbidden or V is defined. The corresponding truth table is:

p⇒q	**t**	**f**	$\perp$
t	**t**	**f**	$\perp$
f	**t**	**t**	**t**
$\perp$	**t**	$\perp$	$\perp$

Table 8.2: Truth Table for Schema Implication

However, this is not equivalent to the truth table given in Section 8.3.1. Implication defined via the definition law is not necessarily an entailment operation. In the next subsection we will define entailment according to the earlier given truth table.

8.3.10 Schema Entailment

In Table 8.1 a truth table for three-valued implication is given. This implication operator is a three-valued entailment operator, denoted $\rightarrow$, but does not preserve the definition law as shown in Section 8.3.9. While defining schema negation and schema conjunction we introduced an approach to infer schema representations of the operators according to their truth table. We use the same approach to define the entailment operator for guarded precondition schema.

Given are two schemas (P, Q) and (S, T) and the truth table for entailment in Table 8.1. We encode $\mathbf{t} = P \wedge Q$, $\mathbf{f} = \neg\, P$, and $\perp = P \wedge \neg\, Q$, as well as $\mathbf{t} = S \wedge T$, $\mathbf{f} = \neg\, S$, and $\perp = S \wedge \neg\, T$. We observe that the result of entailment is either *true* or *false*, i.e. there is no undefined area, which in turn means, that both the *gd*- and the *do*-part have to be the same predicate. Following the earlier approach, we derive the guard as being the part, where entailment holds, i.e. if (P, Q) is *true* and (S, T) is *false* or undefined, then $(P, Q) \rightarrow (S, T)$ should be *false* as well. $(P, Q) \rightarrow (S, T)$ is defined if (P, Q) is *false* or undefined. This leads to the following schema for entailment:

W

$Decl\_W$

$a : A$
$b : B$
$d : D$

gd_W

$\neg\ ((P \wedge Q) \wedge (\neg\ S \vee (S \wedge \neg\ T)))$

do_W

$(P \wedge Q \wedge S \wedge T) \vee (\neg\ P \vee (P \wedge \neg\ Q))$

Using the standard predicate calculus to simplify both predicates results in:

W

$Decl\_W$

$a : A$
$b : B$
$d : D$

gd_W

$(P \wedge Q) \Rightarrow (S \wedge T)$

do_W

$(P \wedge Q) \Rightarrow (S \wedge T)$

i.e. there is no undefined area. Of course, this means that the question whether one schema entails another can always be answered. It is also worth noting the relation between entailment and implication, though it does not have the same properties, as shown in the previous subsection.

Self-application of Entailment: $U \rightarrow U = \text{TRUE}$

A guarded precondition schema always entails itself:

$$
\begin{aligned}
& U \rightarrow U \\
\equiv\ & \{\text{Definition of } U\} \\
& (P, Q) \rightarrow (P, Q) \\
\equiv\ & \{\text{Definition of } \rightarrow\} \\
& ((P \wedge Q) \Rightarrow (P \wedge Q), (P \wedge Q) \Rightarrow (P \wedge Q)) \\
\equiv\ & \{\text{Classical Predicate Calculus}\} \\
& (\text{true}, \text{true}) \\
\equiv\ & \{\text{Definition of TRUE}\} \\
& \text{TRUE}
\end{aligned}
$$

One example of applying the entailment operator is found in refinement proofs. We refer to Section 8.6 for more details.

Double Entailment. Like double implication in classical logic, we define the double entailment $U \leftrightarrow V$ as the conjunction of the entailment $U \rightarrow V$ and $V \rightarrow U$. This results in substituting classical implication with double implication in the *gd*- and *do*-part of the above given definition of entailment.

Paraconsistency. We can also determine whether or not our calculus is paraconsistent by showing whether or not $U \rightarrow (\neg\, U \rightarrow V)$ is a theorem of our calculus.

$$
\begin{array}{ll}
& U \rightarrow (\neg\, U \rightarrow V) \\
\equiv & \{\text{Definition of } U \text{ and } V\} \\
& (P, Q) \rightarrow (\neg\, (P, Q) \rightarrow (R, S)) \\
\equiv & \{\text{Schema Negation}\} \\
& (P, Q) \rightarrow ((\neg\, (P \wedge Q), \neg\, Q) \rightarrow (R, S)) \\
\equiv & \{\text{Schema Entailment}\} \\
& (P, Q) \rightarrow \\
& \qquad ((\neg\, (P \wedge Q) \wedge \neg\, Q) \Rightarrow (R \wedge S), (\neg\, (P \wedge Q) \wedge \neg\, Q \Rightarrow (R \wedge S))) \\
\equiv & \{\text{Cl. de Morgan Law, Cl. Absorption Law, Schema Entailment}\} \\
& ((P \wedge Q) \Rightarrow (\neg\, Q \Rightarrow (R \wedge S)), (P \wedge Q) \Rightarrow (\neg\, Q \Rightarrow (R \wedge S))) \\
\equiv & \{\text{Classical Definition Law, Cl. de Morgan Law}\} \\
& (\neg\, P \vee \neg\, Q \vee Q \vee (R \wedge S), \neg\, P \vee \neg\, Q \vee Q \vee (R \wedge S)) \\
\equiv & \{\text{Simplification}\} \\
& (\text{true}, \text{true}) \\
\equiv & \{\text{Definition of TRUE}\} \\
& \text{TRUE}
\end{array}
$$

This theorem is a particular form of ECQ (ex contradictione quodlibet) which has to be rejected in a paraconsistent logic, because it allows an arbitrary schema to be inferred from a set of contradicting schemas. Hence, this calculus is not paraconsistent.

8.4 The Application of the Schema Operators: An Example

In this section, we present a small example to illustrate and validate the introduced schema representation as well as the use of the schema calculus. We introduce a simple specification involving even and odd numbers. We illustrate how the schema calculus can be used to combine schemas to form a larger specification.

We define two schemas *Even* and *Odd* which describe two operations for even and odd numbers. The operation *Even* works as follows: Given an even natural number, the result shall be a number which is less or equal to the given number. In contrast, given an odd natural number, the result shall be a number greater or equal to the given number. Being a natural number is a necessary requirement for the operation to be performed, therefore it is part of the guard. We assume the existence of the predicates *isEven* and *isOdd*.

*_Even_*

Decl_Even

$a? : \mathbb{N}$
$b! : \mathbb{Z}$

gd_Even

true

do_Even

$isEven(a?) \wedge b! \leq a?$

Above, we used underscores around the schema name to denote that it has not been normalised yet. Normalising the schema *_Even_* results in:

Even

Decl_Even

$a? : \mathbb{Z}$
$b! : \mathbb{Z}$

gd_Even

$a? \geq 0$

do_Even

$isEven(a?) \wedge b! \leq a?$

The already normalised operation *Odd* is specified as follows:

Odd

Decl_Odd

$a? : \mathbb{Z}$
$b! : \mathbb{Z}$

gd_Odd

$a? \geq 0$

do_Odd

$isOdd(a?) \wedge b! \geq a?$

Note, each operation can be invoked upon both even and odd natural numbers but in only one case the outcome is guaranteed. Using the schema calculus we combine both schemas to create a single operation where the outcome is defined for both even and odd numbers.

$$Numbers == Even \lor Odd$$

i.e.

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{Numbers}} \\ \hline
\begin{array}{|l}
\multicolumn{1}{l}{\textit{Decl_Numbers}} \\ \hline
a? : \mathbb{Z} \\
b! : \mathbb{Z} \\ \hline
\end{array} \\
\begin{array}{|l}
\multicolumn{1}{l}{\textit{gd_Numbers}} \\ \hline
a? \geq 0 \\ \hline
\end{array}
\quad
\begin{array}{|l}
\multicolumn{1}{l}{\textit{do_Numbers}} \\ \hline
a? \geq 0 \\
((isEven(a?) \land b! \leq a?) \lor \\
\ (isOdd(a?) \land b! \geq a?)) \\ \hline
\end{array} \\ \hline
\end{array}$$

Further we would like to report whether the operation was performed, i.e. that a natural number was given. Therefore we introduce the *REPORT* type

$$REPORT \ ::= \ error \ | \ no\_error$$

We define an operation schema that can always be executed and the only thing it does is to report its successful execution.

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{Ok}} \\ \hline
\begin{array}{|l}
\multicolumn{1}{l}{\textit{Decl_Ok}} \\ \hline
report! : REPORT \\ \hline
\end{array} \\
\begin{array}{|l}
\multicolumn{1}{l}{\textit{gd_Ok}} \\ \hline
\text{true} \\ \hline
\end{array}
\quad
\begin{array}{|l}
\multicolumn{1}{l}{\textit{do_Ok}} \\ \hline
report! = no_error \\ \hline
\end{array} \\ \hline
\end{array}$$

We can join this schema with *Numbers* to report its successful operation:

$$NumbersOk == Numbers \land Ok$$

i.e.

NumbersOk

Decl_NumbersOk

$a? : \mathbb{Z}$
$b! : \mathbb{Z}$
$report! : REPORT$

gd_NumbersOk

$a? \geq 0$

do_NumbersOk

$a? \geq 0 \wedge report! = no\_error$
$((isEven(a?) \wedge b! \leq a?) \vee$
$(isOdd(a?) \wedge b! \geq a?))$

This schema will only report *no_error* if the number $a?$ given was a natural. In case $a?$ was not a natural number we want to report an *error*, therefore, we define

Error

Decl_Error

$a? : \mathbb{Z}$
$report! : REPORT$

gd_Error

$a? < 0$

do_Error

$report! = error$

Putting everything together we derive the complete operation

$$Complete == NumbersOk \vee Error$$

i.e.

Complete

Decl_Complete

$a? : \mathbb{Z}$
$b! : \mathbb{Z}$
$report! : REPORT$

gd_Complete

true

do_Complete

$(a? \geq 0 \wedge$
$report! = no\_error \wedge$
$((isEven(a?) \wedge b! \leq a?) \vee$
$(isOdd(a?) \wedge b! \geq a?)))$
$\vee$
$(a? < 0 \wedge report! = error)$

which is now total, i.e. it can always be invoked and the outcome is guaranteed. Given any number it will now report either its failure to be able to perform a valid operation if the number is less or equal to zero or, if it is a natural number, it will be applied according to the condition set in the *do*-part.

Furthermore, this schema can now be translated into standard Z, where it is represented as:

$$
\begin{array}{|l}
\hline
\textit{do\_Complete} \\
a? : \mathbb{Z} \\
b! : \mathbb{Z} \\
\textit{report}! : \textit{REPORT} \\
\hline
(a? \geq 0 \land \textit{report}! = \textit{no\_error} \land \\
\qquad\qquad ((\textit{isEven}(a?) \land b! \leq a?) \lor (\textit{isOdd}(a?) \land b! \geq a?))) \\
\qquad \lor \\
(a? < 0 \land \textit{report}! = \textit{error}) \\
\hline
\end{array}
$$

Note that it does not matter which interpretation is used, either the guarded or precondition interpretation work correctly, because the operation is total and normalised.

