Issues in Validation and Executability of Formal Specifications in the Z Notation

by

Margaret Mary West

Submitted in accordance with the requirements for the degree of Doctor of Philosophy.

The University of Leeds
School of Computing

October 2002

The candidate confirms that the work submitted is her own and the appropriate credit has been given where reference has been made to the work of others.

Abstract

The work considers issues in the execution of the Z notation in a logic programming language. A subset of Z which is capable of being animated is identified, together with the necessary theoretical foundations for the relationship of Z to its executable form. The thesis also addresses the transition from research results to potentially useful tools. The thesis has 4 major parts:

- Tools Survey: A survey of tools which support the animation of Z is presented and the advantages (and disadvantages) to be gained from an animating system which uses a logic programming language are discussed. Requirements, particularly correctness, are described and discussed and weaknesses in the current tools are identified.

- Correctness – Program Synthesis: If a program can be deduced directly from the specification, then it is partially correct with respect to the specification. This method of obtaining a program from a specification is one form of *logic programming synthesis.* We examine such formal links between a specification (in Z) and an executable form and also some translation techniques for synthesising a logic program from a Z specification. The techniques are illustrated by examples which reveal important shortcomings.

- Translation Rules to Gödel: New techniques for the animation of Z utilising the Gödel logic programming language are presented which circumvent these shortcomings. The techniques are realised via translation rules known as *structure simulation.* Two substantial case studies are examined as proof of concept. These indicate both the coverage of the Z notation by structure simulation and the practicality of the rules.

- Correctness – Abstract Approximation: Published criteria for correctness of an animation are compared and contrasted with the method of Abstract Interpretation (AI). In AI a concrete semantics is related to an approximate one that explicitly exhibits an underlying structure present in the richer concrete structure. In our case, the concrete semantics is Z associated with ZF set theory. The approximate semantics of the execution are the outputs of Z.

  The criteria are applied to a logic programming language (the original was applied to a functional language). Formal arguments are presented which show that the structure simulation rules obey the criteria for correctness. Finally, areas of work which had been omitted by the original authors are presented explicitly.

Acknowledgements

I would first like to thank Paul Mukherjee for his encouragement and supervision during the first years of my studies. Next, I would like to thank Graham Birtwistle for supporting me for the final years leading to writing up of the thesis, and commenting on numerous drafts.

I worked with Lee McCluskey of the University of Huddersfield on the IMPRESS project and we co-authored the papers indicated in **Declarations** on page iii. The project was challenging and (incidentally) provided a useful background to my thesis. Thanks are due to him for many fruitful discussions during this time.

Barry Eaglestone (now at the University of Sheffield) originally suggested the animation of the Z specification by Prolog, and we co-authored the paper indicated on page iii. I would like to thank him, for the subsequent development of the research has lead to some extremely interesting and challenging work over the years.

Staff at Leeds and Huddersfield Universities have provided both help and friendship over the years and I would like to thank them.

Lastly, my family have shown me much patience and understanding over the years of study and I owe them a great debt of gratitude!

Declarations

The following references which I have authored or co-authored were extensively quoted in Chapter 2:

M. M. West and T. L. McCluskey, "The Application of Machine Learning Tools to the Validation of An Air Traffic Control Domain Theory", *International Journal on Artificial Intelligence Tools*, Volume 10 Number 4, (December 2001), pp 613 – 637

T. L. McCluskey and M. M. West, "The Automated Refinement of a Requirements Domain Theory", *Journal of Automated Software Engineering, Special Issue on Inductive Programming*, Volume 8 , Number 2 , (May 2001), pp 193 – 216

Parts of Chapters 3 and 4 have been published in the following articles, which I have authored or co-authored:

M M West and B M Eaglestone, "Software Development: Two Approaches to Animation of Z Specifications Using Prolog ", *Software Engineering Journal*, Volume 7, Number 4, (July 1992) pp 264-276

M M West, "Types and Sets in Gödel and Z ", *ZUM'95 – 9th International Conference of Z User's, September 1995, Limerick, Ireland*, Lecture Notes in Computer Science 967, pp 389–407

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Background

The public demand for increased safety exists alongside the economic imperative for manufacturing and servicing institutions to increase functionality at lower costs and (if possible) with low environmental impact. To achieve the latter, a computer component is frequently included in manufacturing, transport and medical systems. For example, technological improvements to road transport in Europe and the United States have involved the significant employment of computer systems in vehicles, in roadside equipment and in traffic control [116]. The Air Transport Industries make use of computers in, for example, decision support systems for aircraft flight plans [121, 79], 'fly-by-wire' computer systems in aircraft [111] and in collision avoidance on aircraft [53]. Other high integrity systems involving a computer component include financial systems, such as on-line banking [107]. The security of computer systems remains a problem on account of the activities of terrorists and other criminals [77].

There is an on-going account of computer problems which have resulted in risk to the general public [87]. One of these was the Therac 25 linear accelerator which was used for radiation therapy for cancer patients [112]. A computer malfunction caused overdosing of patients resulting in injuries and deaths. Fortunately, problems resulting in serious injury or death are rare, but they underline the fact that we require assurances that any equipment with safety or reliability implications behaves as we expect. However if such a system contains software it is unlikely to have been exhaustively tested because the number of possible states of the system (involving all conceivable paths through the program for all inputs) would be too large. In order to provide sufficient confidence in the safety and reliability of a high integrity system we must rely on an assessment of its development *process*. Development must be via a strict stage-by stage process and be accompanied by tangible output at each stage, that is via a system *life-cycle*. Such systems should be *engineered*.

'Traditional' electro-mechanical systems are developed in two stages, 'design' and 'production' [61]. The 'design stage' consists of *requirements, design development* and the *prototype creation*. Once a prototype is created, the quality of the product is maintained through its production phase and quality control. However, engineering processes for systems which have embedded software differ from electro-mechanical systems in that software does not have a corresponding 'production' phase; all the effort goes into its design. 'Production' of multiple copies is straightforward, for the 'quality' lies in the design. A typical life-cycle for a computer based system consists of

1. Systems Requirements and Analysis;
2. Design;
3. Coding;
4. Testing.

This is somewhat of an over-simplification, for the first few stages of the life-cycle are within a system/engineering environment. Software is always part of a larger system, sometimes known as the *containing system*, which can comprise hardware, humans and frequently an electromechanical system, such as traffic control equipment. The overall system implications are particularly important for high integrity systems, for failure of a system component may ultimately lead to loss of human life or significant financial losses. Thus (for example) safety considerations will be involved in all stages of the life cycle, leading to an enhanced version of the life-cycle called the *safety life cycle* [2]. In this case the successful performance of the end-product depends on adequate Hazard Analysis, and the correct identification and documentation of any associated safety requirements. Subsequent implementation of the system must take account of these safety requirements; they add an extra dimension to system development.

Two activities which take place throughout the development process are *Verification* and *Validation* [44]. Validation is derived from the Latin 'validate' meaning to confirm or ratify. It is concerned with demonstrating the consistency and completeness of a description with respect to the initial ideas of what the system should do. Verification is derived from the Latin 'veritas', meaning truth. It is the comparison of the output of each individual phase of system development with the output of the previous phase. Verification always requires a comparison to be made, the objective being to ensure that the output from the new phase fulfils the requirements specified in the outputs of the previous phase. Informally, validation can be thought of as answering the question "are we building the right system?", whereas verification can be thought of as answering the question "are we building the system right?". *Testing* is used as a part of both of these activities. Acceptance tests are agreed with the user that the system functions in a manner expected. This is an example of a test for validation. Unit testing (of software components) and integration testing (of the software component with the whole) are for the purposes of both verification and validation.

Validation of a computer system involves the establishment of appropriate requirements for the system and this presents problems. Gladden [40] reports experiencing 35% of delivered software as not being used because of the distance between it and the user's concept of the system. A problem with requirements which are written in a natural language is that such a specification can be ambiguous. The same is not true if the specification is *formal*. The use of *formal methods* in the development of computer systems is widely recognised by the software engineering community [33]. Formal methods involves the use of mathematics to model computer systems – just as in other Engineering disciplines. Formal Methods are recommended by many standards bodies concerned with Safety-Critical systems (e.g. IEC 1508 [2]) and for some they are mandatory (e.g. DEFSTAN 00-55 [1]).

Formal methods can be used to aid system development in the following manner:

1. Systems Requirements and Analysis: The functional elements of a computer system are modelled via a formal specification. Formal reasoning is then used to attempt to ensure that the specification has desirable functionality but with no undesired side-effects. A *flaw* or *error* in a formal specification is a part which gives rise to undesired behaviour in the delivered system. However the correspondence between the formal specification and the requirement expressed by the customer cannot be proved and a formal specification can never be said to be *correct*;

2. Design: Refinement is the process of moving from abstract specifications to less abstract or 'concrete' specifications, via some data or operation transformation which allows the behaviour of the abstract system to be replicated by the refined (concrete) specification. Since the abstract and concrete systems are both formal objects, the design can be shown to be correct with respect to the specification. *Proof tools* are used to aid the formal arguments;

3. Coding: The design is further refined to become code and the latter shown to be correct with respect to the design;

4. Testing: a version of the specification is *executed* in order to demonstrate its functionality and detect flaws. In addition a specification can be used to generate tests [30].

DEFSTAN 00-55 recommends mature specification languages including the Z notation [105], a language based on set theory and logic: A particular recommendation of the standard is for the execution of the formal specification. Since many specification languages (including the Z notation) are non-executable, the standard suggests *animation*, the translation of a specification to an executable form. Subsequent animation of the specification then allows its functionality to become apparent. Non-executable specification languages such as Z allows the specifier to concentrate his or her effort on expounding *what* a specification does and are in general easier for humans to understand. In contrast executable languages must contain information on *how* specified functionality should be achieved and the extra information can be distracting to humans.

Animation is also known as *prototyping*, for prototyping is also used in other engineering disciplines to create a working model of the engineered product. The draft standard IEC 1508 states that the aim of prototyping/animation is:

> "To check the feasibility of implementing the system against the given constraints. To communicate the specifier's interpretation of the system to the customer, in order to locate misunderstandings."

The use of the animation and prototyping tools is somewhat akin to conventional testing of software through the use of test cases. In addition the test cases can also be preserved and used for validating the final system [122]; the results of the animation are compared with the results of the final tests.

Animation and proof are *complementary* activities [85]. Formal proof compensates for the fact that tests used for animation can seldom be exhaustive. On the other hand there is no use in seeking a formal proof of a property of a specification if counter examples indicate that the property is not present. In summary, although executable languages are undesirable for a specification, tools which execute a formal specification are necessary as a means of testing. Although a specification cannot be proved as correct, tools which *animate* it can at least make it possible to test to see that a specification conforms to our intuition in specific cases. The requirements and development of animation tools for Z form the subject of this thesis, together with the choice of language for animation.

1.2 Logic Programming Languages for Animation

The executable form of the specification needs to be in a form which "challenges" the specification [95]. The challenge can take the form of executing the specification to ensure that it behaves as expected. If a logic programming language is used for animation purposes [123], the tests can take the form of queries of the "what if" variety: given predicate `Pred(x,y,z,w)` the value(s) of `w` can be established where `x, y, z` are ground or (in principle) the value(s) of `x`, where `y, z, w` are ground. For example, in the IMPRESS and FAROAS projects [121, 79, 80], a tool was developed which successfully supported validation of requirements in an air traffic control (ATC) domain. A formal specification of the domain requirements was animated using the logic programming language Prolog and a test set provided by ATC experts. A set of tools based on machine learning techniques was then used both to trace any flaws in the specification and to correct them.

The Prolog logic programming language comprises the Horn clause subset of first order logic and has been presented as suitable for executing the Z notation. Examples are [119, 70, 67, 69, 68, 29]. This is because of the basis of both languages (Z and Prolog) in first order logic and the 'obvious' translation technique of transforming Z to a set of Horn clauses. However there are difficulties in representing sets in Prolog as formal objects. Further, Prolog is unsound, for many reasons including the existence of extra-logical features such as 'assert' and 'cut' and problems with negation if the literal is not ground. The logic programming language Gödel has a greatly improved declarative semantics compared with Prolog, and supports a set data type. Apart from some well-defined exceptions, a program in Gödel is defined as a theory in first order logic and an implementation must be sound with respect to this semantics [56]. Gödel has a flexible computation rule that can be constrained by user-defined control declarations and ensures that all calls to negative literals are ground. This means that it is possible to present some formal arguments that show the link between the formal specification (in Z) and its executable form and this is mandatory in DEFSTAN 55.