8.5 Operation Applicability

In this section we re-cap the notions of operation applicability as introduced in Section 7.3.4. We distinguished a number of regions of before states that are of interest. In particular, we presented the precondition, i.e. the well-defined region; the guard, i.e. the enabled region; the undefined region, i.e. the guard permits the operation but no outcome is specified; finally, the impossible region where the operation is blocked. Here, we use the newly developed schema calculus to revise and validate our earlier definitions. For example, the definitions in Section 7.3.4 return schemas based on the standard Z notation but here we return guarded precondition schemas. We also present some meta-theoretical investigations of the relation between the different regions.

8.5.1 Schema Precondition

In standard Z, the precondition of an operation is defined as the existential quantification over the after state and output variables, i.e.

$$\text{pre } Op = \exists\, S', \textit{outs}! \bullet Op$$

We introduced existential quantification so it seems natural to investigate the result of calculating the precondition of a guarded precondition schema, i.e. the result of applying existential quantification to the after states and outputs of a guarded precondition schema. For example, for the schema *NormalisedSchema* in Section 7.2.1, the precondition is calculated as follows:

$$\mathrm{pre}\, Normalised\_Schema = \exists\, a' : \mathbb{Z} \bullet Normalised\_Schema$$

i.e.

PreNormalised_Schema

Decl
$$a : \mathbb{Z}$$

gd
$$\exists\, a' : \mathbb{Z} \bullet a \in \mathbb{N} \wedge a' \in \mathbb{N}$$

do
$$\begin{aligned} &\exists\, a' : \mathbb{Z} \bullet a \in \mathbb{N} \wedge a' \in \mathbb{N} \\ &\qquad \wedge (a')^2 \leq a < (a'+1)^2 \end{aligned}$$

$\equiv$ {Simplification and Instantiation}

PreNormalised_Schema

Decl
$$a : \mathbb{Z}$$

gd
$$a \in \mathbb{N}$$

do
$$a \in \mathbb{N}$$

The precondition of a guarded precondition schema is another schema, where the guard contains a predicate that permits the operation and the *do*-part contains the precondition predicate, i.e. the precondition is the condition such that the outcome of the operation is well-defined.

We briefly present the preconditions of some operations defined in this chapter for illustrative purpose. First, the preconditions of the operations *On* and *Off* of the heat control system are

$$\begin{aligned} \mathrm{pre}\, On = (&heat? > heat\_min \wedge fan = Off, \\ &heat? > heat\_min \wedge fan = Off \wedge heat? \geq heat\_max) \\ \mathrm{pre}\, Off = (&heat? < heat\_max \wedge fan = On, \\ &heat? < heat\_max \wedge fan = On \wedge heat? \leq heat\_min) \end{aligned}$$

Using Axiom 1 we can simplify those schemas by removing the *gd*-predicate from the *do*-part, i.e.

$$\text{pre } On = (heat? > heat\_min \wedge fan = Off, heat? \geq heat\_max)$$
$$\text{pre } Off = (heat? < heat\_max \wedge fan = On, heat? \leq heat\_min)$$

Finally, the preconditions for the number example from the last section are

$$\text{pre } Even = (a? \geq 0, isEven(a?))$$
$$\text{pre } Odd = (a? \geq 0, isOdd(a?))$$
$$\text{pre } Numbers = (a? \geq 0, \text{true})$$
$$\text{pre } Complete = (\text{true}, \text{true})$$

8.5.2 Schema Guard

The guard of an operation is the area that allows the operation to take place. We define the schema guard Op of an operation schema (P, Q) to be the schema $\exists S', outs! \bullet (P, true)$, where $(P, true)$ is the schema obtained from applying an operator gd, which returns the guard of the operation. Then guard Op = pre gd Op

GuardNormalised_Schema

Decl	
$a : \mathbb{Z}$	
gd	*do*
$a \in \mathbb{N}$	true

Please note the difference between gd, *gd*, and guard. The first one is the operator returning the guarded part of an operation, the second is the guarded part of a schema, and the third is the schema that consists of the guarded part of a schema where the *do*-part is set to *true*.

The operations *On* and *Off* are only supposed to be applied if the fan is not in the mode it will be switched to and if the current temperature is within the correct range. These constraints are expressed by the guard of the operation.

$$\text{guard } On = (heat? > heat\_min \wedge fan = Off, \text{true})$$
$$\text{guard } Off = (heat? < heat\_max \wedge fan = On, \text{true})$$

If the guard is not fulfilled, the operation can not be applied. Furthermore, if the operation is not applicable, the guard must have blocked it, i.e.

$$\neg\ Op \leftrightarrow \neg \text{ gd } Op$$

which we can show to hold by

$$
\begin{aligned}
&\neg\ Op \leftrightarrow \neg\ \text{gd}\ \ Op\\
\equiv\ &\{\text{Definition of } Op \text{ and gd } Op\}\\
&\neg\ (P, P \wedge Q) \leftrightarrow \neg\ (P, \text{true})\\
\equiv\ &\{\text{Schema Negation}\}\\
&(\neg\ (P \wedge Q), \neg\ P) \leftrightarrow (\neg\ P, \neg\ P)\\
\equiv\ &\{\text{Schema Double Entailment}\}\\
&((\neg\ (P \wedge Q) \wedge \neg\ P) \Leftrightarrow \neg\ P, (\neg\ (P \wedge Q) \wedge \neg\ P) \Leftrightarrow \neg\ P)\\
\equiv\ &\{*\}\\
&(\text{true}, \text{true})\\
\equiv\ &\{\text{Definition of TRUE}\}\\
&\text{TRUE}
\end{aligned}
$$

We perform the step marked * separately by taking only one of the two predicates into account:

$$
\begin{aligned}
&\neg\ (P \wedge Q) \wedge \neg\ P \Leftrightarrow \neg\ P\\
\equiv\ &\{\text{de Morgan Law}\}\\
&\neg\ ((P \wedge Q) \vee P) \Leftrightarrow \neg\ P\\
\equiv\ &\{\text{Absorption Law}\}\\
&\neg\ P \Leftrightarrow \neg\ P\\
\equiv\ &\{\text{Equivalence}\}\\
&\text{true}
\end{aligned}
$$

We also show that the precondition of an operation entails the guard of an operation, i.e.

$$\text{pre } Op \rightarrow \text{guard } Op$$

$$
\begin{aligned}
&\text{pre } Op \rightarrow \text{guard } Op\\
\equiv\ &\{\text{Definitions of } Op, \text{ pre and guard}\}\\
&(\exists\, S' \bullet P, \exists\, S' \bullet (P \wedge Q)) \rightarrow (\exists\, S' \bullet P, \text{true})\\
\equiv\ &\{\text{Schema Entailment}\}\\
&(\exists\, S' \bullet P \wedge \exists\, S' \bullet (P \wedge Q) \Rightarrow \exists\, S' \bullet P,\\
&\ \exists\, S' \bullet P \wedge \exists\, S' \bullet (P \wedge Q) \Rightarrow \exists\, S' \bullet P)\\
\equiv\ &\{a \wedge b \Rightarrow a \equiv \text{true}\}\\
&(\text{true}, \text{true})\\
\equiv\ &\{\text{Definition of TRUE}\}\\
&\text{TRUE}
\end{aligned}
$$

This is not really surprising since the purpose of the Axiom 1 from Section 7.3.1 was to ensure this. However, it gives additional confidence to see that it works correctly on the schema level, too.

The idea of the guard is to block the operation under certain constraints, i.e. to make the operation impossible, hence

$$\text{impo}\, Op = \neg\, (\text{guard}\ \ Op)$$

Simplifying this definition yields

$$\begin{aligned}
& \neg\, (\text{guard}\ \ Op) \\
\equiv\ & \{\text{Definition of Op}\} \\
& \neg\, (\text{guard}\ (P, Q)) \\
\equiv\ & \{\text{Definition of guard}\} \\
& \neg\, (\exists\, S', \mathit{outs}! \bullet (P, \text{true}))
\end{aligned}$$

i.e. the operation is impossible if there is no state such that the operation can be applied.

8.5.3 Undefined Schema Application

The area where the guard holds but the precondition does not is the undefined one, i.e.

$$\text{undef}\ \ Op = \text{guard}\, Op \wedge \neg\ \text{pre}\ \ Op$$

Applying the schema calculus this simplifies to

$$\begin{aligned}
& \text{guard}\, Op \wedge \neg\ \text{pre}\ \ Op \\
\equiv\ & \{\text{Definition of guard}\ \ Op \text{ and pre}\ \ Op\} \\
& \exists\, S', \mathit{outs}! \bullet (P, \text{true}) \wedge \neg\ \exists\, S', \mathit{outs}! \bullet (P, Q) \\
\equiv\ & \{\text{Existential Quantification, de Morgan Law for Quantification}\} \\
& (\exists\, S', \mathit{outs}! \bullet P, \text{true}) \wedge \forall\, S', \mathit{outs}! \bullet \neg\, (P, Q) \\
\equiv\ & \{\text{Schema Negation}\} \\
& (\exists\, S', \mathit{outs}! \bullet P, \text{true}) \wedge \forall\, S', \mathit{outs}! \bullet (\neg\, (P \wedge Q), \neg\, P) \\
\equiv\ & \{\text{Universal Quantification}\} \\
& (\exists\, S', \mathit{outs}! \bullet P, \text{true}) \wedge (\forall\, S', \mathit{outs}! \bullet \neg\, (P \wedge Q), \forall\, S', \mathit{outs}! \bullet \neg\, P) \\
\equiv\ & \{\text{Predicate Calculus}\} \\
& (\exists\, S', \mathit{outs}! \bullet P, \text{true}) \wedge (\neg\ \exists\, S', \mathit{outs}! \bullet (P \wedge Q), \neg\ \exists\, S', \mathit{outs}! \bullet P)
\end{aligned}$$

$$
\begin{aligned}
&\equiv \{\text{Schema Conjunction}\}\\
&\qquad (\exists\, S', outs! \bullet P \land \neg\, \exists\, S', outs! \bullet (P \land Q),\\
&\qquad\ \ \exists\, S', outs! \bullet P \land \text{true} \land \neg\, \exists\, S', outs! \bullet (P \land Q) \land \neg\, \exists\, S', outs! \bullet P)\\
&\equiv \{\text{Predicate Calculus}\}\\
&\qquad (\exists\, S', outs! \bullet P \land \neg\, \exists\, S', outs! \bullet (P \land Q), \text{false})
\end{aligned}
$$

which corresponds closely to the definition provided in Section 7.3.4.

Identifying these regions can help the development process of an operation. For example, totalisation means to remove the impossible area and refinement is meant to reduce the undefined region. The final product is a specification with no undefined nor impossible areas. Such a specification can then be translated into standard Z under both the guarded and the precondition interpretation.

8.6 Refinement Calculations

In the last chapter we developed an intuitive understanding for the refinement conditions for guarded precondition schemas. We presented three conditions, applicability, correctness and strengthening of the guard. Here we develop the notion of refinement further by using the schema entailment operator $\rightarrow$ instead of $\vdash$, as well as the presented schema calculus. We demonstrate the refinement conditions by means of an example.

8.6.1 Refinement Conditions

Given an abstract operation $AOp = (gd\_AOp, do\_AOp)$ and a concrete operation $COp = (gd\_COp, do\_COp)$ both over the same state $State$ with input $x? : X$ and output $y! : Y$, then COp refines AOp, denoted $AOp \sqsubseteq COp$, if and only if the following three conditions hold:

$$
\begin{aligned}
&(1)\ \ \forall\, State;\ x? : X \bullet \text{pre}\, AOp \rightarrow \text{pre}\, COp\\
&(2)\ \ \forall\, State;\ State';\ x? : X;\ y! : Y \bullet \text{pre}\, AOp \land COp \rightarrow AOp\\
&(3)\ \ \forall\, State;\ State';\ x? : X;\ y! : Y \bullet \text{gd}\, COp \rightarrow \text{gd}\, AOp
\end{aligned}
$$

Conditions (1) and (3) together ensure the precondition is the upper bound for strengthening the guard and the guard is the lower bound for weakening the precondition.

8.6.2 Example

Given are the two operation schemas *Filter* and *C_Filter* as introduced in the last chapter

Filter

- *Decl_Filter*: $a? : \mathbb{Z}$; $b! : \mathbb{Z}$
- *gd_Filter*: $a? > 0$
- *do_Filter*: $isEven(a?)$; $b! \leq a?$

C_Filter

- *Decl_C_Filter*: $a? : \mathbb{Z}$; $b! : \mathbb{Z}$
- *gd_C_Filter*: $a? > 0$; $b! < a?$
- *do_C_Filter*: $isEven(a?)$; $b! = a?/2$

Using the schema calculus we show now formally, that *C_Filter* refines *Filter*.

First, we calculate the preconditions of both operations:

$$\text{pre}\,Filter = (a? > 0, a? > 0 \land isEven(a?))$$
$$\text{pre}\,C\_Filter = (a? > 0, a? > 0 \land isEven(a?))$$

8.6.3 Applicability

The operation *Filter* is applicable if an even natural number is given. The refined operation *C_Filter* must be applicable under the same conditions. This properties follows simply from the preconditions.

$$(1)\quad \forall\, a? : \mathbb{Z} \bullet \text{pre}\,Filter \rightarrow \text{pre}\,C\_Filter$$

$$\begin{aligned}
& \text{pre}\,Filter \rightarrow \text{pre}\,C_Filter \\
\equiv\ & \{\text{Definitions of pre}\,Filter,\ \text{and pre}\,C_Filter\} \\
& (a? > 0, a? > 0 \land isEven(a?)) \rightarrow (a? > 0, a? > 0 \land isEven(a?)) \\
\equiv\ & \{\text{Schema Self-Entailment}\} \\
& \text{TRUE}
\end{aligned}$$

8.6.4 Correctness

The operation *C_Filter* can always be applied when *Filter* could. Next, we prove correctness, i.e. whether the result of *C_Filter* is a possible result of *Filter* if applied in the same situation.

$$(2)\quad \forall\, a? : \mathbb{Z};\ b! : \mathbb{Z} \bullet \text{pre}\,Filter \land C\_Filter \rightarrow Filter$$

$$
\begin{aligned}
& \operatorname{pre} \mathit{Filter} \wedge \mathit{C\_Filter} \rightarrow \mathit{Filter} \\
\equiv\ & \{\text{Definitions of } \operatorname{pre} \mathit{Filter},\ \mathit{C\_Filter}, \text{ and } \mathit{Filter}\} \\
& \qquad ((a? > 0, a? > 0 \wedge \mathit{isEven}(a?)) \\
& \quad \wedge\ (a? > 0 \wedge b! < a?, \mathit{isEven}(a?) \wedge b! = a?/2)) \\
& \rightarrow (a? > 0, \mathit{isEven}(a?) \wedge b! \leq a?) \\
\equiv\ & \{\text{Schema Conjunction}\} \\
& \qquad (a? > 0 \wedge b! < a?, a? > 0 \wedge b! < a? \wedge \mathit{isEven}(a?) \wedge b! = a?/2) \\
& \rightarrow (a? > 0, \mathit{isEven}(a?) \wedge b! \leq a?) \\
\equiv\ & \{\text{Schema Entailment}\} \\
& \quad ((a? > 0 \wedge b! < a? \wedge \mathit{isEven}(a?) \wedge b! = a?/2) \\
& \quad \Rightarrow (a? > 0 \wedge \mathit{isEven}(a?) \wedge b! \leq a?), \\
& \quad (a? > 0 \wedge b! < a? \wedge \mathit{isEven}(a?) \wedge b! = a?/2) \\
& \quad \Rightarrow (a? > 0 \wedge \mathit{isEven}(a?) \wedge b! \leq a?)) \\
\equiv\ & \{*\} \\
& (\text{true}, \text{true}) \\
\equiv\ & \{\text{Definition of TRUE}\} \\
& \text{TRUE}
\end{aligned}
$$

We look at the above reasoning step marked $*$ separately. Schema entailment is the implication of the conjunction of the *gd*- and *do*-part in both guard and effect. We consider now only one of the two schema predicates:

$$
\begin{aligned}
& \qquad (a? > 0 \wedge b! < a? \wedge \mathit{isEven}(a?) \wedge b! = a?/2) \\
& \Rightarrow (a? > 0 \wedge \mathit{isEven}(a?) \wedge b! \leq a?) \\
\equiv\ & \{\text{Splitting of } b! \leq a?\} \\
& \qquad (a? > 0 \wedge b! < a? \wedge \mathit{isEven}(a?)) \wedge b! = a?/2 \\
& \Rightarrow (a? > 0 \wedge \mathit{isEven}(a?) \wedge b! < a?) \vee (a? > 0 \wedge \mathit{isEven}(a?) \wedge b! = a?) \\
\equiv\ & \{x \wedge y \Rightarrow x \vee z = \text{true}\} \\
& \text{true}
\end{aligned}
$$

Hence, correctness holds.

8.6.5 Strengthening of the Guard

Finally, the guard of *C_Filter* must not be less restrictive than the guard of *Filter*:

(3) $\forall\, a? : \mathbb{Z};\ b! : \mathbb{Z} \bullet \operatorname{gd} \mathit{C\_Filter} \rightarrow \operatorname{gd} \mathit{Filter}$

$$
\begin{aligned}
& \mathrm{gd}\, C\_Filter \rightarrow \mathrm{gd}\, Filter \\
\equiv\ & \{\text{Definition of gd } C\_Filter \text{ and gd } Filter\} \\
& (a? > 0 \wedge b! < a?, \mathrm{true}) \rightarrow (a? > 0, \mathrm{true}) \\
\equiv\ & \{\text{Schema Entailment}\} \\
& (a? > 0 \wedge b! < a? \Rightarrow a? > 0, a? > 0 \wedge b! < a? \Rightarrow a? > 0) \\
\equiv\ & \{\text{Classical Definition Law for Implication and de Morgan Law}\} \\
& (\neg\,(a? > 0) \vee \neg\,(b! < a?) \vee (a? > 0), \\
& \ \neg\,(a? > 0) \vee \neg\,(b! < a?) \vee (a? > 0)) \\
\equiv\ & \{\text{Classical Law of Excluded Middle}\} \\
& (\mathrm{true}, \mathrm{true}) \\
\equiv\ & \{\text{Definition of TRUE}\} \\
& \mathrm{TRUE}
\end{aligned}
$$

All three properties hold and, therefore, the schema *C_Filter* refines the schema *Filter*, i.e. it could replace it without a user noticing it.

8.7 Summary

The aim of this chapter was to develop a schema calculus for schemas that can represent un(der)definedness in Z more explicitly than those in standard Z. We provided a set of rules to enable a specifier to join schemas, to calculate certain properties of schemas, including its precondition and guard. We also showed, that this schema calculus can be used in the refinement process.

Note, however, that it was necessary to distinguish between schema implication, as defined via negation and disjunction, and entailment. Nevertheless, it turned out that the definition of the entailment operator is based upon classical implication itself. Furthermore, we demonstrated that not all reasoning rules from classical logic hold within this work. This even led to the question, whether the calculus possesses the property of being paraconsistent. This had to be rejected due to the fact that a variant of EC does hold.

The development of the schema calculus has been based on a three-valued semantics. We have not formally shown that the calculus is sound and complete with respect to this semantics. This remains future work. However, the proof theoretical properties that we validated in this chapter give us enough confidence to believe that the calculus is correct.

Chapter 9

Conclusion

The aim of this thesis was to investigate the support we can give to reasoning about inconsistent specifications written in the Z notation. We decided to explore the usefulness of applying a paraconsistent logic to achieve our goal. It turned out that paraconsistent logics had not been applied extensively to reasoning about theories in rich languages, like Z. On the other side, some of the logics we studied also provided information on how to handle incomplete information. This raised our interest in studying the problem of underdefinedness in Z.

Inconsistency is a re-occurring problem in specification development. Our work provides some insights on how inconsistent specifications can be better managed. We used quasi-classical logic to reason about Z specifications. This, however, required several improvements of QCL. Our work is therefore not only relevant to the Z community. We provide the logicians interested in paraconsistency with a complex application area leading to new challenges for their research. In the context of this work it included to add a theory of equality to QCL.

Specifying and handling underdefinedness in Z has been a topic of research for some time. Our work contributes to the development in this area by providing a modified schema representation, by presenting refinement rules and by introducing a schema calculus. We decided to base our work on a three-valued interpretation because it provides an intuitive account for operation applicability. Furthermore, we were able to use previous research on three-valued logics to guide our work.

The main results of this thesis relate to our work on quasi-classical logic, on applying QCL to reasoning about Z specifications and refinement of inconsistent Z specifications, as well as to our research on handling underdefinedness in Z. We present a summary of our findings, followed by a discussion on the value of our work, including suggestions for improvements, followed by a more general account of possible future work.

9.1 Results

Our work contributes both to the development of quasi-classical logic and to the research on the Z notation. This work consists mainly of three parts: the introduction and development of QCL, the application of QCL to reasoning about inconsistent Z specifications and the presentation of a schema notation for underdefinedness, including refinement rules and a schema calculus.

9.1.1 Quasi-Classical Logic

In Chapter 4 we introduced quasi-classical logic and we investigated some notions of logical equivalence for QCL. First, it turned out that a notion of logical equivalence based only on the QC consequence relation is not sufficient because it is not transitive. However, it is a necessary condition for equivalence to hold. Then we investigated other possibilities to define an appropriate notion of equivalence. As a part of this investigation we found that the absorption laws do not generally hold in QCL. We finally defined a strong notion of equivalence based on the equivalence of weak and strong model classes.

In Chapter 5 we developed quasi-classical logic with equality. We presented the semantics and proof theory for reasoning about equality. Furthermore, we showed the validity of the one-point rule in QCL, a rule commonly used to eliminate existential quantification.

In Appendix A we present an implementation of the QCL tableau method based on leanT^AP by (Beckert and Posegga, 1994). We consider this work in progress because we have not yet verified our implementation. It has been tested though on available examples. Our implementation contributes to the usability of QCL as a tool to reason about inconsistent theories. It also raised a small issue regarding some weakening of focusing in the disjunction S-rule of QCL.

9.1.2 The Application of Quasi-Classical Logic in Z

In Chapter 6 we applied QCL to reasoning about inconsistent Z specifications. The outcome was that QCL allows fewer but more useful, inferences in the presence of inconsistency.

In Section 6.4 we developed a notion of quasi-classical precondition. This enables the analyst to determine the intended applicability of the operation. By comparison with the standard precondition it is possible to check operations for consistency.