However little work has been done in establishing such a formal link [17]. One method is *program synthesis*, the transformation of Z in its first-order form to a set of clauses, and hence to a logic program. Another is suggested by [17], who present the notion of *abstract approximation*, whereby a concrete semantics (Z) is related to an approximate one (the semantics of the animating language). Thus an animation is 'correct' if the program execution underestimates the concrete Z interpretation. In their paper, the authors have also suggested other requirements for a useful tool in the animation of Z: *coverage* (of the Z grammar), *sophistication* (less likely to go into an infinite loop), and *efficiency* (performance of the animation). However, one requirement for animation which is not identified is the facility of the tool to trace the source of unexpected results from the animator. In Chapter 2 of this thesis we survey some of the tools which have been developed for support of formal specifications. The survey of tools includes those which animate Z specifications and we look at case studies which have been undertaken, as indicative of the coverage of Z provided by the tool. The tools performance and efficiency is also surveyed in so far as this information is available. Correctness issues are also examined. We found that there are weaknesses in all of these areas, the most significant being the lack of attention to correctness. Also, the ability of the answers to identify flaws in the specification is rarely mentioned. Breuer and Bowen present a prototype animator which adheres to their proposed correctness criteria, however it has not been used for any case studies, for it does not allow for any sets apart from the integers. The subject of this thesis is to attempt to address all of these areas, to produce rules for translation of a Z specification which cover the important features of Z and which produces an animation which perform well and can be applied to 'real world' case studies. We would also wish to be able to trace any flaws discovered in the specification after animation. For this we chose Gödel, a logic programming language with an implementation which is sound with respect to the semantics of first order logic and with sets and types.

The reason for this is that we ultimately show that the rules we present (in Chapter 4) do adhere to the correctness criteria of Breuer and Bowen so that any animation is *correct*. The framework and proofs are in Chapter 5.

1.3 Contributions

Our research addresses the lack of formal methods technology, the transition from research results to potentially useful tools. A subset of Z which is capable of being animated is identified, together with the necessary theoretical foundations for the relationship of Z to its executable form. The work presented here examines issues concerned with the execution of the Z notation in a logic programming language. Contributions to research are described as follows:

(1) Requirements – Animation Tools for Z A survey of tools which animate Z is presented in Chapter 2 and the strengths and weaknesses identified. Animation tools are used to discover flaws in formal specifications. However there is then the necessity to trace and subsequently correct these flaws and this is rarely discussed in work concerning the tools. This requirement is evident when examining the achievements of the IMPRESS project which used a logic programming language to animate a formal specification. The strength of the logic program is the potential for tracing the source of unexpected results of the animations;

(2) Correctness – Program Synthesis If a program is deduced directly from the specification, then it is *partially correct* with respect to the specification. This method of obtaining a program from a specification is one form of *logic programming synthesis*. This formal link between a specification (in Z) and its executable form is investigated, together with possible translation techniques which potentially synthesise a logic program from a Z specification. The techniques are illustrated by examples, and reasons provided as to why this is not a suitable method. The work is described in Chapter 3 of the thesis;

(3) Structure Simulation New techniques in the animation of Z utilising the Gödel logic programming language are presented. These techniques are realised via translation rules known as *structure simulation*. Two substantial case studies are examined, indicating the coverage of the Z notation by structure simulation and the practicality of the rules. The work is described in Chapter 4 of the thesis;

(4) Formalisation of Structure Simulation This contribution is presented in Chapter 5 which contains criteria for correctness of the animation previously presented by other authors [17]. The criteria are applied to a logic programming language (whereas the original was applied to a functional language).

Formal arguments are presented which show that the structure simulation rules obey the criteria for correctness. Furthermore, areas of work which had been omitted by the original authors are presented explicitly in this thesis.

1.4 Thesis Structure

The structure of this thesis is as follows:

Chapter 2 presents previous work both by ourselves and by other authors. The chapter involves **Contribution 1** of the work presented in this thesis in the extension of the work of Breuer and Bowen in identifying requirements for animators of Z.

The chapter commences with an illustrative example of two ATC projects (FAROAS and IMPRESS). The example is of a tool which has successfully supported validation of requirements in the air traffic control (ATC) domain. The previous work of other authors includes examples of software tools which support formal notations, and in particular tools which support the Z notation. These are described and discussed and weaknesses in the supporting methodology are identified. In [17], requirements for tools which animate Z are presented. However a missing requirement is the potential of the tool to identify flaws in the specification. The ability to trace and correct flaws in a specification is the main achievement of the IMPRESS project which used a logic programming language to animate a formal specification. The strength of the logic program is the potential for tracing flaws in the specification. The necessity of tracing and subsequently correcting these flaws is rarely discussed in work concerning the tools. The advantages (and disadvantages) which accrue from an animating system which utilises a logic programming language are presented;

Chapter 3 discusses formal program synthesis, the logical derivation of a program from a specification using (for example) rules of inference such as resolution, combined with clausal form transformation. **Contribution 2** of this thesis is the identification of the link between finite set theory and Z, the attempt to exploit this link by using it to synthesise an animation and the reasons for the failure. Z is based on *Zermelo Fraenkel Set Theory* (ZF) [36, 22] and this is compared with the axioms for the *theory of finite sets* [76] in logic programming. The concluding part of the chapter describes the difficulties in proceeding with the method, and why it was abandoned;

In *Chapter 4*, the logic programming language Gödel is briefly described, together with its advantages as an animation language. The manner in which Gödel implements the finite sets of [76] is also presented. **Contribution 3** of this thesis is structure simulation which is the adaptation of Z schema characteristics so that the logical structures of the specification are preserved as far as possible in the resulting model. The rules for the translation of Z via structure simulation are presented using simple examples. *Two* substantial specifications are then translated to Gödel: the *Unix file store* and an *assembler*;

In *Chapter 5*, *abstract interpretation* is explained informally, in which a concrete semantics is related to an approximate one that explicitly exhibits an underlying structure implicitly present in the richer concrete structure. The criteria for a correct animation of Z are based on *abstract approximation*, which itself is based on abstract interpretation. In abstract approximation the approximate semantics of the animations of Z is compared with the richer concrete semantics of Z associated with ZF set theory. The criteria are applied to compare a Z specification, and its animated version using structure simulation, using a subset of Z. Structural induction is used to show that the animated version of the specification underestimates the Z specification. This covers **Contribution 4** for in order for the comparison to be made, the resolution inference rule is framed in a novel way. Furthermore, the theory of finite sets is extended to include the possibility of non-termination when computing a set term;

In *Chapter 6* we present a summary of the preceding chapters and some suggestions for further work, including the automation of the tool. **Contribution 1** concerned the potential of the tool to trace flaws in the specification. The suggestions for further work include the development of tools which exploit the 'what if' capabilities of a logic program by enabling flaws in the specification to be traced and subsequently corrected. A further suggestion was for the adoption of a functional logic program for animation;

Appendices A – D are as follows: *Appendix A* presents the ZF theory of Sets and *Appendix B*, the theory of finite sets. These support **Contribution 2** of this thesis which was the link which was established between the two theories in Chapter 3. *Appendix C* presents the Gödel code which was developed in

Chapter 4 for the animation of Z for the case studies and this supports **Contribution 3**. *Appendix D* presents the proofs of correctness of the animations and is a fuller version of the proofs of *Chapter 4*. This supports **Contribution 4**.

Chapter 2

Review of Related Work

The term "Formal Methods" is defined on the Formal Methods Europe web page[1]

> "Formal methods are mathematical approaches to software and system development which support the rigorous specification, design and verification of computer systems. The use of notations and languages with a defined mathematical meaning enable specifications, that is statements of what the proposed system should do, to be expressed with precision and no ambiguity. The properties of the specifications can be deduced with greater confidence and replayed to the customers, often uncovering facets implicit in the stated requirements which they had not realised. In this way a more complete requirements validation can take place earlier in the life-cycle, with subsequent cost savings."

This chapter provides examples of some commonly used formal methods and specification languages (notations). Support for formal methods includes both validation and verification, and we discuss the reason why validation is so important and present an example of an environment which supports it. The environment was developed in two projects concerning Air Traffic Control (ATC) procedures and is described. In general, formal notations are not executable, although there are some directly executable formal notations Larch [49], Spill [65], ObJ [102], Horn Clauses [27]. This chapter explains why it is preferable for specifications to be

[1] http://www.fmeurope.org/

non-executable. We also explain how it is sometimes possible to execute formal specification notations via a translation to an executable language: such a translation is known as an animation. We provide examples of formal notations, together with generic tools which support them. These notations include 'Z': it is noted that there is a dearth of usable tools which support the animation of the Z formal notation. Examples of the work of other researchers in the animation of Z are then provided and compared. Requirements for animations of Z which have been presented by other authors are extended. The extension draws on the experience of the ATC domain projects and this forms **Contribution 1** of this thesis.

2.1 Tools for Verification and Validation

In Chapter 1 of this thesis we indicated the role of formal methods in all phases of software development. In a survey of the use of tools by commerce and industry [8], the main benefits of the use of formal methods are seen as

- the lack of ambiguity of mathematics compared with natural language in specification and design;
- the potential of mathematics for reasoning and proof.

The participants were asked what they thought were barriers to the use of formal methods and over 50% cited the lack of usable tools. There are tools for *verification* and for *validation* (or both). Verification tools enable a proof that (for example) a refinement of a specification is correct with respect to a specification, or that the code which implements the refinement is also correct. Validation tools aid the specifier in his/her task of specifying a system which satisfies a user's (often informal) requirements. Validation of a *formal* model has problems and advantages: it may be harder for a non-computing professional to understand, and be more detailed than a conventional requirements document [91]. On the other hand, the formality brings with it the opportunity for powerful tool support. The use of formal techniques and the production of a formal model of requirements can be said to produce *enhanced quality* rather than *correctness* [15], for the correspondence between the formal specification and the requirement expressed by the customer cannot be proved; a formal specification can never be said to be correct.

The problems in coordination of large software system projects are detailed in [35], which emphasises the importance of requirements analysis and tracking and the effect a good quality specification has on subsequent design and implementation. In [74], 387 software errors uncovered during the development of Voyager and Galileo space-craft are analysed. This analysis showed that errors in identifying or understanding functional or interface requirements frequently lead to safety-related software errors; this is because safety-related functional faults are particularly related to imprecise or unsystematic specifications. The requirements documentation for a software system is important because it

- provides primary input to design stage;
- may provide a (legally binding) contract between supplier and client;
- provides a baseline against which acceptance tests are carried out.

In the FAROAS[2] project, a formal specification in *many sorted first order logic* (here called *msl*) was established for ATC procedures. A later project which built on the results of the FAROAS project was the IMPRESS[3] project.

The reasons for the choice of *msl* for the specification language included its suitability for the ATC domain and its expressiveness. The choice of formalism took place early in the project and before the *Formal Requirements Engineering Environment (FREE)* was developed. In order to improve the quality and accuracy of this specification a diverse range of validation strategies were used, and this is described in the next subsection for illustration purposes.

2.1.1 An Environment Supporting Validation

The FREE is described in full in [80] and supports requirements written in *msl*. It was utilised as an aid in validating the formalisation of requirements in an air traffic control domain, viz the separation standards for aircraft over the eastern North Atlantic Ocean. Air traffic in airspace over the eastern North Atlantic is controlled by air traffic control centres in Shannon, Ireland and Prestwick, Scotland. It is the responsibility of air traffic control officers to ensure that air traffic in this airspace is separated in accordance with minima laid down by the International Civil Aviation Organisation. Central to the air traffic control task are the processes of *conflict prediction* – the detection of potential separation violations between aircraft flight profiles and *conflict resolution* – the planning of new conflict free flight profiles. The controllers have tool assistance available for their tasks in the form of flight

[2]Formalisation and Animation of Rules for Oceanic Aircraft Separation
[3]IMProving the quality of formal REquirements SpecificationS

data processing system (FDPS) which maintains detailed information about the controlled airspace. A key component of the FDPS is the 'conflict software', that provides assistance for the processes of conflict prediction and resolution. A new FDPS is currently being developed and the FAROAS project was chiefly concerned with requirements capture for the conflict prediction component of the proposed FDPS. The axioms for the domain were written in *msl* and denoted the 'Conflict Prediction Specification' (the *CPS*). The *CPS* was created to contribute towards the requirements specification for a decision support system for air traffic controllers. The functions of the FREE tool included four processes:

- parsing the 'Theoretical Form' (TF) of the formal specification and identifying syntactic errors;
- testing – an 'Execution Form' (EF) of the specification was automatically generated (in the logic programming language Prolog) and batches of expert-derived test cases were used to compare expected and actual results. The raw test data was translated to *msl* via the FREE environment, and hence to Prolog;
- viewing the specification in non-technical form – a 'Validation Form' (VF) of the specification was generated and was used by Air Traffic Control experts for visual inspection;
- reasoning about internal consistency – a reduced minimal set of the specification axioms were shown to be consistent.