In Section 6.5 we investigated the process of refinement of inconsistent Z specifications. We established the notion of QC applicability extending the standard notion. QC applicability is stronger than the standard form with respect to inconsistencies in the sense that it validates fewer refinements. We also investigated a notion of QC correctness.

9.1.3 Guarded Precondition Schemas

In Chapter 7 we developed a schema representation that enables the representation of both guards and preconditions in a single notation. We generalised previous work by allowing arbitrary predicates in the guards. This required, however, a notion of guard calculation, similar to precondition calculation. Operations were given a three-valued semantics to capture the intuition behind their applicability. This led to a rather simple notion of operation refinement.

In Section 7.5 we developed rules to verify the correctness of the refinement of guarded precondition schemas. Operation refinement is seen as removal of underdefinedness and non-determinism. It is a feature of our refinement conditions that they provided boundaries for weakening of the precondition and strengthening of the guard. Furthermore, we showed that the given conditions generalise the standard operation refinement rules in both guarded and precondition interpretation.

Finally, in Chapter 8, we developed a schema calculus for guarded precondition schemas. We established the validity of our schema operators by proving several conditions that seem useful to hold. It turned out, however, that the law of the excluded middle, the contradiction law and the definition law for implication do not hold. We defined a new schema entailment operator to facilitate reasoning about guarded precondition schemas. We showed its validity by re-considering operation refinement.

9.2 Discussion

9.2.1 Z and QCL

The goal of our research is to manage inconsistencies in formal specifications written in the Z notation. The current view is that an inconsistent Z specification is meaningless. Previous work on handling inconsistency in Z focused, therefore, on creating consistent specifications. This includes to either avoid or eradicate inconsistencies or on separating contradicting concerns into hierarchies of consistent viewpoints. Our work, however, provides a novel view on the problem by proposing to manage inconsistency by means of using a paraconsistent logic to reason about inconsistent Z specifications.

In standard logic, a single inconsistency in a set of assumptions leads to the problem of triviality, i.e. any well-formed formula in the given language is a valid conclusion from the assumptions. The formal specification language Z is based on standard predicate logic. Therefore, it is said that an inconsistency in a Z specification renders the specification meaningless. Paraconsistent logics, however, avoid triviality in the presence of inconsistency. Therefore, they are suitable to our task of managing inconsistency and we decided to investigate the Z notation being supported by a paraconsistent logic.

We chose quasi-classical logic to reason about Z specifications because we think that its properties make it rather suitable for this task. Furthermore, QCL has been previously applied to reasoning about specifications. These specifications, however, were written in standard predicate logic. One challenge we faced was to investigate QCL's usefulness for reasoning about formulae constructed using a language much richer than predicate logic. This opened some interesting directions for research on QCL itself, as discussed below.

We see our work on managing inconsistency in Z only as a starting point. We provide, however, some interesting insights into the nature of inconsistency in Z specification and its consequences, in particular, to refinement of operations. We showed that QCL allows fewer but more useful conclusions to be drawn from inconsistent specifications. This should help to analyse even inconsistent Z specifications in more detail and thus facilitate validation and verification without constant removal of inconsistency.

The process of developing an abstract specification towards an implementation, i.e. refinement, is an important task in software engineering. In order for refinement to be useful, however, requires the abstract specification to be consistent. Managing inconsistency and being able to derive and verify only useful refinements seems to reduce the problem of inconsistency. The theory of refinement developed in this work is not yet complete. We are missing a QC correctness condition to further eliminate non-corresponding refinements. However, the idea of using both standard and QC refinement rules together can prove valuable.

Note, the aim is not to build inconsistent specifications, a task not very difficult, nor to distract from the danger of inconsistency, in particular in later stages of the development. Our work serves the purpose to understand the intention behind an inconsistent specification and, thus, to give it a meaning. Given a meaning, such specifications can be useful to guide further development.

9.2.2 QCL and Z

On the other side, a real-world specification notation like Z provides an interesting field of research for logicians interested in paraconsistent logic. Being well-established in the formal methods community and undergoing standardisation, the Z notation cannot be altered much. For example, the Z standard determines the meanings of the logical operators. Therefore, it is not really possible to change the meaning of negation or implication. This eliminates a wide range of paraconsistent logics from being applicable with respect to Z.

Furthermore, the Z notation is a very expressive language. It is based on first-order predicate logic with equality and incorporates an extensive mathematical toolkit. Equality, however, is a property not often considered in paraconsistent logics. In particular, quasi-classical logic did not provide means to reason about equality. Therefore, we contributed to the development of QCL by incorporating reasoning about equality.

We note, however, that equality introduces a problem of "partial" triviality. Reasoning about equality is achieved by grouping all equal terms into equivalence classes. However, in the presence of inconsistency two classically distinct equivalence classes collapse to form just one class. In the case of numbers, in particular, this leads to all numbers belonging to the same equivalence class if there is one single inconsistency.

(Mortensen, 1995) links this problem to the property of functionality of equality. He proposes to weaken functionality to control the collapse of equivalence classes. This could, for example, include to apply functionality only in the consistent case. The consequences of such an approach are, however, not clear yet. Another approach could be to follow QCL's idea of using compositional and decompositional rules. The equality rules, however, seem not to fit such a distinction. Both problems suggest, though, that equality and paraconsistency have an interesting link that needs further investigation.

Reasoning about Z specifications includes a variety of tasks. For example, it is common to determine the precondition of an operation to check the applicability of an operation. Investigating such a task raised new questions on what the precondition is of an inconsistent operation and how to simplify a precondition. In particular, we needed to look at the notion of logical equivalence in QCL and of the validity of the one-point rule. It follows from these examples that the Z notation provides an interesting benchmark for the applicability of a paraconsistent logic, like QCL.

9.2.3 Underdefinedness in Z

It has been observed that it is sometimes convenient to use a combination of the guarded and precondition interpretation to allow both modelling of refusals and underdefinedness. Our work contributes to investigations into this issue by extending previous work on the representation of both guards and precondition. It is novel in the sense that we used a non-standard semantics of operations viz. an interpretation in three-valued logic. Furthermore, our notion is more expressive by allowing after-state variables in the guard.

Refinement is an important concept in developing specifications further. Our operation refinement conditions enable the continuous development of guarded precondition schemas considering guards and preconditions at the same time. This approach ensures that preconditions cannot be weakened beyond the guard and that the guard cannot be strengthened further than the precondition. This is an essential difference to the work by (Strulo, 1995).

The Z schema calculus is used to structure and develop specifications. By providing a schema calculus for guarded precondition schemas we facilitate structural development of specifications modelling underdefinedness explicitly. Furthermore, the entailment operator enables us to analyse specifications in much the same way as in standard Z.

9.3 Future Work

This thesis draws to an end but our research is just at its beginning. During our investigation many questions were raised and only a few could be answered here. The future work consist of further investigations of QCL, of analysing its applicability to Z further, in particular the notion of refinement, and the handling of underdefinedness and inconsistency in combination. Furthermore, we are interested in applying our research to more elaborate examples.

9.3.1 Properties of Quasi-Classical Logic

For quasi-classical logic the property of transitivity fails in general. (Tennant, 1984) notes that for his logic transitivity fails as well, but only where it ought to, i.e. transitivity fails only in the presence of inconsistency. Such a property for QCL would certainly be interesting when analysing consistent theories. This would, in general, make QCL more useful when applying it not only to investigate inconsistent but also consistent theories.

We identified a problem of “partial” triviality when adding equality to QCL. Analysing the relationship between equality, functionality and inconsistency in the context of QCL can provide more insight into reasoning about inconsistencies in general and about inconsistencies in Z in particular. Further research into equality also includes extending the theorem prover with equality rules.

The Z standard does not fix a logic for Z and it is said that any logic compliant with the standard is sufficient. The question that follows is whether QCL is a sufficient logic with respect to the Z standard?

9.3.2 Refinement of Inconsistent Specifications

One major motivation for this work is the belief in a theory that allows continued development of specifications despite the presence of inconsistencies. Refinement is one of the processes of specification development from an abstract form to a more concrete representation. Refinement is also the process of adding information. This can, however, lead to the introduction of inconsistencies. The idea behind the alternative precondition regions is to support refinement in the presence of overdefinedness. Current investigations suggest that a combination of classical and quasi-classical refinement rules can support detection and controlled removal of inconsistencies. However, this relation requires further investigation.

Our work focused on operation refinement. However, data refinement is frequently used to develop a more concrete representation of the system's components. For example, sets are a mathematical notion which are usually not used in programs. During data refinement they are turned into sequences or arrays thus providing a more concrete representation. Inconsistencies can, for instance, occur due to different opinions on the concrete representation. Thus, managing inconsistency in data refinement and subsequently in operation refinement is an important issue to look at.

9.3.3 Inconsistency and Underdefinedness

So far, we only considered local inconsistencies. Surely, to develop a practically useful way of managing inconsistency we need to consider global inconsistencies too. On such a scale, however, it becomes even more important to identify an order of "harmfulness" of inconsistencies.

In our interpretation of pairs of guarded precondition schemas (gd_Op, do_Op) we identified only three regions. Clearly, we could further distinguish the areas $\neg\ gd\_Op \wedge \neg\ do\_Op$ and $\neg\ gd\_Op \wedge do\_Op$. The latter area might be regarded as representing "miracles" or inconsistency. This leads to the problem of detecting and managing inconsistency between the guarded and the preconditioned region.

9.3.4 Applications

A theory of refinement in the presence of inconsistency can contribute to work on viewpoint specifications (Boiten et al., 1999), where the unification of two or more viewpoints is defined as the least common refinement of the viewpoints. So far, the verification of this property also contains a consistency check between the viewpoint specifications. However, this forces removal of the inconsistency to unify the viewpoints. Our work can possibly support viewpoint unification and the analysis of the resulting specification without necessarily removing the inconsistency.

The usefulness of after-state variables in guards is sometimes doubted. This is due to the evaluation of the guard before executing the operation. However, there are at least two application domains that could benefit from after-state guards. In genetic programming, for example, a range of solutions is calculated provided some initiation but only a small set of the solutions are selected according to some given criteria. These criteria can possibly be expressed in terms of after-state guards.

(Turski, 2001) presents an unorthodox way of specifying behaviour. He uses so called doubly guarded actions where two guards are associated with each action: the *preguard* is specifying the condition in which the action is to be started and the *postguard* is specifying under which condition the result is to be accepted. Again, we think that this is expressed within our approach of guarded precondition schemas. Thus, it would be interesting to investigate the applicability of our notation with respect to these applications.

Finally, some case studies on handling inconsistency in large projects using QCL and the Z notation would be interesting to further validate our approach and the usefulness of inconsistency tolerant methods. Also, a case study using guarded precondition schemas to model reactive behaviour would further support our work.

Bibliography

Abrial, J.-R. (1974). Data Semantics. In Klimbie, J. W. and Koffeman, K. L., editors, *IFIP TC2 Working Conference on Data Base Management*, pages 1–59. Elsevier Science Publishers (North-Holland).

Abrial, J.-R. (1996). *The B-Book: Assigning Programs to Meanings*. Cambridge University Press.

Anderson, A. R. and Belnap, N. D. (1975). *Entailment: The Logic of Relevance and Necessity*, Volume 1. Princeton University Press.

Arieli, O. and Avron, A. (1998). The Value of the Four Values. *Artificial Intelligence*, 102(1):97–141. Online `http://www.math.tau.ac.il/~aa/papers.html` (08/02/2001).

Arthan, R. D. (1992). On Free Type Definitions in Z. In Nicholls, J. E., editor, *Z User Workshop, York 1991*, Workshops in Computing, pages 40–58. Springer-Verlag. Online: `http://www.lemma-one.com/papers/18.ps` (26/07/2002).

Balzer, R. (1991). Tolerating Inconsistency. In *Proceedings of the 13th International Conference on Software Engineering*, pages 158–165. IEEE Computer Society Press / ACM Press.

Barden, R., Stepney, S., and Cooper, D. (1994). *Z in Practice*. Prentice Hall.

Barrett, G. (1989). Formal Methods Applied to a Floating-Point Number System. *IEEE Transactions on Software Engineering*, 15(5):611–621. Online: `http://www.comlab.ox.ac.uk/oucl/publications/monos/PRG-58-IEEETSE.ps.gz` (16/09/2002).

Batens, D. (1999). Inconsistency-Adaptive Logics. In Orlowska, E., editor, *Logic at Work. Essays Dedicated to the Memory of Helena Rasiowa*, Studies in fuzziness and soft computing, Volume 24, pages 445–472, Heidelberg, New York. Physica-Verlag.

Batens, D. (2000). A Survey of Inconsistency-Adaptive Logics. In (Batens et al., 2000), pages 49–73.

Batens, D. and De Clercq, K. (1999). A Rich Paraconsistent Extension of Full Positive Logic. Online: `http://logica.rug.ac.be/central/writings/index.html` (24/04/2001).

Batens, D., Mortensen, C., Priest, G., and Bendegem, J.-P. V., editors (2000). *Frontiers of Paraconsistent Logic*. Research Studies Press Ltd., Baldock, Hertfordshire, England.

Beckert, B. (1997). Semantic Tableaux with Equality. *Journal of Logic and Computation*, 7(1):39–58.

Beckert, B. and Posegga, J. (1994). leanTAP: Lean, Tableau-based Theorem Proving. In Bundy, A., editor, *Proc. CADE-12*, Lecture Notes in Artificial Intelligence 814, Nancy, France. Springer Verlag. Online: `http://i12www.ira.uka.de/leantap/` (23/06/2001).

Belnap, N. D. (1977a). How a Computer Should Think. In Ryle, G., editor, *Contemporary Aspects of Philosophy*, pages 30–56. Oriel Press, Stocksfield.

Belnap, N. D. (1977b). A Useful Four-Valued Logic. In Dunn, M. J. and Epstein, G., editors, *Modern Uses of Multiple-Valued Logic*, Volume 2 of *Episteme*, Chapter 1, pages 8–37. D. Reidel Publishing Company, Dordrecht, The Netherlands.

Ben-Ari, M. (2001). *Mathematical Logic for Computer Science*. Springer-Verlag, London, 2nd edition. (First published 1993).

Bert, D., Bowen, J. P., Henson, M. C., and Robinson, K., editors (2002). *ZB2002: Formal Specification and Development in Z and B / Second International Conference of B and Z Users*. Lecture Notes in Computer Science 2272. Springer-Verlag Berlin, Grenoble, France.

Besnard, P. and Hunter, A. (1995). Quasi-Classical Logic: Non-Trivializable Classical Reasoning from Inconsistent Information. In Froidevaux, C. and Kohlas, J., editors, *Proceedings of the ECSQARU European Conference on Symbolic and Quantitive Approaches to Reasoning and Uncertainty*, Lecture Notes in Artificial Intelligence 946, pages 44–51, Berlin. Springer Verlag. Online: `http://www.cs.ucl.ac.uk/staff/A.Hunter/papers.html` (08/02/2001).

Béziau, J.-Y. (2000). What is paraconsistent logic? In (Batens et al., 2000), pages 95–112. Online: `http://www.cs.bham.ac.uk/~esslli/notes/beziau/wplb.ps` (08/02/2001).

Bicarregui, J. and Ritchie, B. (1995). Invariants, Frames and Postconditions: a Comparision of the VDM and B Notations. *IEEE Transactions on Software Engineering*, 21(2):79–89. also in: Proceedings of FME'93, Woodcock and Larsons (eds.) LNCS 670, Springer-Verlag. Online: `http://theory.doc.ic.ac.uk:80/~jcb1/ieee.ps` (24/08/2002).

Boiten, E., Derrick, J., Bowman, H., and Steen, M. W. A. (1999). Constructive consistency checking for partial specification in Z. *Science of Computer Programming*, 35(1):29–75. Online: `http://www.elsevier.com/cas/tree/store/scico/sub/1999/35/1/567.pdf` (15/10/2001).

Boiten, E. A., Derrick, J., Bowman, H., and Steen, M. W. A. (1995). Unification and multiple views of data in Z. In van Vliet, J. C., editor, *Computing Science in the Netherlands*, pages 73–85. Online: `http://www.cs.ukc.ac.uk/pubs/1995/191/` (24/08/2002).

Bowen, J. P., Dunne, S., Galloway, A., and King, S., editors (2000). *ZB2000: Formal Specification and Development in Z and B / First International Conference of B and Z Users*. Lecture Notes in Computer Science 1878. Springer-Verlag Berlin, York, UK.

Bowen, J. P., Fett, A., and Hinchey, M. G., editors (1998). *ZUM '98: The Z Formal Specification Notation, Proceedings of the 11th International Conference of Z Users*. Lecture Notes in Computer Science 1493. Springer Verlag, Berlin Heidelberg New York.

Bowman, H., Derrick, J., Linington, P., and Steen, M. W. A. (1996). Cross Viewpoint Consistency in Open Distributed Processing. *Software Engineering Journal*, 11(1):44–57. Special Issue on Viewpoints, editors: A. Finkelstein and I. Sommerville.

da Costa, N. C. A. (1974). On the Theory of Inconsistent Formal Systems. *Notre Dame Journal of Formal Logic*, 15(4):497–510.

da Costa, N. C. A. (2000). Paraconsistent Mathematics. In (Batens et al., 2000), pages 165–179.

da Costa, N. C. A., Béziau, J.-Y., and Bueno, O. A. S. (1995). Aspects of Paraconsistent Logic. *Bulletin of the IGPL*, 3(4):597–614. Online: `http://www.mpi-sb.mpg.de/igpl/Journal/V3-4/#Dacosta` (08/02/2001).

Damásio, C. V. and Pereira, L. M. (1998). A Survey of Paraconsistent Semantics for Logic Programs. In Besnard, P. and Hunter, A., editors, *Reasoning with Actual and Potential Contradictions*, Volume II of Handbook of Defeasable Reasoning and Uncertainty Management Systems (Gabbay, D. and Smets, Ph., editors), pages 277–364. Kluwer Academic Publishers, Dortrecht, The Netherlands. Online: http://www.univ-ab.pt/~cd/investigacao/artigos/hand97.ps.gz (21/09/1999).

deCharms, C. (1997). Approaching the Tibetian View of Mind. In *Two Views of Mind: Abhidharma and Brain Science*, pages 25–28. Snow Lion Publications, Ithaca NY 14851-6483. See http://www.keck.ucsf.edu/~decharms/.

Derrick, J. and Boiten, E. (2001). *Refinement in Z and Object-Z: Foundations and Advanced Applications*. Formal Approaches to Computing and Information Technology. Springer Verlag.

Derrick, J., Bowman, H., and Steen, M. W. A. (1995). Maintaining Cross Viewpoint Consistency using Z. In Raymond, K. and Armstrong, L., editors, *IFIP TC6 International Conference on Open Distributed Processing*, pages 413–424, Brisbane, Australia. Brisbane, Australia, Chapman & Hall. Online: http://www.cs.ukc.ac.uk/pubs/1995/187/index.html (13/04/1999).

Doornbos, H. (1994). A relational model of programs without the restriction to Egli-Milner constructs. In Olderog, E.-R., editor, *PROCOMET '94*, pages 357–376. IFIP.

Duke, R., Rose, G., and Smith, G. (1994). Object-Z: A Specification Language Advocated for the Description of Standards. Technical report 94-45, Software Verification Research Centre, School of Information Technology, The University of Queensland, Brisbane 4072. Australia. Online: http://svrc.it.uq.edu.au/Bibliography/svrc-tr.html?94-45 (05/08/2002).

Easterbrook, S. (1993). Domain modelling with hierarchies of alternative viewpoints. In *Proceedings of the IEEE International Symposium on Requirements Engineering*, pages 65–72. IEEE. Technical Report Online: http://www.cogs.susx.ac.uk/cgi-bin/htmlcogsreps?csrp252 (29/08/2002).

Easterbrook, S. and Chechik, M. (2001a). 2nd International Workshop on Living with Inconsistency (Part of ICSE 2001). Online: http://www.cs.toronto.edu/~sme/IWLWI-01/ (29/08/2002).

Easterbrook, S. and Chechik, M. (2001b). A Framework for Multi-Valued Reasoning over Inconsistent Viewpoints. In *Proceedings of the 23rd International Conference on Software Engeneering (ICSE-01)*, pages 411–420, Los Alamitos, California. IEEE Computer Society. Online: http://www.cs.toronto.edu/~sme/papers/2001/ICSE01.pdf (03/06/2001).

Easterbrook, S. and Nuseibeh, B. (1996). Using ViewPoints for inconsistency management. *Software Engineering Journal*, 11(1):31–43. Online: http://www.doc.ic.ac.uk/~ban/pubs.html (13/08/2002).

Finkelstein, A., Gabbay, D., Hunter, A., Kramer, J., and Nuseibeh, B. (1994). Inconsistency Handling in Multi-Perspective Specifications. *IEEE Transactions on Software Engineering*, 20(8):569–578. Online: `http://www.cs.ucl.ac.uk/staff/A.Hunter/papers.html` (19/09/1999).

Fischer, C. (1997). CSP-OZ: A Combination of Object-Z and CSP. Technical Report TRCF-97-2, Universität Oldenburg, Fachbereich Informatik, PO Box 2503, 26111 Oldenburg, Germany. Online: `http://theoretica.Informatik.Uni-Oldenburg.DE/~fischer/techreports.html` (10/01/2000).

Fischer, C. (1998). How to Combine Z with Process Algebra. In (Bowen et al., 1998), pages 5–23.

Fitting, M. C. (1996). *First-Order Logic and Automated Theorem Proving*. Graduate Texts in Computer Science. Springer-Verlag, Berlin, 2nd edition. 1st ed., 1990.

Gabbay, D. M. and Hunter, A. (1991). Making inconsistency respectable 1: A logical framework for inconsistency in reasoning. In Jorrand, P. and Kelemen, J., editors, *Foundations of Artificial Intelligence Research*, Lecture Notes in Computer Science 535, pages 19–32. Springer Verlag. Online: `http://www.cs.ucl.ac.uk/staff/A.Hunter/papers.html` (08/02/2001).