A fragment of data in its *msl* and Prolog forms is shown in Figure 2.1. The data comprises part of a segmented flight path of a single aircraft with call sign *XXXX*, where the real call sign and some other details are undisclosed for confidentiality reasons. A non-atomic axiom is given in Figure 2.2, which also contains its executable and validation forms. The axiom defines a temporal relation between two aircraft which at some point are using the same profile track. Variables are universally quantified by default, and represented by capitalised identifiers, and the symbol 'E' is used for existential quantification. Note that the executable form of the *CPS* respects the modular structure of the original, for all or part of each Prolog clause represents an axiom. Also each Prolog clause logically follows from the original axiom in the *CPS*, following [73]. The process was general enough to cover all the axioms of the *CPS* which comprised over 300 non-atomic, quantified *msl* expressions. The initial form of the *CPS* was built up from the existing manual system and other documents supplied by ATC experts. The five validation procedures outlined above were subsequently run in accordance with a test plan [80] and various errors in the initial encoding of the requirements were identified and removed. However in a specification of this complexity the manual forms of validation had a limited (although essential) contribution, and batch testing formed the most dominant part of validation. The validation processes involving execution of a derived prototype were fully automated, and therefore could be easily re-executed after maintenance of the *CPS*. The 'testing' of the specification was achieved by animation. This included

1. testing of 'lower level' axioms. These included 'auxiliary' axioms (which defined ATC terms) and also axioms defining trigonometrical relationships;
2. translating to TF (in *msl*), then to EF (in Prolog) historically generated raw data from pairs of flight profiles. For each pair there is an expert decision as to whether the pair are 'in conflict' or 'conflict free'.

The aircraft XXXX is flying at a constant speed of 0.86 Mach and at a height of 35,000 feet and is planned to enter the first and second segments at 1042hrs and 1122hrs respectively:

```
%% TF of data fragment
"(the_Segment(profile_XXXX, 57 N ; 010 W ; FL 350 ; FL 350 ;
  10 42 GMT day 0, 57 N ; 020 W ; FL 350 ; FL 350 ;
  11 22 GMT day 0, 0.86) belongs_to profile_XXXX)".

%% EF of data fragment
the_Segment(profile_XXXX,fourD_pt(threeD_pt(twoD_pt(lat_N(57),
long_W(10)),fl_range(fl(350),fl(350))),time(10,42,0)),
fourD_pt(threeD_pt(twoD_pt(lat_N(57),long_W(20)),
fl_range(fl(350),fl(350))),time(11,22,0)),0.86)
                      belongs_to profile_XXXX.
```

Figure 2.1: Flight Data and its Prolog Form

Theoretical and Executable Forms of axiom

```
" (Segment1 and Segment2 are_after_a_common_pt_from_which_profile
    _tracks_are_same_thereafter)
   =>
 [ (the_aircraft_on Segment1 precedes_the_aircraft_on Segment2)
 <=>
 E Segment3
  [(Segment3 belongs_to the_Profile_containing(Segment2)) &
    the_entry_2D_pt_of(Segment3) = the_entry_2D_pt_of(Segment1) &
    the_exit_2D_pt_of(Segment3) = the_exit_2D_pt_of(Segment1) &
    (the_entry_Time_of(Segment3) is_later_than the_entry_Time_of(Segment1)) ]] "

the_aircraft_on_segment1_precedes_the_aircraft
             _on_segment2(Segment1,Segment2):-
are_after_a_common_pt_from_which_profile_tracks
_are_same_thereafter(Segment1,Segment2),
    the_Profile_containing(Segment2,Profile1),
    Segment3 belongs_to Profile1,
    the_entry_2D_pt_of(Segment3,Two_D_pt1),
    the_entry_2D_pt_of(Segment1,Two_D_pt2),
    same_2D_pt(Two_D_pt1,Two_D_pt2),
    the_exit_2D_pt_of(Segment3,Two_D_pt3),
    the_exit_2D_pt_of(Segment1,Two_D_pt4),
    same_2D_pt(Two_D_pt3,Two_D_pt4),
    the_entry_Time_of(Segment3,Time1),
    the_entry_Time_of(Segment1,Time2),
    Time1 is_later_than Time2, !.
```

Validation Form of axiom

> *'For any two segments Segment1 and Segment2, in the case where Segment1 and Segment2 occur after a common point from which the tracks of their profiles are the same, we say that the aircraft1 on Segment1 precedes the aircraft2 on Segment2 if there exists a Segment3 in the Profile containing Segment2 such that Segment1 and Segment3 have the same entry and exit points, and aircraft2 enters Segment3 later than aircraft1 enters Segment1.'*

Figure 2.2: An Axiom of the *CPS* and its Executable and Validation Forms

The expert decision for each pair of aircraft is compared with the results of the animation run. If the expert decision is at variance with the prototype's decision it is evident that there are flaws in the specification which need removing. However where tests fail it is still very difficult to identify the faulty or incomplete requirements. For this reason, the FREE tool was later extended (in the IMPRESS project) to incorporate machine learning and theory revision, for example training examples were used which had the effect of inductively altering the executable form of the theory [121, 79]. The fact that this form of the *CPS* closely followed the structure of the original was one reason that these tasks were possible.

2.1.2 Executability of Formal Specifications

The *CPS* described in Section 2.1.1 is written in *msl* which is itself non-executable. The issue of executability is examined by Hoare [58], in which three forms of the greatest common divisor function (gcd) are compared. The first version involves logic, set theory and arithmetic which is (in theory) 'executable' via a search of all proofs. It can be paraphrased:

Greatest Common Divisor: The greatest common divisor z of positive non-zero integers x, y is such that z is the greatest member of the set of numbers satisfying z divides x and z divides y;

Divides: "p divides q" means "there exists a positive whole number w such that $pw = q$";

Greatest of a Set: "p is the greatest member of a set S" means "p is in S and no member of S is strictly greater than p".

The first version is refined and translated to the logic programming language Prolog to produce the second version. However the resulting Prolog program would not terminate because of problems with negation. The third version (which does terminate) is a functional program:

$$
\begin{aligned}
&gcd(x, y) = x \ \ if \ \ x = y;\\
&gcd(x, y) = gcd(x - y, y) \ \ if \ \ x > y;\\
&gcd(x, y) = gcd(y, x) \ \ if \ \ x < y.
\end{aligned}
$$

The gcd of a number pair is calculated as the result of a terminating sequence of substitutions and this is proved by Hoare. However as can be seen, the ease of understanding of the latter is less good than the original version. In other words, clarity of requirements decreases with executability of the gcd function[4]. The inference of this paper is that clarity should not be sacrificed for executability. Hoare suggests that the high level language chosen for program implementation should be as close as possible to the original specification. In addition it should not be so inefficient that 'program tricks' are necessary to ensure execution. Hoare also suggests the use of such an implementation for a rapid check on the adequacy of a specification.

Some notations are themselves executable and examples are Larch [49], Spill [65], ObJ [102], Horn Clauses [27]. Examples of non-executable notations are the Z

[4] However in *current* declarative languages such as Gödel or Haskell, it is possible to write a gcd function which looks very similar to its definition.

notation [105], the Vienna Development Method-SL [64], the B Abstract Machine Notation [5], CSP [57], CCS [82], LOTOS [14]. There are conflicting points of view about the advisability of a notation being executable. The opposing views are in [52] and [38]. The former argues *against* executable specifications in that implementation details needed to make the specification executable are undesirable. This is so because there is a tendency for over-specification because of the need to follow algorithms in the specification. The authors provide several examples to illustrate. A possible assignment of values of two set-valued variables $s1, s2$ can be expressed :

$$s1, s2 := s1 \cap s2, s1 \cup s2.$$

This simple, *clear* expression contrasts with the many procedures involving linked lists required to specify the same assignment in an executable language. The authors also state their opinion that proving general theorems about properties of the specification is a more powerful tool for validation, and this procedure is more difficult with an executable specification. In order to verify that an implementation meets its specification it is necessary for a comparison to be made but it is hard to match an executable specification against an implementation. However the authors end the paper by distinguishing between an executable specification and executable prototype of the specification, in that an executable prototype is acceptable but typically has implementation detail.

The argument *for* executable specifications in [38] is that when a specification is executed its behaviour can be demonstrated to potential users. Thus validation problems are tackled before design decisions are made. An example is given of LSL, a logic programming language which can cope with sets. The authors provide a translation of $\cup$ and $\cap$ to LSL, demonstrating that logic programming languages are *declarative*: they are capable of stating *what* is to be computed, and not dwelling too much on *how*. In response to the argument that executable specifications cannot be reasoned about, they reply that reasoning is limited in its application to validation.

A full discussion and summary of the two opposing views is contained in [42]. It is apparent that although the authors of the two papers disagree on many issues, there is a consensus:

1. clarity of specification is of great importance;

2. the ability to be able to reason about a specification is an important validation task;

3. the potential is required for showing that a refinement agrees with the original specification;

4. the ability is also required to demonstrate the functionality of a formal specification to a customer or user.

In order to achieve consistency between (1) – (4) above, the solution is to utilise a non-executable formal specification for requirements then translate to an executable form to demonstrate and to test its functionality as was described in Section 2.1.1 with the FAROAS and IMPRESS projects.

2.2 Formal Notations, Methods and Tools

2.2.1 Formal Notations

The survey described in [8] lists the most popular notations (as used by industries) as the Z notation [105], the Vienna Development Method Specification Language (VDM-SL) [64], and the B Abstract Machine Notation (B AMN) [5]. These are all *model-based*: they model in terms of mathematical structures such as sets, lists, sequences, mappings and are based on states of the system. They contrast with *algebraic* notations which are based on equational logic/algebra (examples are OBJ, Larch [39, 48]) and with *process algebras* which deal with Concurrent and Communicating systems (examples are CSP [57], CCS [82], LOTOS [14]).

Of the model-based notations, Z is strong on modularity but weak on development support. In contrast B AMN is fairly strong on both modularity and development support (as it is associated with a *method*). However B AMN is more restrictive than Z. VDM-SL is also associated with a method and is strong on development support. However it is weak on modularity. The next section describes some model-based specification notations and the tools which support them. The notations are restricted to the most popular of the survey in [8].

2.2.2 Software Tools which support Formal Methods

2.2.2.1 The Vienna Development Method

The Vienna Development Method (VDM) [64] is a collection of techniques for the formal specification and development of computing systems consisting of (i) a specification language called VDM-SL, (ii) rules for data and operation refinement to the level of code, and (iii) a proof theory allowing for rigorous arguments involving both the properties of specified systems and the correctness of design decisions. The proof theory involves a form of logic called the *logic of partial functions*. The refinement of a specification into a design is called *reification* in VDM and there are proof obligations as to whether the design correctly reifies the specification. VDM arose from research on formal semantics of programming languages at IBM's Vienna Laboratory in the sixties and seventies [64]. VDM-SL has been standardised by ISO [89]; it is a model-oriented specification language which also provides basic types like natural numbers, characters, and type constructors like sets and maps.

Tools which support VDM include the *IFAD VDM-SL Toolbox* [45, 46], and *Mural* [83]. VDM has also been translated to Prolog for the purposes of animation [12]. The IFAD VDM-SL Toolbox [45] is a set of tools that supports the development of formal specifications using the ISO VDM-SL standard. Although standard VDM-SL does not support modules, the Toolbox itself supports a structuring mechanism based on the concept of modules. The VDM-SL Toolbox supports syntax and type checking of the specification and also the animation of a subset of the specification. VDM++ is an object-oriented extension of Standard VDM-SL which includes concurrency [34] and is supported by the IFAD VDM++ Toolbox.

Two case studies which utilised the IFAD VDM-SL Toolbox are presented in [85]. The case studies involved two specifications: the Single Transferable Vote system (STV) and the NewSpeak language specification. The Toolbox analysis and animation facilities were used to validate the specifications and revealed errors. For example, type analysis exposed the attempted use of subtraction with non-numeric arguments. A subset of the VDM specification of the STV system was executed via the utilisation of 13 'standard' test cases. This revealed that a set had been used inappropriately in part of the specification, as opposed to a sequence, which would have been correct. There is a discussion in the paper as to whether or not animation and proof complement each other or overlap. The conclusion is that *formal* proof and animation complement each other, whereas *informal* proof (a proof outline) and animation can overlap. Several errors in the STV specification were identified by both informal proof and by animation. However an error detected by formal proof would have evaded detection by animation. On the other hand formal proof is difficult and time consuming and time can be saved in attempting to prove a hypothesis if counter examples indicate it to be false.

Proof support for VDM is described in [10]. For example, the Mural proof tool was used to aid the development of a VDM specification of a memory model (shared memory synchronisation for parallel processing). The tool automatically translated the VDM specification to its corresponding theory (formal language and set of inference rules). The proof obligations which were also generated were proved interactively with the tool. Although the tool will aid the user with suggestions as to the verification of a line of proof, the thinking comes from the user for the set of inference rules used by Mural is visible. The proofs were an aid to a simpler specification, in that the intuition required by the user in discharging the proof led to a clearer specification.

2.2.2.2 The B Abstract Machine Notation

B AMN [5] is state based and utilises the concept of a 'state machine'. A central feature of B AMN is its module structuring capability; a B AMN specification is composed of several 'abstract machines'. A typical abstract machine state comprises several variables which are constrained by a machine invariant and initialised. Operations on the state contain explicit preconditions; the postconditions are expressed as 'generalised substitutions', giving the language a 'program-like feel'. Machine composition is achieved by (for example) the INCLUDES mechanism which allows one machine to alter the data of another. B AMN is associated with 'the B method' [98, 125], and there are rules whereby machines can be refined and subsequently implemented by code. Support is provided for B AMN and the B method by the *B-Toolkit*:

- document preparation: it allows the editing of source files, translation to a marked-up form (i.e. latex) and viewing and printing of the documents;
- analyser: it allows syntax checking and type checking of the source files. Machines can *call* other machines and each called machine is also checked;
- animation: it allows a check that the machine does what is required and it is for *validation*;
- proof obligation generation: this generates proof obligations (theorems) for checking consistency. For example a proof obligation is generated which checks that the invariant is obeyed *initially*. If a set is to be a certain type, then this must be true initially, and the condition must remain true after each operation;
- proof: the proof obligations can be discharged – the theorems are proved using the automatic, or interactive prover;

- refinement leading to code: refinement is the process of moving from abstract specifications to less abstract specifications, via some data or operation transformation which allows the behaviour of the abstract system to be 'simulated' by the more refined system. Thus, all behaviour of the abstract specification is replicated by the refined specification.

The proof engine of the B-Toolkit uses backwards and forwards inference, and rewriting is treated as a special form of backwards inference and is used for animation. The B User Trials project which explored the application of the B AMN method of formal software development is presented in [11] and describes six case studies. There is a comparison of B AMN with both VDM and Z and some shortcomings of the B AMN notation are also described. For example the B AMN notation does not lend itself as easily as VDM and Z to capturing the complex data types required by the case studies. Also (compared with Z) the mechanism for composing abstract machines is restrictive, for only one machine can include a particular machine. The restriction is to facilitate proof. However, although the automatic theorem prover is a considerable aid, it was found that those theorems which were not provable automatically were more difficult to prove interactively than in, say, Mural. This was because, and unlike Mural, in that version of the Toolkit the proof rules were not visible to the user[5].

However, in spite of these limitations, the B AMN notation and Toolkit have been found to speed up the development of an avionics subsystem. A case study [113] describes the MIST and SPECTRUM projects. The aim of the MIST project was to compare the application of the B-Method to the application of conventional software engineering techniques. The software functions under consideration were coded (and tested) in Ada using parallel teams, one using a formal approach and the other a conventional one and it was found that the effort required by the former team was about 80% of that of the latter.

2.2.2.3 The Z Notation

The initial work in the development of the Z notation was at Oxford University in the early 1980's. Its designers' intention was for the major part of the notation to be 'conventional' first order logic and set theory. However the notation has a modular form: the data types, constraints on data types and the means of updating data types are grouped into *schema* which are composed using *schema calculus*. The no-

[5]In later versions of the Toolkit the proof rules were made visible.

tation (like B AMN) supports the concept of a state and processes on it, but it is not obligatory to structure Z specifications in this way. When a specification is structured by means of states and associated processes, the proof obligations are similar to those of B, that the state can exist initially, and that its invariant is preserved after the operations on it. The notation is in the process of standardisation [90] and is described more fully in Chapter 3. Supported tools for Z include the Z/EVES proof tool [81], the FUZZ typechecker [105] and the Formaliser interactive editor and type checker [37]. The HOL theorem proving system has been used for proof support [16]. A number of tools exist which integrate Z with a semi-formal method. Examples are the SAZ system [75], which extracts functional information (which is subsequently formalised) from SSADM products and [19], which outputs Z schemas from data flow diagrams and entity relationship models. A recent development is the 'Community Z Tools' (CZT) project[6] which aims to build an integrated set of tools for Z which are to be made available over the world wide web.
Z/EVES: The current version of Z/EVES includes a graphical user interface, syntax and type checker, and an interactive proof tool. In addition to checking whether or not the 'standard' proof obligations can be discharged, the proof checker ascertains the status of partial function variables, that no schema relies on a result outside the function domain. This is because Z is based on first order logic where the status of undefined terms is ambiguous [109]. This is not the case with VDM, whose proof theory is based on the logic of partial functions.

Z/EVES can be used to generate *guards* (extra conditions) which ensure that all elements of a specification are fully defined. This is known as *domain checking*. Case studies which utilise Z/EVES are described in [26, 96]. The first study concerns the Sliding Window Protocol, a model of message dispatch and reception. One of the purposes of the protocol is the detection of messages which are sent but subsequently lost and then resent. Analysis of the protocol resulted in the division of the protocol into *sender, receiver, the link for sending messages, the link for acknowledging messages*. Each process has an initial state and the first task of analysis was to discharge a proof obligation that for each process, its initial state was valid. Operations of the system which potentially altered the system state included *sending a new message*, *re-sending a previous message*, *receiving a message* etc. A further task was to attempt to modify the specification and subsequently attempt to re-prove the theorem that the invariant still held. The surprising result was that this theorem was still provable. This led to the further analysis – the result was that

[6]See URL http://web.comlab.ox.ac.uk/oucl/work/andrew.martin/CZT/.

Notation	Proof and Type Checking Tools	Animation	Support for System Development
Z	Z/EVES, FUZZ Formaliser	No Supported Tool	Refinement via Proof tools
VDM	Mural	VDM-SL Toolbox	Mural
B AMN	B-Toolkit	B-Toolkit	B-Toolkit

Table 2.1: Formal Notations and Supporting Tools

the modification resulted in the safety property being contravened and *was* therefore significant. The second case study describes how Z/EVES can be used for domain checking [96]. This paper describes how many Z specifications have been examined and those with unprovable guards were found to be in the majority. An example is the specification of a file editor in [60], where no account is taken that a file might be empty.

Table 2.1 presents a comparison of the formal notations just described and the support tools available. In the case of the Z notation the Z/EVES tool just described supports type checking and proof and is available to the public; the Formaliser interactive editor supports type checking and is available commercially. Animation is available in B AMN via the B-Toolkit and in VDM via the Toolbox. The last column refers to support for specification refinement and reification towards design and code. This is available in VDM and in B AMN and (to a lesser extent) in Z via proof tools. However there is a dearth of supported animation tools for Z. Such tools as do exist, together with research work in their development is presented briefly in the next subsection.

2.2.3 Animation of Z (Current Tools)

2.2.3.1 Choice of Language

Early work in the animation of Z is presented in [108] which describes a line-by-line translation of an Access Control System. The animation was via the Prolog programming language. Since then there have been many tools, both proposed and actual, for animation of Z. Breuer and Bowen [17], have identified requirements for an animation (or *interpretation* of Z as it is sometimes termed). They include *coverage* of Z, *sophistication* (less likely to go into an infinite loop), *efficiency* (performance) and *correctness*. The latter requirement arises because the animation and the specification are both formal objects. The authors make the point that many animation techniques focus on the first four concepts and have ignored the correctness of the particular technique. They suggest that an animation be considered 'correct' if it accords with set theoretic considerations. They further suggest that a non-terminating computation is 'correct' in the sense that if an answer is never obtained it can never be incorrect. Inevitably, some techniques which have good terminating properties will fail to cover some parts of Z and there is a trade-off between some of these requirements. One of the criteria for correctness is that the programming language which animates should reflect the recursive structure of Z: the functional language of Miranda is chosen as an example. The declarative nature of Z means that the most natural choices for programming languages for animation of Z are also declarative as opposed to *imperative* (procedural).

Imperative (or procedural) languages specify explicit sequences of steps to follow to produce a result, while declarative languages describe relationships between variables in terms of functions or inference rules. An example of animation of Z by a procedural language is the ZANS tool [62], which utilises the C programming language. A further proposal for animation by a procedural language describes the development of a new computation model for Z [43], where the specification is first translated to a set based expression language (the μZ calculus) and then to Java. Logic programming languages are the principal form of relational language chosen for the animation of Z and examples are [29, 32, 119, 117, 55]. Functional languages include [63, 31, 17, 41, 115, 101, 100, 114].

Logic programming languages involve relations which require the input of one or more of its parameters and will return as output combinations of alternative value(s) of its other parameters. A function, in principle, returns a single value, however the value might be a tuple. Functions can return a partial answer, e.g. part of a list, thus allowing an unbounded list to represent an infinite set. Logic programming does not allow this, but does allow backtracking to provide the values one at a time. It is claimed that a functional logic language for animation can take advantage of both the logic and functional paradigms and an example is Mercury [124]. The following describes some examples of animation techniques which typify the functional approach.

2.2.3.2 Functional Languages

Examples of animation using functional languages include Miranda [31, 17, 63, 3], Haskell [41, 101, 100, 114], and a Lisp like language based on a subset of Z called Z^{--} [115]. The translations to Haskell differ: in [41] the Z specification is translated without alteration and set theory is provided separately by a module called ZPrelude.

The paper presents equivalences between selected operations in Z and operations in Haskell. However not all operations in Z have a straightforward Haskell equivalent. Moreover before translation takes place, each expression involving schema calculus must be expanded out into a full schema. The JaZA animator [114] takes a similar approach to schema calculus and also makes a particular point of how it deals with undefined terms: the paper includes a comparison of how other work treats undefinedness. In [101, 100], the Z specifications are refined to FunZ, an extension of Haskell with a Z like flavour. The developer is able to prove properties about the system design using either the Z notation or Haskell. Advantages of the method as perceived by the authors are that the FunZ document serves as a record of the design process and the user may prove the final implementation correct with respect to the initial specification.

The translation from Z to Miranda described in [63] is a straightforward translation of the given Z, whereas the translation described in [3] refines the Z specification first in a manner based on model refinement. The refinement is checked with correctness rules. The translation [17] also deals with correctness – however in a converse manner. The Z specification is proved to be a refinement of the resulting Miranda code. The proof method is known as *abstract approximation* and based on the concept of *abstract interpretation*. There are differences in the two papers in how given sets are modelled, and how schema calculus is treated. In the former paper given sets are represented by Miranda types, such as $STRING$, whereas in the latter, integers only are represented. Also, in [3], schema calculus is modelled, whereas in [17], all schema expressions have to be unfolded before animation.

Z^{--} [115] is a Z-like programming language which is a refinement of the Z notation. Thus specifications in Z must be refined before the translation can take place. The prototype Z^{--} interpreter described in the paper is written in Lisp. The design intention is for the interpreter to be correct, cover all of Z, be efficient and be sophisticated enough to cover all directly executable constructions. Sets are of two kinds; they can *either* be finite *or* be potentially infinite. The latter are modelled by higher order functions.

In [32] there is a table which compares and contrasts the different manner in which features of Z are modelled in Prolog animators. Table 2.2 presents our summary of properties of animation tools for Z in functional languages using similar features to those of [32]. Table 2.2 indicates which tools have been implemented, how sets are represented, whether the Z specification is first refined and which tool directly models schema connectives in the animation. The tables roughly replicate the criteria of [17] for animation. 'Set implementation' and 'schema calculus' are chosen as a rough measure of coverage of Z for most of the other common constructs of Z are covered in the animation tools selected in the tables. The modelling of schema calculus also has other implications, as will be seen. There is also an indication in the table of whether published case studies have been animated. In compliance with the views Breuer and Bowen have expressed we have added a 'Correctness Issues' feature to the table which indicates whether correctness issues are discussed or directly addressed.