Gabbay, D. M. and Hunter, A. (1999). Negation and Contradiction. In Gabbay, D. M. and Wansing, H., editors, *What is Negation?*, Volume 13 of *Applied Logic Series*. Kluwer Academic Publishers. Online: `http://www.cs.ucl.ac.uk/staff/A.Hunter/papers.html` (01/03/2001).

Ghezzi, C. and Nuseibeh, B. (1998). Guest Editorial: Introduction to the Special Section: Managing Inconsistency in Software Development. *IEEE Transactions on Software Engineering*, 24(11):906–907.

Ghezzi, C. and Nuseibeh, B. (1999). Guest Editorial: Introduction to the Special Section: Managing Inconsistency in Software Development. *IEEE Transactions on Software Engineering*, 26(6):782–783.

Hähnle, R., Beckert, B., and Gerberding, S. (1994). $_3T^AP$ *- The Many-Valued Theorem-Prover*, 3rd edition. Online: `http://i12www.ira.uka.de/~threetap` (28/08/2002).

Hayes, I. J., editor (1987). *Specification Case Studies*. Prentice Hall International Series in Computer Science. Prentice Hall International (UK) Ltd., 1st edition.

Hayes, I. J., editor (1993). *Specification Case Studies.* Prentice Hall International Series in Computer Science. Prentice Hall International (UK) Ltd., 2nd edition.

Hayes, I. J., Jones, C. B., and Nicholls, J. E. (1993). Understanding the differences between VDM and Z. Technical Report UMCS-93-8-1, Department of Computer Science, University of Manchester. Online: `http://www.cs.man.ac.uk/cstechrep/Abstracts/UMCS-93-8-1.html` (05/08/2002).

Hehner, E. C. R. (1993). *A practical theory of programming.* Springer Verlag.

Hehner, E. C. R. (1999). Specifications, programs, and total correctness. *Science of Computer Programming*, 34(3):191–205. Online `http://www.elsevier.com/cas/tree/store/scico/sub/1999/34/3/563.pdf` (09/05/2000).

Henson, M. C. (1998). The standard logic of Z is inconsistent. *Journal of Formal Aspects of Computing*, 10:243–247. Online: `ftp://ftp.essex.ac.uk/pub/csc/technical-reports/z.ps` (23/06/2001).

Henson, M. C. and Reeves, S. (2000). Investigating Z. *Journal of Logic and Computation*, 10. Online: `ftp://ftp.essex.ac.uk/pub/csc/technical-reports/jlc.ps` (23/06/2001).

Herre, H. (1998). *A Paraconsistent Semantics for Generalized Logic Programs.* Augustusplatz 10-11, 04109 Leipzig, Germany. Preliminary copy (paracs.ps.gz).

Herre, H. and Pearce, D. (1992). Disjunctive Logic Programming, Constructivity and Strong Negation. In Pearce, D. and Wagner, G., editors, *Logics in AI - European Workshop JELIA'92, Berlin, Germany, September 1992*, Lecture Notes in Artificial Intelligence 633, pages 391–410. Springer Verlag, Berlin, Heidelberg.

Hoare, C. A. R. and He Jifeng (1998). *Unifying Theories of Programming.* Prentice Hall, London.

Hunter, A. (2000). Reasoning with Contradictory Information Using Quasi-Classical Logic. *Journal of Logic and Computation*, 10(5):677–703. Online: `http://www.cs.ucl.ac.uk/staff/A.Hunter/papers.html` (08/02/2001).

Hunter, A. (2001). A Semantic Tableau Version of First-Order Quasi-Classical Logic. In Benferhat, S. and Besnard, P., editors, *Symbolic and Quantitative Approaches to Reasoning with Uncertainty, 6th European Conference, ECSQARU 2001, Toulouse, France, September 19-21, 2001, Proceedings*, Lecture Notes in Artificial Intelligence 2143, pages 544–555. Springer Verlag. Online: `http://www.cs.ucl.ac.uk/staff/a.hunter/papers/foqc.ps` (26/08/2002).

Hunter, A. and Nuseibeh, B. (1997). Analysing Inconsistent Specifications. In *Proceedings of the 3rd International Symposium on Requirements Engineering (RE'97)*, pages 78–86. Annapolis, USA, IEEE Computer Society Press. Online: `http://www.cs.ucl.ac.uk/staff/a.hunter/papers.html` (14/02/2001).

Hunter, A. and Nuseibeh, B. (1998). Managing Inconsistent Specifications: Reasoning, Analysis, and Action. *ACM Transactions on Software Engineering and Methodology*, 7(4):335–367. Online: `http://www.cs.ucl.ac.uk/staff/A.Hunter/papers.html` (08/02/2001), `http://www.doc.ic.ac.uk/~ban/pubs.html` (13/08/2002).

ISO/IEC 13568 (2002). *- Information technology - Z formal specification notation - Syntax, type system and semantics.* ISO/IEC 13568, 1st edition.

IWLWI (1997). ICSE'97 Workshop on "Living with Inconsistency". Online: `http://www.cs.uoregon.edu/~fickas/icse-workshop/` (29/08/2002/).

Jackson, D. (1995). Structuring Z Specifications with Views. *ACM Transactions on Software Engineering and Methodology*, 4(4):365–389. Online: `http://www.cs.cmu.edu/afs/cs/project/compose/www/paper_abstracts/dnj/CMU-CS-94-126.html` (22/09/1999).

Jackson, D. and Jackson, M. (1996). Problem Decomposition for Reuse. *IEE/BCS Software Engineering Journal*, 11(1):19–30. Online: `http://sdg.lcs.mit.edu/~dnj/publications.html` (18/09/1999).

Jacky, J. (1997). *The Way of Z: Practical Programming with Formal Methods*. Cambridge University Press.

Jones, C. B. (1990). *Systematic Software Development Using VDM*. Prentice Hall International Series in Computer Science. Prentice Hall International (UK) Ltd, London, 2nd edition. Online: `ftp://ftp.ncl.ac.uk/pub/users/ncbj/ssdvdm.ps.gz` (05/08/2002).

Jones, C. B. and Shaw, R. C. F., editors (1990). *Case Studies in Systematic Software Development*. Prentice Hall International. Online: `ftp://ftp.ncl.ac.uk/pub/users/ncbj/cases.ps.gz` (05/08/2002).

Josephs, M. B. (1991). Specifying reactive systems in Z. Technical Report PRG-19-91, Programming Research Group, Oxford University Computing Laboratory.

King, S. (1990). Z and the Refinement Calculus. In Bjørner, D., Hoare, C. A. R., and Langmaak, H., editors, *Proceedings of the 3rd International Symposium of VDM Europe on VDM and Z : Formal Methods in Software Development*, Lecture Notes in Computer Science 428, pages 164–188, Berlin. Springer Verlag.

Lano, K., Bicarregui, J., Fiadeiro, J., and Lopes, A. (1997). Specification of Required Non-determinism. In Fitzgerald, J., Jones, C. B., and Lucas, P., editors, *FME'97: Industrial Applications and Strengthened Foundations of Formal Methods (Proc. 4th Intl. Symposium of Formal Methods Europe, Graz, Austria, September 1997)*, Lecture Notes in Computer Science 1313, pages 298–317. Springer-Verlag. Online: `http://theory.doc.ic.ac.uk:80/~jcb1/fme97.ps` (15/10/1999).

Lewis, C. I. and Langford, C. H. (1932). *Symbolic Logic*. Dover Publications, New York.

Meheus, J. (2002). Inconsistencies in scientific discovery: Clausius's remarkable derivation of Carnot's theorem. In Krach, H., Vanpaemel, G., and Marage, P., editors, *History of Modern Physics*, pages 143–154. Turnhout, Brepols.

Miarka, R., Boiten, E., and Derrick, J. (2000). Guards, Preconditions, and Refinement in Z. In (Bowen et al., 2000), pages 286–303. Online: `http://www.cs.ukc.ac.uk/pubs/2000/1130` (17/11/2000).

Miarka, R., Derrick, J., and Boiten, E. (2002). Handling Inconsistencies in Z using Quasi-Classical Logic. In (Bert et al., 2002), pages 204–225. Online: `http://www.cs.ukc.ac.uk/pubs/2002/` (01/02/2002).

Mortensen, C. (1995). *Inconsistent Mathematics*. Kluwer Academic Publishers, Dordrecht.

Nicholls, J. (1995). *Z Notation: Version 1.2*. Z Standards Panel.

Nix, C. J. and Collins, B. P. (1988). The use of Software Engineering, including the Z Notation, in the Development of CICS. *Quality Assurance*, 14(3):103–110.

Nuseibeh, B., Easterbrook, S., and Russo, A. (2000). Leveraging Inconsistency in Software Development. *IEEE Computer*, 33(4):24–29. Online: `http://www.doc.ic.ac.uk/~ban/pubs.html` (13/08/2002).

Nuseibeh, B., Easterbrook, S., and Russo, A. (2001). Making inconsistency respectable in software development. *Journal of Systems and Software*, 58:171–180. This is a revised and expanded version of (Nuseibeh et al., 2000). Online: `http://www-dse.doc.ic.ac.uk/~ban/pubs/jss2001.pdf` (13/01/2002).

Nuseibeh, B., Kramer, J., and Finkelstein, A. (1994). A Framework for Expressing the Relationships Between Multiple Views in Requirements Specification. *IEEE Transactions on Software Engineering*, 20(10):760–773. Online: `ftp://cs.ucl.ac.uk/acwf/papers/tse94.icse.ps.gz` (11/12/1998).

Potter, B., Sinclair, J., and Till, D. (1991). *An Introduction to Formal Specification and Z.* Prentice Hall International Series in Computer Science. Prentice Hall International (UK) Ltd.

Priest, G. (1998). Dialetheism. Stanford Encyclopedia of Philosophy. Online: `http://plato.stanford.edu/entries/dialetheism/` (11/01/1999).

Priest, G. (2000). Motivations for Paraconsistency: The Slippery Slope from Classical Logic to Dialetheism. In (Batens et al., 2000), pages 223–232.

Priest, G. et al., editors (1989). *Paraconsistent Logic: Essays in the Inconsistency.* Philosophia Verlag.

Priest, G. and Tanaka, K. (1996). Logic, Paraconsistent. Stanford Encyclopedia of Philosophy. Online: `http://plato.stanford.edu/entries/logic-paraconsistent/` (11/01/1999).

Reeves, S. V. (1987). Adding Equality to Semantic Tableaux. *Journal of Automated Reasoning*, 3(3):225–246.

Rescher, N. and Manor, R. (1970). On Inference from Inconsistent Premisses. *Theory and Decision*, 1:179–217.

Rodrigues, O. and Russo, A. (1998). A Translation Method for Belnap Logic. Technical Report, Imperial College Research Report DoC 98/7. Online: `http://www.doc.ic.ac.uk/~ar3/belnap-trans.pdf` (08/02/2001).

Saaltink, M. (1997). *The Z/EVES User's Guide.* ORA Canada, Suite 1208, One Nicholas Street; Ottawa, Ontario K1N 7B7; Canada. Online: `ftp://ftp.ora.on.ca/pub/doc/97-5493-06.ps.Z` (21/09/1999).

Schneider, S. (2001). *The B-Method – An Introduction.* Palgrave Macmillan Publishers Ltd, Houndsmills, Basingstoke, Hampshire, RG21 6XS, England.