Tool	Goodman (1995)	Z $^{--}$	Breuer, Bowen (1993)	Abdallah et. al. (2000)
Implemented	Yes	Yes	Library obtainable	No
Sets	Infinite	Infinite	Infinite	Finite
Refinement of Z	No	Yes	No	Yes
Schema Connectives	No	Yes	No	Yes
Correctness Issues	Proof Method outlined	Discussed	Correctness Criteria	Correctness Criteria
Case Studies	'small to medium sized'	Mathematical toolkit in Z	No	Birthday Book

Table 2.2: Animation Tools for Z in Functional Languages

2.2.3.3 Logic Programming Languages

The SuZan project described in [67, 68, 69, 29], utilised 'pure' Prolog to try to ensure that the Prolog correctly represents the Z. In order to find possible answers to queries a 'generate and test' method is used, wherein many possible answers are generated, and each is tested for compliance with the logical relation. Efficiency is improved by utilising correctness preserving program re-ordering code and the removal of duplicate clauses. There is a large library of Prolog for set operators, but for schema calculus there is only a suggested implementation. The telephone network system described in [51] is animated in Prolog. Animation of the network revealed some unexpected results, but the source of this specification flaw is not identified in the paper, nor is any mention made of any trace mechanism to seek out the flaw. The EZ project [32] uses similar rules of translation to the SuZan project. However greater coverage of Z is achieved by the use of AI search systems which allow for greater efficiency. Answers to queries also use a generate and test strategy. The paper provides a comparison between EZ and other methods.

The approaches just described, of animations of Z via logic programs contrast with West [119, 117]. In both of the latter publications, *specific* examples of variables are provided to interrogate the specification; the philosophy is to *test* a set of variables and possibly obtain remaining variable(s) to see if they satisfy particular requirements. This translation technique was utilised in a case study which identified safety requirements for Pelican Crossing equipment, described in [118] and more briefly in [120]. The equipment had been operating in a dangerous manner, for the audible signal which indicates to blind and partially sighted people when it is safe to cross had been sounding when the vehicle and pedestrian lights were not operative. In the case study the states of the system are related to the formal safety requirements via proof outline(s) and animation. The papers and case study demonstrate the advantage of the utilisation of a logic programming language with the philosophy of 'testing' variables using some test strategy. The same technique is adopted by PiZA [55]. However in PiZA, and unlike the animation described in [119], variable declarations are not converted to Prolog.

Table 2.3 summarises approaches to animation in Prolog of specifications in the Z notation. It additionally indicates the way in which the predicates are queried to provide the execution, viz the manner of constraint satisfaction, a property specific to logic programs.

Tool	SuZan	EZ	West (1992, 1995)	PiZA
Implemented	Yes	Yes	No	Yes
Sets	Finite	Finite	Finite	Finite
Refinement of Z	No	No	No	No
Schema Connectives	Yes	Yes	Yes	No
Constraint Satisfaction	Generate and Test	Generate and Test	Individual Tests	Individual Tests
Correctness Issues	Pure Prolog	Pure Prolog	Discussed	Not Discussed
Case Studies	Telephone System, vending machine, lift	Convex Hull	Assembler, Pelican Controller	300 page Malpas translator

Table 2.3: Animation Tools for Z in Prolog

2.2.4 Animation of Z (Requirements)

2.2.4.1 Features of Z Animators

Features of Z animators have been presented in Tables 2.2 and 2.3. A further feature of animation tools is described in [114], the ability of the tool to deal with undefined terms. (We deal with this in Chapter 5 as a separate issue.) Breuer and Bowen in [17] have also identified requirements for animation of Z. In this section we link the requirements of the latter with the features presented in the tables to identify weaknesses in the tools and their methodology. We also extend the requirements of Breuer and Bowen in a manner indicated. We first present a discussion, then a summary in Figure 2.2.4.3.

An important requirement for an animation is the ability to trace the flaw in the specification after a test has given an unexpected result. This property of an animation was used to great effect in the ATC projects described in Section 2.1.1. For traceability, the animation should reflect the modularity of Z. This has implications (for example) for schema calculus for in that case flaws in a specification are easier to trace. It can be seen from the tables that each of the Prolog animations models schema calculus but for two of the functional languages, schema calculus expressions have to be unfolded prior to animation. However all the functional languages implement infinite sets, but the Prolog animations implement finite sets only. Refinement is used to make the Z specifications executable, where 'refinement first' is designed to achieve greater efficiency of an animation tool. Similarly, for logic programs which attempt to satisfy a constraint, the table differentiates between different testing methods. In contrast, the ability to animate published case studies is seen as a measure of both sophistication and coverage of the tool in question. It is notable that PiZA was used for validation of a very large case study. An implementation of a translator capable of translating a subset of the rules of [119] was developed as an undergraduate project in [13]. However translation from Z to Prolog was mainly by hand so case studies other than those indicated were never tried. For other (implemented) tools it is not clear whether or not an attempt was made to animate other specifications, nor is the result of any such attempt mentioned.

2.2.4.2 Test Strategy

For animation purposes it is necessary for test data to be translated to a suitable form (as in the case of flight data in Section 2.1). A further requirement for animation is a suitable test strategy, such as that adapted for software code. There has been a deal of research concerning software testing [86, 94, 93] and the following is a summary of some of the issues. Computer-based systems are (typically) a mix of application software, system software, hardware, data and people, and may well control an electro-mechanical system. Testing the interface(s) of the software with its containing system is termed *black-box* or *functional* testing, where tests are derived from the functional specification without regard to the structure of the software. In contrast, *white-box* or *structural* testing examines the structure of the program. The drawback of black-box testing is that it concentrates on desired functionality being present and does not deal with the fact that *un*desired functionality may also be present. In addition black-box tests may succeed for the wrong reasons. It is thus necessary to investigate the structure of the program (and underlying design documentation) to exercise the code as fully as possible.

Black-box tests are for validation and would normally include acceptance tests supplied by the customer/user of the implemented system. White-box testing addresses both validation and verification in the sense that it asks

- questions about the structure of the system: "are we building the system right?";
- questions about undesired functionality: "are we building the right system?".

Black-box testing is designed to include tests which check

- each software function;
- non-overlapping classes of input which are treated identically (equivalence classes);
- the boundary values of the equivalence classes for data input;
- how the software will cope with invalid input;
- input with safety or integrity implications;
- any initial and terminal state(s).

The latter does not consider the structure of the software and this contrasts with white-box testing which focuses on program structure. *Unit* testing forms an important part of this, where lowest level modules are tested first. This is followed by *integration* testing which is a check of how the module integrates with the rest of the system. Thus a module is called from all other directly linked modules and each module (unit) should be tested regarding

- the module interface;
- local and global data structure;
- important execution paths;
- error handling paths/safety requirements;
- boundary conditions.

Tools which generate test cases from state based formal notations have been proposed by Dick and Faivre [30]. VDM-SL specifications are provided for illustrative purposes and test generation includes

1. partition analysis - the precondition for each operation is transformed so that disjoint partitions of state values become apparent;
2. test sequencing – partitions of state values are used to construct a Finite State Automaton and paths through it which cover all required tests;
3. the generation of input values for validation of the implementation. These are obtained from constraints in the specification.

The method for test generation outlined above covers both black-box and white-box testing. Test domains can correspond either to abstract specifications or (with refinement) to concrete implementations. The former leads to validation of the specification and the latter investigates the structure of the implementation.

Treharne et. al. [113] also use boundary value analysis and equivalence partitioning to generate tests from a formal specification written in B AMN. The formal specification is then refined and eventually implemented in a development supported by the B-Toolkit. The test cases (which are in an abstract form) are themselves refined to form test cases for the implementation. The PROST-Objects project [106] has developed a method for formally specifying tests and describes similar techniques which are applied to Z specifications.

2.2.4.3 Weaknesses in Existing Tools

The tables and supporting literature indicate weaknesses in existing animation tools for Z and a summary of these weaknesses are as follows:

1. There is a lack of methods which can cope with published 'real' case studies. This is particularly so for functional languages. Exceptions are mainly confined to animation in logic programming languages such as SuZan [29], EZ [32], and West [119, 117]. However there is little detail about the case studies animated by EZ in [32], and little detail as to the performance of SuZan in [29]. However the authors of [17] report their experience in using SuZan that it took "several minutes to insert a virtual coin into a virtual vending machine". (It is fair to point out that computers have improved their execution times greatly since the tool was first developed.);

2. Correctness issues are rarely discussed in any of the literature, the main exception being [17], in which correctness criteria are presented in detail. They are briefly discussed in [119, 117] and [29] where a version of 'pure' Prolog is used in order not to introduce non-declarative features;

3. Animation is a form of testing and strategies for animation can be compared with those for testing software outlined in Section 2.2.4.2. It is important that the source of an unexpected result can be traced in the animation but this property is not discussed in any detail in any of the literature supporting the tools from Tables 2.2 and 2.3. It is dealt with briefly in [119, 120], where it is mentioned that a logic program can be *traced* to discover and correct the origin of the flaw. If schema calculus is implemented, this makes it easier to trace flaws. Any process which distances an animation from its specification also makes it harder to trace flaws. Refinement of specifications before animation is designed to improve executability. However it is likely that it will equally make flaws harder to trace.

2.3 Summary

We have identified in this chapter some advantages in the use of formal methods in the development of software systems, for example the lack of ambiguity of mathematics compared with natural language, and that mathematics has the potential for reasoning and proof. However the perceived difficulty of the customer and/or developer in understanding mathematics can present problems for the validation of the requirements specification document, which provides primary input to design. Executing a formal specification to demonstrate its functionality is one solution.

The arguments for and against executability of such a specification have been set out, and the conclusion was that the clarity of the specification should supersede its executability, and furthermore the ability to be able to reason about a specification is of prime importance. A solution is to specify formally using a non-executable language, then *animate* or *interpret* the specification using a programming language. In other words, *proof* and *animation* can be used as complementary methods of validation. Work was then presented by researchers in the animation of Z using functional and logic programming languages. These declarative languages were deemed most suitable for animation since they are more capable of capturing the declarative aspects of Z.

We argue that there are advantages to be gained from an animating system which utilises a logic programming language, as opposed to a functional language for executing Z. Functional programs have the facility for infinite set implementation, but these present difficulties for logic programs. However logic programs enable a "what" approach of specification to be modelled via constraint predicates. Queries of the "what if" variety can then be posed: given predicate `Pred(x,y,z,w)` the value(s) of `w` can be established where `x, y, z` are ground or (in principle) the value(s) of `x`, where `y, z, w` are ground. This capability (we feel) outweighs the inability to use infinite sets. Furthermore, the B-Toolkit implements only finite sets in its animations. *Animation* tests specific examples (as for the ATC case studies), while *proof* mainly involves generalities, and may involve infinite sets.

After a specification is animated, it will be necessary to be able to trace the sources of any discrepancies between real and expected results. Therefore, the closer the animation to the specification is the easier it will be able to trace and correct any flaws. This was the case with the *CPS* (ATC domain specification), described in Section 2.1.1, where the structure of the *CPS* was reflected in its Prolog animation. Therefore we feel that a specification ought not to be refined before implementation, for refinement adds extra detail. Also (if possible) the modular form of the specification in schema calculus and referencing ought to be retained. The requirement that it ought to be possible to trace the source of an unexpected result of an animation forms **Contribution 1** of this thesis.

The weaknesses of current animation tools have been indicated, and these particularly include the lack of tools suitable for 'real' case studies, and the lack of attention to 'correctness' of the tools. Although some of the animation tools deal with case-studies, none of these are known to be correct. Conversely the prototype animator of [17] is correct but has not been used with any case studies. We address this lack, by presenting the rule base for a tool which has the potential to animate real case studies, and furthermore is correct. Our approach to the animation of Z via a logic programming language constitutes the major part of the next three chapters. Thus since the proposed animation involves finite sets, Chapter 3 outlines the semantics of the Z notation and compares this with the theory of finite sets in logic programming. The *CPS* utilised a form of logic program synthesis to generate the animation, and Chapter 3 will also explore the possibility of logic program synthesis to obtain a logic program from a Z specification. This is with a view to obtaining a correct animation of a Z specification via logic program synthesis. Gödel is a logic programming language which involves types and sets and Chapter 4 examines three other requirements for animation tools, first presented in [117], the *practical* ones of coverage, efficiency and sophistication and presents two case studies which animate Z in the logic programming language, Gödel. We also examine a possible test strategy for animation which can be compared with strategies for testing software. Chapter 5 explores a different approach to correctness, that described in [17], where an animation is correct if it abstracts a specification, i.e. does not contain any unwanted extra information. Some issues concerning undefinedness will also be discussed in that chapter.