Schwanke, R. W. and Kaiser, G. E. (1988). Living with Inconsistency in Large Systems. In *Proceedings of the International Workshop on Software Version and Configuration Control*, pages 98–118, Grassau, Germany.

Smith, A. (1992). On Recursive Free Types in Z. In Nicholls, J. E., editor, *Z User Workshop, York 1991*, Workshops in Computing, pages 3–39. Springer-Verlag.

Smith, G. (2000). *The Object-Z Specification Language*. Advances in Formal Methods. Kluwer Academic Publishers.

Smullyan, R. M. (1968). *First-Order Logic*, Volume 43 of *Ergebnisse der Mathematik und ihrer Grenzgebiete*. Springer-Verlag, New York.

Spivey, J. M. (1992). *The Z Notation: A Reference Manual*. Prentice Hall International Series in Computer Science. Prentice Hall International (UK) Ltd., 2nd edition. Out-of-print. Online: `http://spivey.oriel.ox.ac.uk/~mike/zrm/index.html` (26/07/1998).

Steen, M. W. A., Bowman, H., Derrick, J., and Boiten, E. A. (1997). Disjunction of LOTOS specifications. In Mizuno, T., Shiratori, N., Higashino, T., and Togashi, A., editors, *Formal Description Techniques and Protocol Specification, Testing and Verification: FORTE X / PSTV XVII '97*, pages 177–192, Osaka, Japan. Chapman & Hall. Online: `http://www.cs.ukc.ac.uk/pubs/1997/350` (20/01/2000).

Stepney, S., Barden, R., and Cooper, D., editors (1992). *Object Orientation in Z*. Workshops in Computing. Springer-Verlag.

Stoddart, B., Dunne, S., and Galloway, A. (1999). Undefined expressions and logic in Z and B. *Formal Methods in System Design: An International Journal*, 15(3):201–215. Online: `http://wheelie.tees.ac.uk/users/w.j.stoddart/undefzb.ps` (12/02/2000).

Strulo, B. (1995). How Firing Conditions Help Inheritance. In Bowen, J. P. and Hinchey, M. G., editors, *ZUM'95: The Formal Specification Notation*, Lecture Notes in Computer Science 967, pages 264–275. Springer Verlag.

Tennant, N. (1984). Perfect Validity, Entailment and Paraconsistency. *Studia Logica*, 43(1–2):181–200.

Turski, W. M. (2001). Programming for Behaviour. In Hoare, C. A. R., Broy, M., and Steinbrüggen, R., editors, *Engineering Theories of Software Construction*, NATO Science Series: Computer and Systems Sciences Vol. 180, pages 135–148. IOS Press.

Urbas, I. (1990). Paraconsistency. *Studies in Soviet Thought*, 39(3–4):343–354.

Valentine, S. H. (1998). Inconsistency and Undefinedness in Z – A Practical Guide. In (Bowen et al., 1998), pages 233–249.

van Lamsweerde, A., Darimont, R., and Letier, E. (1998). Managing Conflicts in Goal-Driven Requirements Engineering. *IEEE Transactions on Software Engineering*, 24(11):908–926. Online: `ftp://ftp.info.ucl.ac.be/pub/publi/98/Conflicts-TSE.ps.gz` (11/01/1999).

Vermeir, T. (2001). Inconsistency-adaptive arithmetic. Online: `http://logica.rug.ac.be/central/writings/index.html` (24/04/2001).

Wittgenstein, L. (1964). *Philosophische Bemerkungen.* Basil Blackwell, Oxford. Aus dem Nachlaß, herausgegeben von Rush Rhees.

Woodcock, J. and Davies, J. (1996). *Using Z - Specification, Refinement, and Proof.* Prentice Hall International Series in Computer Science. Prentice Hall Europe. Online: `http://www.comlab.ox.ac.uk/igdp/usingz/` (18/10/2001).

Appendix A

QC-LeanTaP: A Tableau-Based Theorem Prover for QCL

We present some work in progress on a tableau-based theorem prover for QCL. Our theorem prover, called QC-LeanTaP, is based on the work by (Beckert and Posegga, 1994) on leanT^AP which we introduce first. Then we turn to a small program to calculate the conjunctive negation normal form of a first order predicate formula. We do not skolemize existential predicates, unlike the version used for leanT^AP. Finally, we present the tableau-based theorem prover for QCL.

A.1 leanT^AP

LeanT^AP is a complete and sound theorem prover for classical first-order logic based on free-variable semantic tableau. The unique thing about leanT^AP is that it is probably the smallest theorem prover around: The original leanT^AP program is only about 12 lines of Prolog.

```
prove((E,F),A,B,C,D) :- !,prove(E,[F|A],B,C,D).
prove((E;F),A,B,C,D) :- !,prove(E,A,B,C,D),prove(F,A,B,C,D).
prove(all(I,J),A,B,C,D) :- !,
   \+length(C,D),copy_term((I,J,C),(G,F,C)),
   append(A,[all(I,J)],E),prove(F,E,B,[G|C],D).
prove(A,_,[C|D],_,_) :-
    ((A= -(B);-(A)=B) -> (unify(B,C);prove(A,[],D,_,_))).
prove(A,[E|F],B,C,D) :- prove(E,F,[A|B],C,D).
```

(Beckert and Posegga, 1994) describe the basic version of leanT^AP, which is an implementation of standard free-variable semantic tableau. An extended report on leanT^AP and the source code can be anonymously ftp-ed from i12ftp.ira.uka.de:pub/posegga/LeanTaP.ps.Z and

i12ftp.ira.uka.de:pub/posegga/LeanTaPsrc.shar.Z

LeanT^AP is written in Sicstus Prolog but to port it to GProlog was easily done. The prover lives in leantap.pl and is defined as the predicates prove/2 and prove_uv/2. See the comments there for details.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% $Id: leantap.pl,v 2.3 1994/12/14 18:09:13 posegga Exp $
% Sicstus Prolog
% Copyright (C) 1993: Bernhard Beckert & Joachim Posegga
%                     Universitaet Karlsruhe
%                     Email: {beckert|posegga}@ira.uka.de
%
% Purpose:  \LeanTaP: tableau-based theorem prover for NNF.
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

:- module(leantap,[prove/2,prove_uv/2]).

:-      use_module(library(lists),[append/3]).
:-      use_module(unify,[unify/2]).

%%%%%%%%%% BEGIN OF TOPLEVEL PREDICATES

% -----------------------------------------------------------------
% prove(+Fml,?VarLim)
% prove_uv(+Fml,?VarLim)
%
% succeeds if there is a closed tableau for Fml with not more
% than VarLim free variables on each branch.
% prove_uv uses universal variables, prove does not.
%
% Iterative deepening is used when VarLim is unbound.
% Examples:
%
% | ?- prove((p(a) , -p(f(f(a))) , all(X,(-p(X) ; p(f(X))))), 1).
% no
% | ?- prove((p(a) , -p(f(f(a))) , all(X,(-p(X) ; p(f(X))))), 2).
% yes
%
```

```
prove(Fml,VarLim) :-
        nonvar(VarLim),!,prove(Fml,[],[],[],VarLim).
prove(Fml,Result) :-
        iterate(VarLim,1,prove(Fml,[],[],[],VarLim),Result).

prove_uv(Fml,VarLim) :-
        nonvar(VarLim),!,prove(Fml,[],[],[],[],[],VarLim).

prove_uv(Fml,Result) :-
        iterate(VarLim,1,prove(Fml,[],[],[],[],[],VarLim),Result).

iterate(Current,Current,Goal,Current) :- nl,
        write('Limit = '),
        write(Current),nl,
        Goal.

iterate(VarLim,Current,Goal,Result) :-
        Current1 is Current + 1,
        iterate(VarLim,Current1,Goal,Result).

%%%%%%%%% END OF TOPLEVEL PREDICATES

% -----------------------------------------------------------------
% prove(+Fml,+UnExp,+Lits,+FreeV,+VarLim)
%
% succeeds if there is a closed tableau for Fml with not more
% than VarLim free variables on each branch.
%  Fml: inconsistent formula in skolemized negation normal form.
%       syntax: negation: '-', disj: ';', conj: ','
%       quantifiers: 'all(X,<Formula>)',
%                    where 'X' is a prolog variable.
%
%  UnExp:    list of formula not yet expanded
%  Lits:     list of literals on the current branch
%  FreeV:    list of free variables on the current branch
%  VarLim:   max. number of free variables on each branch
%            (controls when backtracking starts and alternate
%            substitutions for closing branches are considered)
%

prove((A,B),UnExp,Lits,FreeV,VarLim) :- !,
        prove(A,[B|UnExp],Lits,FreeV,VarLim).
```

```
prove((A;B),UnExp,Lits,FreeV,VarLim) :- !,
        prove(A,UnExp,Lits,FreeV,VarLim),
        prove(B,UnExp,Lits,FreeV,VarLim).

prove(all(X,Fml),UnExp,Lits,FreeV,VarLim) :- !,
        \+ length(FreeV,VarLim),
        copy_term((X,Fml,FreeV),(X1,Fml1,FreeV)),
        append(UnExp,[all(X,Fml)],UnExp1),
        prove(Fml1,UnExp1,Lits,[X1|FreeV],VarLim).

prove(Lit,_,[L|Lits],_,_) :-
        (Lit = -Neg; -Lit = Neg) ->
        (unify(Neg,L); prove(Lit,[],Lits,_,_)).

prove(Lit,[Next|UnExp],Lits,FreeV,VarLim) :-
        prove(Next,UnExp,[Lit|Lits],FreeV,VarLim).

% ---------------------------------------------------------------
% prove(+Fml,+UnExp,+Lits,+DisV,+FreeV,+UnivV,+VarLim)
%
% same as prove/5 above, but uses universal variables.
% additional parameters:
% DisV:   list of non-universal variables on branch
% UnivV:  list of universal variables on branch

prove((A,B),UnExp,Lits,DisV,FreeV,UnivV,VarLim) :- !,
        prove(A,[(UnivV:B)|UnExp],Lits,DisV,FreeV,UnivV,VarLim).

prove((A;B),UnExp,Lits,DisV,FreeV,UnivV,VarLim) :- !,
        copy_term((Lits,DisV),(Lits1,DisV)),
        prove(A,UnExp,Lits,(DisV+UnivV),FreeV,[],VarLim),
        prove(B,UnExp,Lits1,(DisV+UnivV),FreeV,[],VarLim).

prove(all(X,Fml),UnExp,Lits,DisV,FreeV,UnivV,VarLim) :- !,
        \+ length(FreeV,VarLim),
        copy_term((X,Fml,FreeV),(X1,Fml1,FreeV)),
        append(UnExp,[(UnivV:all(X,Fml))],UnExp1),
        prove(Fml1,UnExp1,Lits,DisV,[X1|FreeV],[X1|UnivV],VarLim).

prove(Lit,_,[L|Lits],_,_,_,_) :-
        (Lit = -Neg; -Lit = Neg ) ->
        (unify(Neg,L); prove(Lit,[],Lits,_,_,_,_)).
```