Chapter 3

Correctness – Program Synthesis

This chapter describes methods for deriving a logic program from a specification using logic program synthesis [28, 59] and is an updated version of West and Eaglestone [119]. A sound technique for Z to logic program translation must be based on some correspondence between the two underlying theories. The semantics of the Z notation is outlined and differences noted with the *Zermelo Fraenkel Set Theory* (ZF) on which it is based [36, 22]. A simple example of a specification in Z is provided for illustrative purposes. Key axioms for the *theory of finite sets* [76] in logic programming are presented (in Appendix B). The transformation of standard logic to its clausal form and hence to a logic program is explained next and an attempt is made to transform set theory axioms to their clausal form. The concluding part of the chapter describes the difficulties in proceeding with the method, and why it was abandoned. The link between the two formalisms forms **Contribution 2** of this thesis.

3.1 The Z Notation

The Z notation is described fully in [105, 103] and is outlined here. The reduced set of ZF axioms on which Z is based, known as ZF-without-replacement, is presented in Appendix A. In contrast to ZF theory where sets are built from the empty set $\varnothing$, the sets of the Z notation are *typed*, and a typed set is a *carrier* of a type. The notion of typing is used to describe and name sets which are required by a particular specification. As a consequence, in the Z notation the empty set is not unique, for a different empty set characterises each type.

The semantics of Z is described by means of a 'world' of sets W in which the set-theoretic operations of Z can take place [103]. The sets of W are the specification types, and each type must be disjoint. The replacement axiom is omitted from Z because of consistency problems: this reduced set of axioms is known to have models [36]. For each of the ZF axioms there is a corresponding operation in Z. It is necessary for W to be closed under the operations of the specification, and this will be so for ZF-without-replacement. The set axioms chosen for Z are adequate for our purposes as they are sufficient to model all the sets we might require, including the set of real numbers.

The 'sets as types' feature means that formulae in ZF containing expressions such as

$$\forall\, x \bullet \mathcal{A}(x)$$

where $\mathcal{A}(x)$ is a wff appears (in Z) as

$$\forall\, x : \tau \bullet \mathcal{A}(x)$$

where τ is some set valued object.

Ordered pairs, such as (a, b), are usually derived from ZF as in [110]:

$$(a, b) = \{\{a\}, \{a, b\}\}.$$

However this would cause problems with incompatibility if a, b were of different types and for this reason ordered pairs are defined axiomatically. In a similar manner the natural numbers are defined by the Peano axioms rather than as a definition arising from the infinity axiom.

In Z, types are *given* if we are not interested in their internal structure, or derived from them using type constructors. Thus if T_i denotes a typed set, further typed sets can be defined recursively as

(i) Power set of T_i, $\mathbb{P}\ (T_i)$, whose type variables are subsets of T_i;

(ii) Cartesian product, $T_1 \times T_2 \ldots \times T_n$, whose type variables are tuples.

Further derived types are *schemas*, *axiomatic descriptions* and *free types.*

A Schema is a typed set each of whose members is a collection of identifiers, or named type variables, the set of names being a given type.

Schemas in Z are defined either vertically or horizontally. The vertical definition of *Schema* is as follows:

$$\begin{array}{|l}
\hline \textit{Schema} \\
D_1;\ \ldots;\ D_n \\
\hline
CP_1;\ \ldots;\ CP_m \\
\hline
\end{array}$$

where each D_i is a declaration of the form $x_i : \tau_i$, where τ_i is set valued, and each CP_i is a predicate constraining the variables of the schema. If $m = 0$ then the constraining predicate is *true*.

The horizontal definition of *Schema* is

$$Schema \mathrel{\widehat{=}} [D_1;\ \ldots;\ D_n \mid CP_1;\ \ldots;\ CP_m].$$

A binding for a schema provides values of x_i which satisfy $CP_1 \wedge \ldots \wedge CP_m$. Supposing that *Schema* has N variables, then its binding is represented by

$$< x_1 \Rrightarrow val_1, \ldots, x_N \Rrightarrow val_N >$$

where '$x_i \Rrightarrow val_i$' means that each variable x_i is associated with a value val_i. A schema specifies a set of such bindings, denoted by $\{S \bullet \theta S\}$;

An Axiomatic description provides the definition of global constants and variables and has the form

$$\begin{array}{|l}
D_1;\ \ldots;\ D_n \\
\hline
CP_1;\ \ldots;\ CP_m
\end{array}$$

where $D_1;\ \ldots;\ D_n$ are declarations and $CP_1;\ \ldots;\ CP_m$ are predicates;

A free type is a (possibly recursive) structure involving global constants and types. Thus

$$T ::= c_1 \mid \ldots \mid c_m \mid d_1\langle\!\langle E_1[T]\rangle\!\rangle \mid \ldots \mid d_n\langle\!\langle E_n[T]\rangle\!\rangle.$$

The restrictions on the constants $c_1 \ldots c_m$ and variables $d_1 \ldots d_n$ are described fully in [104]. In brief, a sufficient condition for a free type definition to be consistent is that each of $E_1[T], \ldots E_n[T]$ is finitary. Examples of finitary objects are cartesian products, sets of finite sets and finite sequences. Examples of objects which are *not* finitary are power set and infinite sequences.

An example of a free type is a binary tree, and a simple kind of free type can be used to define an enumerated set. E.g

$$TrafficLight ::= Red \mid Amber \mid Green$$

The implication is that constants Red, $Amber$, $Green$ exhaust the set $TrafficLight$ and are all distinct. We shall not be concerned with free types in this thesis, apart from the simplest ones.

A further type is a *generic definition*, where a schema is defined in terms of some untyped parameters. Some of the mathematical toolkit is defined in terms of these generic constructs. A sequence of length n of elements from set X can be defined as a function from $1 \mathinner{..} n$ to X. A non-empty sequence of elements from X has type denoted by $seq_1 X$. The head of the sequence is then defined:

$$[X]$$
$$head : seq_1\ X \rightarrow X$$
$$\forall\, s : seq_1\ X \bullet head\ s = s(1)$$

The double bar denotes a generic description.

3.1.1 Small File System Example

A simple example of a specification of a file system is presented to illustrate the basic structures of Z. We shall return to the example in Chapters 4 and 5, for further illustrative purposes. The small file system involves a single given set

$$[FileId]$$

which models the set of file identifiers, whose members are not explicitly provided. There are a maximum number of files, $MaxFiles$, a natural (non-zero) number which is defined axiomatically:

$$MaxFiles : \mathbb{N}_1$$

and is an axiomatic description in which there is one declaration and zero constraining predicates.

We define the file system in terms of its state variables which are $Files$, the files on the system, and $Count$, a count of the files. The state schema is $FileSys$.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{FileSys}} \\
\hline
Files : \mathbb{F}\ FileId \\
Count : 0 \mathinner{\ldotp\ldotp} MaxFiles \\
\hline
\#Files = Count \\
\hline
\end{array}
$$

The finite subsets of $FileId$ are denoted $\mathbb{F}\ FileId$. The declarations of types in the upper part of the schema box mean that $Files$ is a finite subset of $FileId$ and $Count$ is a natural number between 0 and $MaxFiles$ inclusive. The predicate constrains the state variables in that $Count$ is the number of files, $\#Files$. The schema denoting any change in state variables is, by convention, $FileSys'$.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{FileSys}'} \\
\hline
Files' : \mathbb{F}\ FileId \\
Count' : 0 \mathinner{\ldotp\ldotp} MaxFiles \\
\hline
\#Files' = Count' \\
\hline
\end{array}
$$

The variables of $FileSys'$ are primed and denote the values after some operation. Note that their types and predicate constraint are the same as for the unprimed version.

In this very simple specification, there is only one way of modifying the file system. Files are added, one at a time and each time a file is added $Count$ is incremented. The state variable types and modified values are specified by the schema $AddFID$.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{AddFID}} \\
\hline
FileSys, FileSys' \\
NewFile? : FileId \\
\hline
Count < MaxFiles \\
NewFile? \notin Files \\
Files' = Files \cup \{NewFile?\} \\
Count' = Count + 1 \\
\hline
\end{array}
$$

The appearance of the schemas $FileSys, FileSys'$ in the declarations of $AddFID$ means that their declarations are included with the declarations of $AddFID$ and that their predicates are conjoined with the predicate of $AddFID$. '$FileSys, FileSys'$' is usually shortened to '$\Delta FileSys$'. The file to be added is $NewFile?$, the decoration ? denoting it to be an input variable. The predicate gives the restrictions on the input and state variables before the operation can take place and also the value of the variables after the operation. Hence $Count$ is less than $MaxFiles$, $NewFile?$ is not already a file and $Files'$ is $Files$ with the new file added. The count is incremented.

A binding of $AddFID$ associates values of the schema variables to their names. For example if $FileId$ is $\{Fid1, Fid2, Fid3, Fid4\}$ a possible value of a binding is

$$< Files \Rrightarrow \{Fid1, Fid2\}, Files' \Rrightarrow \{Fid1, Fid2, Fid3\},\\ \quad Count \Rrightarrow 2, Count' \Rrightarrow 3, NewFile? \Rrightarrow Fid3 > .$$

The binding of each variable within $< \ldots >$ is consistent with both the declarations and predicate of $AddFID$.

Note that $AddFID$ is in the *constructive style* where the primed state $\Delta FileSys$ is expressed explicitly in terms of the unprimed state. An example of an *unconstructive* style of specification for $AddFID$ is one containing a predicate of the form:

$$\begin{array}{l} \quad \ldots \\ Files \subseteq Files' \wedge \\ Files = Files' \setminus \{NewFile?\} \\ \quad \ldots \end{array}$$

The next section presents the theory of finite sets, to indicate the similarities and differences between finite sets and ZF.

3.2 The Theory of Finite Sets

In order for sets to be manipulated mechanically, it is necessary to present them in a form closer to the level of a computer (such as a list). The axioms for such sets and set operations are presented in detail in Manna and Waldinger [76]. This presents the theory of some fundamental structures of computer science as "theories with induction", and includes the "theory of finite sets" . There are two unary predicate symbols $element(u), set(x)$[1] and $(u \circ x)$ denotes a binary insertion function: the result of inserting element u in set x. The constant symbol $\varnothing$ denotes the empty set. A binary predicate symbol $u \in x$ denotes membership. The text uses a distinctive notation; for example quantification is expressed $\forall\, set(x) : p(x)$ and 'logical and' is *and*. We retain the quantification notation, but we write & for conjunction,

[1]an element can also be a set

for compatibility with later chapters. Key axioms of the theory and the induction principle are in Appendix B and include axioms for set generation, membership, set equality and set union. For example set equality axioms are

Element multiplicity $\forall\, element(u) : \forall\, set(x) : u \circ (u \circ x) = (u \circ x);$

Element exchange $\forall\, element(u) : \forall\, element(v) : \forall\, set(x) : u \circ (v \circ x) = v \circ (u \circ x).$

Sets can be conceptualised as lists where multiplicity and order of elements is irrelevant, and there are many representations of the same set. Set operations resembling those of ZF can be derived from the axioms. For example set equality can be defined in terms of subset:

$$\forall\, set(x) : \forall\, set(y) : x = y \Leftrightarrow x \subseteq y \;\&\; y \subseteq x$$

The transformation of logical expressions of this "set as list" theory to their clausal form and (hence) Prolog formed the basis of the translation technique via program synthesis and is explained informally in the next subsection. It is the most obvious strategy, arising from the mathematical relationship between Z and the fact that Z and Prolog are related in a mathematical way, and this relationship can be realised by a formal synthesis of the Prolog program code.

3.3 Logic Programming Synthesis

A survey of the techniques for synthesising logic programs from specifications is provided by Deville and Lau [28]. The concerns of the survey are that

> "Given a (non-executable) specification, how do we get an (executable) logic program satisfying the program?"

Three main approaches are considered in the survey

1. constructive synthesis (or the 'proofs-as-programs' approach) whereby conjectures about the specification are constructively proved and the program is extracted from the proofs. An example is Henson [54], who presents the constructive set theory TK, which can be employed to derive programs from proofs of specifications;

2. deductive synthesis whereby the program P is deduced directly from the specification S. Hogger [59], who also provides a review of literature on the subject,

describes this relationship as one of *partial correctness* between program P and specification S ($S \models P$);

3. inductive synthesis whereby a program can be generalised from a partial specification. The partial program frequently includes *instances* of predicates. An early example of its use is the MIS system as described in [99].