```
prove(Lit,[(UnivV:Next)|UnExp],Lits,DisV,FreeV,_,VarLim) :-
        prove(Next,UnExp,[Lit|Lits],DisV,FreeV,UnivV,VarLim).
```

A.2 Normal Form Calculation for QC-LeanTaP

It follows a small program to calculate the negation conjunctive normal form of a formula in first-order predicate logic. The main difference to the original version by (Beckert and Posegga, 1994) is the inclusion of two rewrite rules (distribution laws) and the removal of the skolemization. The former prevents the disjunction rule to be applied to non-literals and the latter to skolemize existentially quantified formulae. Both conditions are required because of the distinction between S- and U-rules.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% cnnf
%
% GProlog
%
% August 2001: Ralph Miarka
%              University of Kent, Canterbury, UK
%              Email: rm17@ukc.ac.uk
%
% Purpose:
%  - computes conjunctive negation normal form for a
%    formula given in first-order predicate logic
%  - used in conjunction with qc_leantap
%
% based on nnf.pl by
%
% Copyright (C) 1993: Bernhard Beckert & Joachim Posegga
%                     Universitaet Karlsruhe
%                     Email: {beckert|posegga}@ira.uka.de
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% check - xfy means right associative; yfx means left associative

:-      op(400,fy,-).    % negation
:-      op(500,xfy,&).   % conjunction
:-      op(600,xfy,v).   % disjunction
:-      op(650,xfy,=>).  % implication
```

```
:-      op(700,xfy,<=>). % equivalence

% -----------------------------------------------------------------
%  cnnf(+Fml,?CNNF)

% Fml is a first-order formula and
% CNNF is its conjunctive negation normal form.
%
% Syntax of Fml:
%  negation: '-', disj: 'v', conj: '&', impl: '=>', equiv: '<=>',
%
% Syntax of CNNF: negation: '-', disj: ';', conj: ','
%

cnnf(Fml,CNNF) :- cnnf(Fml,CNNF,_).

% -----------------------------------------------------------------
%  cnnf(+Fml,-CNNF,-Paths)
%
% Fml,CNNF    See above.
% Paths:      Number of disjunctive paths in Fml.

cnnf(Fml,CNNF,Paths) :-
        (Fml = -(-A)      -> Fml1 = A;
         Fml = -all(X,F)  -> Fml1 = ex(X,-F);
         Fml = -ex(X,F)   -> Fml1 = all(X,-F);
         Fml = -(A v B)   -> Fml1 = -A & -B;
         Fml = -(A & B)   -> Fml1 = -A v -B;
         Fml = A v (B & C)        -> Fml1 = (A v B) & (A v C);
         Fml = (A & B) v (A & C)  -> Fml1 = A & (B v C);
         Fml = (A => B)   -> Fml1 = -A v B;
         Fml = -(A => B)  -> Fml1 = A & -B;
         Fml = (A <=> B)  -> Fml1 = (A & B) v (-A & -B);
         Fml = -(A <=> B) -> Fml1 = (A & -B) v (-A & B)),!,
        cnnf(Fml1,CNNF,Paths).

cnnf(all(X,F),all(X,CNNF),Paths) :- !,
        cnnf(F,CNNF,Paths).

cnnf(ex(X,Fml),ex(X,CNNF),Paths) :- !,
        cnnf(Fml,CNNF,Paths).

cnnf(A & B,CNNF,Paths) :- !,
```

```
        cnnf(A,CNNF1,Paths1),
        cnnf(B,CNNF2,Paths2),
        Paths is Paths1 * Paths2,
        (Paths1 > Paths2 -> CNNF = (CNNF2,CNNF1);
                            CNNF = (CNNF1,CNNF2)).

cnnf(A v B,CNNF,Paths) :- !,
        cnnf(A,CNNF1,Paths1),
        cnnf(B,CNNF2,Paths2),
        Paths is Paths1 + Paths2,
        (Paths1 > Paths2 -> CNNF = (CNNF2;CNNF1);
                            CNNF = (CNNF1;CNNF2)).

cnnf(Lit,Lit,1).
```

A.3 QC-LeanTaP

QC-LeanTaP is the tableau-based theorem prover for QCL. We almost doubled the rules in comparison with leanT^AP to handle the tableau S- and U-Rules separately. Skolemization is re-introduced at appropriate stages. The crucial rule is, however, the disjunction S-Rule because it is only applicable to literals and it allows focusing. The application of the disjunction S-Rule to literals is ensured by its positioning within the rule system. Focusing is captured by a small rule to select a disjunction of literals to focus over. This is slightly different to the original proof theory of QCL. The disjunction rule used is

$$\frac{\alpha_1 \vee \ldots \vee \alpha_i \vee \ldots \vee \alpha_n}{(\neg\ (\alpha_i \vee \ldots \vee \alpha_n))^* \mid \alpha_1 \vee \ldots \vee \alpha_{i-1}}\ \text{[where } \alpha_1, \ldots, \alpha_n \text{ are literals]}$$

It can be shown that this is a short-hand for the multiple application of the disjunction S-rule. Furthermore, the original disjunction S-rule is a special case of this rule. We have, however, not established the formal proof of soundness and correctness for this variant of the disjunction S-rule yet. Note also that we included the double negation rule because focusing does introduce negation, thus double negations can appear.

We have not established the correctness of this implementation nor any improvements. Thus, we consider it as work in progress. The prover has been tested using range of examples from the publications on QCL.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% qc_leantap
%
```

```
% GProlog
%
% August 2001: Ralph Miarka
%              University of Kent, Canterbury, UK
%              Email: rm17@ukc.ac.uk
%
% Purpose: Lean tableau based prover for Quasi-classical logic
%          by A.Hunter; used in conjunction with cnnf.pl to get
%          the negation normal form
%
% based on \LeanTaP by
%
%   Copyright (C) 1993: Bernhard Beckert & Joachim Posegga
%                       Universitaet Karlsruhe
%                       Email: {beckert|posegga}@ira.uka.de
%   Purpose:  \LeanTaP: tableau-based theorem prover for NNF.
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

:-      include(cnnf).
:-      include(unify).

%%%%%%%%%% BEGIN OF TOPLEVEL PREDICATES

% convert a list of formulae into cnnf
% return a list of cnnfs Fml

cnnffmls([],[]).
cnnffmls([F|Res],[(CNNF,s)|Rem]) :-
        cnnf(F,CNNF),
        write('CNNF = '), write(CNNF),nl,
        cnnffmls(Res,Rem).

prove(Fml,F,VarLim) :-
        cnnffmls(Fml,Res), cnnf(F,CNNF),
        write('CNNFVarLim = '), write(CNNF),nl,
        nonvar(VarLim),!,prove((CNNF,u),Res,[],[],VarLim).

prove(Fml,F,Result) :-
        cnnffmls(Fml,Res), cnnf(F,CNNF),
        write('CNNFResult = '), write(CNNF),nl,
        iterate(VarLim,1,prove((CNNF,u),Res,[],[],VarLim),Result).
```

```
iterate(Current,Current,Goal,Current) :- nl,
        write('Limit = '),
        write(Current),nl,
        Goal.

iterate(VarLim,Current,Goal,Result) :-
        Current1 is Current + 1,
        iterate(VarLim,Current1,Goal,Result).

% -----------------------------------------------------------------
% prove(+(Fml,Sign),+UnExp,+Lits,+FreeV,+VarLim)
%
% succeeds if there is a closed tableau for Fml with not more
% than VarLim free variables on each branch.
%  Fml: list of formulae in negation normal form.
%       syntax: negation: '-', disj: ';', conj: ','
%
%  UnExp:    list of formula not yet expanded
%  Lits:     list of literals on the current branch
%  FreeV:    list of free variables on the current branch
%  VarLim:   max. number of free variables on each branch
%            (controls when backtracking starts and alternate
%            substitutions for closing branches are considered)
%  Sign:     whether S- or U-rules should be used.

% Conjunction S-Rule
prove(((A,B),s),UnExp,Lits,FreeV,VarLim) :- !,
        prove((A,s),[(B,s)|UnExp],Lits,FreeV,VarLim).

% U-Double Negation Rule (actually it also works for the S-Rules)
% needed, because the focus rule can introduce double negation
prove((-(-A),Sign),UnExp,Lits,FreeV,VarLim) :- !,
        prove((A,Sign),UnExp,Lits,FreeV,VarLim).

% Disjunction U-Rule:
prove(((A;B),u),UnExp,Lits,FreeV,VarLim) :- !,
        prove((A,u),[(B,u)|UnExp],Lits,FreeV,VarLim).

% Conjunction U-Rule
prove(((A,B),u),UnExp,Lits,FreeV,VarLim) :- !,
        prove((A,u),UnExp,Lits,FreeV,VarLim),
        prove((B,u),UnExp,Lits,FreeV,VarLim).
```

```
% Disjunction S-Rules:
prove(((A;B),s),UnExp,Lits,FreeV,VarLim) :-
        prove((A,s),UnExp,Lits,FreeV,VarLim),
        prove((B,s),UnExp,Lits,FreeV,VarLim).

% focus
prove(((A;B),s),UnExp,Lits,FreeV,VarLim) :-
        focus((A;B),C,D),
        prove((-(C),u),UnExp,Lits,FreeV,VarLim),
        prove((D,s),UnExp,Lits,FreeV,VarLim).

% Quantification S-rules

% Existential Quantification:
% Skolemize first, then prove skolem. fml
prove((ex(X,Fml),s),UnExp,Lits,FreeV,VarLim) :- !,
        copy_term((X,Fml,FreeV),(Fml,Fml1,FreeV)),
        copy_term((X,Fml1,FreeV),(ex,Fml2,FreeV)),
        prove((Fml2,s),UnExp,Lits,FreeV,VarLim).

prove((all(X,Fml),s),UnExp,Lits,FreeV,VarLim) :- !,
        \+ length(FreeV,VarLim),
        copy_term((X,Fml,FreeV),(X1,Fml1,FreeV)),
        append(UnExp,[all(X,(Fml,s))],UnExp1),
        prove((Fml1,s),UnExp1,Lits,[X1|FreeV],VarLim).

% Quantification U-rules

% Universal Quantification:
% Skolemize first, then prove skolem. fml
prove((all(X,Fml),u),UnExp,Lits,FreeV,VarLim) :- !,
        copy_term((X,Fml,FreeV),(Fml,Fml1,FreeV)),
        copy_term((X,Fml1,FreeV),(ex,Fml2,FreeV)),
        prove((Fml2,u),UnExp,Lits,FreeV,VarLim).

prove((ex(X,Fml),u),UnExp,Lits,FreeV,VarLim) :- !,
        \+ length(FreeV,VarLim),
        copy_term((X,Fml,FreeV),(X1,Fml1,FreeV)),
        append(UnExp,[ex(X,(Fml,u))],UnExp1),
        prove((Fml1,u),UnExp1,Lits,[X1|FreeV],VarLim).

%
prove((Lit,Sign),_,[(L,Si)|Lits],_,_) :-
```

```
        rev(Si,S) ->
        (unify((Lit,Sign),(L,S)); prove((Lit,Sign),[],Lits,_,_)).

prove((Lit,Sign),[Next|UnExp],Lits,FreeV,VarLim) :-
        prove(Next,UnExp,[(Lit,Sign)|Lits],FreeV,VarLim).

% this focus rule is not only for literals
focus((A;B),A,B).
focus((A;B),B,A).

% needed for unification
rev(s,u).
rev(u,s).
```