Since Z is based on ZF, a first order theory, and logic programs can be deduced from logic specifications, the approach in this chapter is confined to deductive synthesis. A most important advantage of the method is that when a program is derived logically from a specification, it is partially correct with regard to its specification [59].

Two examples are given to illustrate two methods of program synthesis. The first method for synthesis of logic programs involves clausal form transformation combined with resolution and the second is the Lloyd-Topor transformation [73]. A technique of obtaining recursive programs via *folding*, originated by Burstall and Darlington [20], is described as being suitable for the kind of program synthesis which involves list manipulation.

3.3.1 Clausal Form Transformation

Kowalski [71] presents a method whereby programs are deduced from specifications using rules of inference such as resolution, combined with clausal form transformation. This use of resolution to synthesise programs contrasts with its more usual use of *running* programs. An example which demonstrates the method and some problems associated with it is provided by the transformation to clausal form of specifications for member and subset. The full method and discussion is contained in Kowalski [71] and only a brief outline and the results are given here. We use '&' for conjunction and $\longleftarrow$ for 'is implied by' to distinguish logic programming notation from Z and ZF. In the clausal form of logic, formulae are restricted to those with the structure:

$$L_1 \vee L_2 .. \vee L_m.$$

The L_i are literals, that is an atom or negation of an atom. Universal quantification is assumed, and existential quantification eliminated. The programming language Prolog is based on the *Horn* clause subset of clausal logic, where formulae contain at most one positive atom, and thus transform straightforwardly to

$$A \longleftarrow B_1 \ \& \ B_2 \ .. \ \& \ B_n.$$

In principle logic formulae can be transformed to their clausal equivalents.

For example, the following is a simplified version of the axioms for set membership and subset of Appendix B:

[**a**] $\neg\ \exists\, z : (z \in \varnothing)$;

[**b**] $\forall\, z : \forall\, u : \forall\, v : z \in (u \circ v) \Leftrightarrow z = u \vee z \in v$;

[**c**] $\forall\, x : \forall\, y : x \subseteq y \Leftrightarrow \forall\, w(w \in x \Rightarrow w \in y)$.

Splitting the "if" and "only if" halves of formula [**c**], for subset, we obtain

$$\forall\, x : \forall\, y : x \subseteq y \longleftarrow \forall\, w : (w \in x \Rightarrow w \in y)$$

and

$$\forall\, x : \forall\, y : x \subseteq y \Rightarrow \forall\, w : (w \in x \Rightarrow w \in y).$$

If we repeatedly apply logical equivalences such as

$$\neg\ (A \vee B) \Leftrightarrow \neg\ A\ \&\ \neg\ B$$

and

$$\neg\ \exists\, x : A \Leftrightarrow \forall\, x : \neg\ A$$

to the above formulae we obtain an equivalent set of three clauses [**d**] **-** [**f**]:

[**d**] $x \subseteq y \vee w(x, y) \in x$;

[**e**] $x \subseteq y \vee (w(x, y) \notin y)$;

[**f**] $x \not\subseteq y \vee z \notin x \vee z \in y$.

In a similar manner formula [**a**] becomes

[**g**] $\neg\ z \in \varnothing$.

In formulae [**d**]**-** [**e**], universal quantification is assumed, and existential quantification has been replaced by the use of an arbitrary function w(x,y), called a *skolem function*. However, note that [**d**] is a *not* a Horn clause, and that by completely breaking down the logic to its constituent clauses we have lost its clarity and cohesiveness. In order to synthesise the recursive Horn clauses necessary for any set implementation we require the use of the resolution or other inference rule.

Resolution matches positive and negative clauses, including appropriate substitution if necessary, so that clauses

$$\neg A_1 \vee B \text{ and } A_2 \vee C \text{ infer } B \vee C$$

if we can find substitutions which will match A_1 and A_2 .

By resolving **[d]** with **[g]** (substituting $z = w(x, y)$ in **[g]**, and $x = \varnothing$ in **[d]**) we obtain

[h] $\varnothing \subseteq y$.

The second part of the Horn clause specification for subset is

[i] $u \circ v \subseteq y \vee u \notin y \vee v \not\subseteq y$

and can be obtained, by further resolutions of clauses **[d]** - **[g]**. Formulae **[h]** and **[i]** are equivalent to the Prolog clauses for subset, where upper case letters denote Prolog variables and `[]` `[A|X]` are respectively the empty list and list with head A and tail X.

```
subset([A|X], Y) :- member(A, Y), subset(X, Y).
subset([], Y).
```

The Prolog clauses for set membership can be similarly derived:

```
member(X, [X|_]).
member(X, [_|Y]) :- member(X,Y).
```

Note that the program synthesis just described is "by hand" and relies on human intelligence to determine which clauses are most suitable for a resolution step, as there is no algorithmic method. Also, as noted by Kowalski a more direct derivation is by a partial conversion of the formulae followed by the application of non-resolution forms of inference. The limitations of the method just described are summarised in Hogger, in that it is not possible to decide which procedures are most suitable, for example which inference rules, for obtaining Horn clauses from logic specifications.

Clausal form transformation relies on the definition of a logic program as a set of Horn clauses. This view of a logic program is limited in that negative information can never be deduced. Lloyd [73] describes how an extra inference rule can also be invoked. This is the *closed world assumption* (CWA), that if a ground atom A cannot be deduced, then infer $\neg A$. However if the CWA is to be sound then the failure of A must be finite (the program must terminate) and also A must be ground. The method below allows the possibility of negative literals.

3.3.2 Lloyd-Topor Transformation

A systematic method of obtaining a Horn Clause program from arbitrary logic specification is the Lloyd-Topor transformation [73]. This is more convenient than "straight" clausal form transformation for it replaces expressions in the form of $Head \longleftarrow Body$ to $Head \longleftarrow New\_Body$. For example, a version of the set-equality definition of Section 3.2 is

$$x = y \Leftrightarrow \forall\, u : u \in x \Rightarrow u \in y \;\&\; u \in y \Rightarrow u \in x.$$

The 'if' part transforms as follows:

$$x = y \longleftarrow \forall\, u : u \in x \Rightarrow u \in y \;\&\; u \in y \Rightarrow u \in x$$
$$x = y \longleftarrow \neg\, \exists\, u : \neg\, (u \in x \Rightarrow u \in y) \vee \neg\, (u \in y \Rightarrow u \in x).$$

A novel feature of the transformation is the introduction of a new predicate (here called aux). Thus set equality becomes

$$x = y \longleftarrow \neg\, aux(x, y)$$
$$aux(x, y) \longleftarrow \exists\, u \neg\, (u \in x \Rightarrow u \in y) \vee \neg\, (u \in y \Rightarrow u \in x)$$

with aux defined in the second statement. By splitting the second predicate to two and re-expressing in Prolog, set equality becomes

```
set_equ(X,Y) :- not(aux(X,Y)).
aux(X,Y) :- member(U,X), not(member(U,Y)).
aux(X,Y) :- member(U,Y), not(member(U,X)).
```

with "member" defined as above and an additional clause `aux(X,Y)` generated. However running the resulting Prolog presents problems. Prolog queries with instantiated X, Y such as

```
| ?- set_equ([1, 2, 3, 2], [2, 3, 1, 1]).

yes
```

return expected responses (noting that order and multiplicity of list elements is irrelevant). However, if either X or Y is unknown, the Prolog system flounders through trying to satisfy predicates of the form `not(p(Z))` where `Z` is unknown. In principle there are an infinite number of answers to

```
set_equ([1, 2, 3], X)
```

given the axiom regarding multiplicity of list elements. The problem can be solved by the simple expedient of adding

```
set_equ(X, X).
```

to the clause database, which can be obtained either by observation or by unfolding of the clauses and further application of resolution.

The above derivation indicates the improvement that the use of the Lloyd-Topor transformation has over 'straight' clause transformation. However it is still limited in that it again requires human intelligence to obtain a workable program. The problem lies in the fact that for Prolog predicates involving set operations to succeed in cases where variables are unknown we need recursion. A method of obtaining recursive programs is briefly reviewed.

3.3.3 Recursive Programs

The problems of obtaining recursive programs from specifications have been tackled in Lau [72] who suggest the technique of *fold/unfold*. The user specifies the form of the desired recursive calls in the form of a folding problem. The fold/unfold technique by Burstall and Darlington [20] involves the following: in folding, given a clause *Head* $\longleftarrow$ *Body*, and suitable substitutions, the occurrence of *Body* in an expression is replaced by *Head*; *unfolding* is the converse. The derivation of subset and member clauses in Section 3.3.1 is a generalisation of the fold/unfold technique. However the techniques for automatically producing recursive logic programs are still problematic [92]. The latter reference presents a method of program synthesis which extracts a program from a fold/unfold proof. However the authors remark that suitable strategies still need to be devised for particular classes of program.

3.4 Summary

The initial work of translation from Z to a logic program involved the search for some algorithmic procedure for converting Z into clausal form and hence into Prolog. Z is based on ZF set theory, a theory of first order logic, and programs can be synthesised from specifications in first order logic, so this would seem the most obvious strategy. Since Z is based on sets, we require a suitable data model of sets in the logic program. In order for sets to be manipulated mechanically, it is necessary to present them in a form closer to the level of a computer (such as a list). However list manipulation presents problems in that recursive programs are required for their manipulation - and these are difficult to synthesise from a specification. We have identified a link between finite sets in ZF and finite set theory (**Contribution 2**). However given the limitations of algorithms for program synthesis outlined previously, a change of direction was determined on. The change of direction involved *Structure Simulation*, which required a "flattening" of Z to a form suitable for a logic program and in this sense is similar to the the original translation process. However, instead of a formal transformation, characteristics of Z schema were identified and adapted so that the logical structures of the specification would be preserved as far as possible in the resulting logic program model. This method is examined in the next chapter.

Chapter 4

Structure Simulation

4.1 Introduction

The previous chapter described attempts to generate a logic program from a Z specification using program synthesis. This chapter describes an alternative method of obtaining a translation of a Z specification, viz. *structure simulation*, a method in which 'Z' is captured in a manner which is amenable to simulation by a logic program. Structure simulation was originated by West and Eaglestone [119]. It involves the translation of a subset of Z to the logic programming language Prolog; a prototype translator is presented in [13]. However, West and Eaglestone identify certain shortcomings of the Prolog language and this chapter presents a simulation by an alternative LP, viz Gödel.

In Section 2 of this chapter we briefly describe the previous work of West and Eaglestone in the use of structure simulation. We also compare the method with other Prolog animators, and outline its respective advantages and disadvantages. The advantages in using the logic programming language Gödel for the simulation are then presented. In Section 3, the main features of the Gödel language are outlined and we provide, in particular, details of its type system and of its support for sets. Section 4 of the chapter describes the structure simulation rules using the 'small file system' from Chapter 3 for illustration. The remainder of the chapter is devoted to two case studies from [51], a simple assembler (Section 5) and the Unix file system (Section 6). The assembler is an update and extension of [119], which presented an animation in Prolog of a simple assembler. The updated version is in Gödel, and we demonstrate its improved qualities as an animator for Z. A version of the Unix file system case study was originally presented in West [117] and includes a representation in Gödel of some of the more complex features of Z. These features of Z (and the Z toolkit) were not covered by the original Prolog animator; structure simulation is **Contribution 3** of the thesis. The chapter ends with a review of the case studies and a summary of the advantages the approach has over animation in Prolog.

4.2 Translation of Z to Prolog

This section describes work done in translating Z to Prolog and includes previous work done by ourselves.

4.2.1 Comparison with Other Methods

Structure simulation is described in West and Eaglestone [119], in which a subset of Z is translated to Prolog; it is a translation strategy in which characteristics of Z schemas are identified and adapted so that the logical structures of the specification are preserved as far as possible in the resulting Prolog interpretation. The technique was applied to the translation of a Z specification for an assembler [51]. It was also applied by West et al. [120] to a real-life example: Pelican Crossing equipment. In the simulation, Z schemas are represented by predicates and a query to a schema should result in zero or more bindings satisfying that schema, depending on the instantiated values. A schema has possible values of its variables established by the signature and restricted by a predicate. Thus the predicate

$$Schema \Leftarrow Signature \wedge Predicate$$

establishes a logical relationship between a schema and its signature and its predicate. The closed world assumption was briefly discussed in Chapter 3, and one of the implications of this assumption is that given a definition of a predicate (such as for *Schema*) although only the 'if' part of the definition is provided, this is equivalent to 'if and only if'.

The signature provides a means of checking that a binding is as specified; the strategy is to provide bindings for some of the schema variables, query the program and generate the bindings for the rest of the variables. The method is supported by a library of Prolog code which models the evaluation of set expressions, including relations and functions. If backtracking is initiated, further bindings are generated and thus the set of bindings satisfying the schema in the model environment is presented incrementally. The binding is such that every variable within a schema is represented by a Prolog variable. Thus specification variables such as time can be explicitly represented, as was the case for Pelican equipment.

The SuZan project is described in [29], in which a subset of Z is animated in Prolog. The principal difference between the simulation technique and that of the SuZan project is the use of predicates in the schema signature to generate data. In the SuZan project, *Schema* is used as a means of a *generate and test* cycle. The signature type constructor predicates are coded in a manner which generate values from instantiated given sets, and these values are subsequently tested for conformance with *Predicate*. Thus a variable, C, which is declared as follows:

$$C : \mathbb{P}\, A$$

translates to `powerset(A, Ps), member(C, Ps)`, where the variable `Ps` denotes the power set of given set `A`. If `Ps` is not instantiated, the first predicate will generate it, the second predicate selecting each member of this power set for testing against other generated variables for conformance with the schema predicate. In contrast, structure simulation would check if a particular value of C was a subset of a given set A before checking it in the predicate.

However for a set A of size a the number of subsets of A is 2^a; the data generated is liable to grow exponentially with respect to the size of the given sets, so the execution process requires some form of control. The SuZan researchers have used devices such as “unfolding” [20] and “filter promotion” for such control purposes. Unfolding replaces a predicate with its logically equivalent set of conjoined predicates, and then removes duplicates. Filter promotion re-orders generating predicates so that fewer predicates will actually generate data, the rest acting as constraints. By suitable instantiation, and using appropriate control facilities, a range of “what if” questions are possible, which are not available in our approach. For example it is possible to generate type variable values from given set instantiations. The second major difference between the two approaches is that the SuZan group use a library of “pure” Prolog to implement set operations. This contains no extra-logical features such as cut whereas the Prolog library utilised by the simulation contains some extra-logical features to curtail fruitless backtracking.

As described in Chapter 2, Breuer and Bowen [17] have identified requirements for an animation. They include coverage of Z, sophistication, efficiency and correctness. An aim of the simulation presented in [119] was that as much coverage of Z as possible is attempted, and that the code should be sophisticated. The assembler is a complex and substantial specification which was successfully animated in Prolog using the simulation technique. Response times to the animation code were of the order of a second and so far as coverage of Z, sophistication and efficiency is concerned, the technique was deemed successful. The animator was also applied, to useful effect, to the Pelican Equipment Study. The method was thought superior to that described in the SuZan project in that both its performance and its sophistication were greater.

The issue of correctness of the simulator was limited to an investigation as to whether program synthesis (described in Chapter 3) was suitable for translation of Z to Prolog and in any case the decision to use extra-logical features in the simulation would frustrate any check for correctness. Any implementation of Prolog is unsound with respect to the semantics of first order logic and this invalidates any attempt to prove the correctness of a Prolog implementation of Z. It is argued here that a better technique is to use the Gödel language [56] for the reasons cited in the next subsection.

4.2.2 Advantages of Gödel

Gödel is adopted for the following reasons:

1. Prolog clauses are selected by the interpreter in the order they appear. This limits the "what if" facility of animation; the deduction of any variable from instantiated variables is not straightforward unless the code is ordered in an appropriate manner. In contrast Gödel has a flexible computation rule that can be constrained by user-defined control declarations;

2. Prolog has difficulties in dealing with negation if the negative literal is not ground (see [7, 73]). In that case the program will flounder or, in the case of some implementations, will issue an error message. The computation rule used by Gödel ensures that all calls to negative literals are ground;

3. Z is based on typed set theory and Prolog lacks support for types or sets. The Prolog predicate "setof" has no declarative meaning. Gödel has been carefully designed and implemented, and supports both types and sets;

4. Prolog is unsound, for many reasons including the existence of extra-logical features such as 'assert' and 'cut'. Also, Prolog has no occur check in the unification algorithm, although some implementations allow for this to be checked. Apart from some well-defined exceptions, a program in Gödel is defined as a theory in first order logic and an implementation must be sound with respect to this semantics;

5. Although it was possible to write code for set operators such as intersection and union, it was necessary to introduce the non-declarative 'cut' extensively in order to avoid fruitless backtracking. However Gödel contains built-in implementations of most of the ZF set operations;

6. The translation of some first order constructs such as quantification was awkward in Prolog. For example universal quantification was achieved only by writing a separate predicate for each of its occurrences. This predicate included the use of 'set-of' to build the set over which the quantifier could range. In contrast, Gödel allows the use of both universal and existential quantification. The existence of built in set operations, universal quantification and the ability to form constructs such as 'ordered pair' made the Gödel code both easy to write and transparent to read.

The next section introduces the Gödel programming language and provides examples of how its support for types and sets can be utilised in animation.

4.3 The Gödel Programming Language

4.3.1 Overview

A full and formal description of the Gödel language which also includes tutorial examples can be found in [56]. A brief introduction to the Gödel language is presented here. The logic programming language Gödel has a greatly improved declarative semantics compared with Prolog, and supports a set data type in a manner described below. Furthermore, Gödel is *typed*.

To allow for program structuring, the language is *modular* and modules can be exported (to modules) and can themselves import (other modules). A Gödel program is a collection of modules. There are a number of system modules and these include modules for integer, rational and floating point arithmetic. Further system modules provide structured data types such as `Set`, `List`, `String`, `Table`. Gödel provides special facilities for meta programming and system modules are provided for reasoning about other Gödel programs.

Language declarations include the categories: base, constructor, constant, function, proposition and predicate. A predicate definition consists of a declaration, specifying the type(s) of its arguments, and a set of statements of the form

```
Head <- Body.
```

where `<-` in Gödel means "if" and in contrast to Prolog, upper case is used for constants and lower case for variables. `Head` is an atom with the defining predicate and `Body` is a formula in first order logic and may be absent. Besides the usual first order constructs such as universal and existential quantification, `Body` can also contain the construction `IF ..THEN..ELSE`.

4.3.2 Types and Sets

Many sorted first order logic provides the basis for Gödel, whose sorts are then *types*. Gödel is strongly typed; each constant, function and predicate must have its type(s) specified. A Gödel program is checked for type correctness during compilation. The goal is subsequently checked at run-time before execution. The `BASE` declaration declares the *base types* of the module. For example a base `FileId` can be declared to represent file identifiers. In addition there are type constructors such as `List/1`, so that `List(a)` is a generic list with parametric type `a`.

In Gödel, sets are supported by the system module `Sets`. Thus `Set(a)` constructs a term whose *intended meaning* is a set of elements of type `a`. The set terms of Gödel can be compared with the "finite sets" of chapter 3. In Gödel `Inc` is a mapping *Include* with the property $Include(d, S) = \{d\} \cup S$, where d are an element and S a set. *Include* can thus be compared with $\circ$. Braces for sets provide some notational sugar. For example `{1, 2, 3}` stands for

```
Inc(1, Inc(2, Inc(3, Null)))
```

where `Null` is the empty set (`{ }`), and the integers are supplied by Gödel. The multiplicity and order of elements in a set is irrelevant and the equality of extensional sets takes this into account:

```
{5, 6, 7} = {7, 7, 6, 5}.
```

Set terms can also be intensional:

```
s = {x : 1 < x < 5}.
```

The precise meaning of this example of an intentional set (term) is

```
ALL [x] (x In s <-> SOME [y]( x = y  & 1 < y < 5)).
```

This semantics for intentional sets was first given in [21]. The `Sets` module also provides functions for set union (`+`) and intersection (`*`). Set difference is represented by (`\`). Predicate `In` represents set membership and `Subset` represents subset. Both are provided by `Sets`. Constructors also support the introduction by the programmer of data types such as trees.

The following presents some queries and answers to a Gödel module `Demo1`, where '`[Demo1] <- `' is the prompt. The module imports Integers, Sets and Rationals.

```
[Demo1] <- x = {1,2,3,4,5} + {4,5,6,7,8} \ {3,4,5,6}.
x = {1,2,7,8}?

[Demo1] <-
IF SOME [x] x = {y : 1 =< y =< 5} & x ~= {} THEN z = x ELSE z = {0}.

z = {1,2,3,4,5}?

[Demo1] <- x Subset {1, 3, 4, 2}.

x = {} ? ;

x = {1} ? ;

x = {1,2,4} ?

Yes
```

The latter query eventually supplies all subsets of `{1, 3, 4, 2}`. In general the program supplies a *set* of answer substitutions. However some of the set may be undefined:

```
[Demo1] <- (x = 0) & ( (y = 3 + x) \/ ( y = 6 - x ) \/ (y = 3/x) ).

x = 0,
y = 3 ? ;
```

```
x = 0,
y = 6 ? ;
Arithmetic exception: division by 0.

[Demo1] <- (x = 1) \/ ({x, 1, 2, 3} = {1, 2, 3, 4}).

x = 1 ? ;
Floundered.  Unsolved goals are:
Goal: {v_1,1//1,2//1,3//1}={1//1,2//1,3//1,4//1}
Delayed on: v_1
```

The problem of set equality when part of the set is non-ground arises because of the under-developed constraint satisfaction properties of Gödel. Moreover, if the query is expressed in a different manner, none of the set may be provided:

```
[Demo1] <- (x = 0) & ( ( y = 3/x)  \/ (y = 3 + x) \/ ( y = 6 - x ) ).
Arithmetic exception: division by 0.

[Demo1] <- ({x, 1, 2, 3} = {1, 2, 3, 4}) \/ (x = 1) .
Floundered.  Unsolved goals are:
Goal: {v_1,1//1,2//1,3//1}={1//1,2//1,3//1,4//1}
Delayed on: v_1
```

We examine further the issue of ‘undefinedness’ in Chapter 5.

The `[Sets]` module is imported by a library of Gödel code which implements relations, functions etc belonging to the Mathematical Toolkit described in [104]. The ‘`Lib`’ module is described in Section 4.3.3.

4.3.3 Translation Architecture and the ‘Lib’ Module

The ’architecture‘ for animation of a Z specification is as follows:

- the `Lib` module, a library of code which implements the Mathematical Toolkit;
- a module containing predicate definitions of schemas associated with a specification. This module imports the `Lib` module.

The full version of `Lib` is in Appendix C and has an EXPORT and LOCAL part. However in this chapter the parts are, in the main, merged. The module includes the implementation of constructions from the Toolkit such as $\bigcup A$ where A is a set of sets:

$$\bigcup A = \{x : X \mid (\exists\, s : A \bullet x \in S)\}.$$

(See also Appendix A.) The characteristic predicate `DUnion` has two arguments, the first of which is a set of sets of some item, and the second of which is the set of items. Its export and local definitions are

```
PREDICATE  DUnion:  Set(Set(a)) * Set(a).


% x is a set of sets of some type and y is the
% distributed union of x
DUnion(x, y)  <- y = {z :  SOME [w] (w In x & z In w) }.
```

A query and answer to `Lib` is shown.

```
[Lib] <- DUnion({{1, 3}, {2, 3}, {5, 1}, {}}, x).
x = {1,2,3,5} ?
```

A partial function f from X to Y is defined in the Toolkit:

$$\{f : X \leftrightarrow Y \mid \forall\, x : X;\; y, z : Y \bullet ((x \mapsto y) \in f) \wedge ((x \mapsto z) \in f)$$
$$\Rightarrow y = z)\}.$$

In order to implement ordered pairs and relations, the `Lib` module also includes a type constructor, `OP`. This type constructor allows the definition of `OrdPair` (shown below). Relations are additionally constrained, for example for functionality.

The predicate `PF` which is next shown models partial function f and is generic to sets of any type. The characteristic predicate `PF` has three arguments, the first of which is a set of ordered pairs from two sets. The second and third arguments are the first and second sets. The predicate definition of `PF` replicates the ZF definition of partial function. The definition is followed by a query.

```
CONSTRUCTOR OP/2.
FUNCTION    OrdPair : a * b ->  OP(a,b).


%%%%% declaration of partial function %%%%%
PREDICATE   PF  : Set(OP(a,b)) * Set(a)  * Set(b).


%%%%% This characteristic predicate determines whether a set is
%%%%% a function from s1 to s2       %%%%%
```

```
 PF(pf, s1, s2) <- ALL [z,x,y]
                       (z In pf & (z = OrdPair(x,y))
                   ->  (x In s1) & (y In s2)  &
                   ALL [u] (OrdPair(x, u) In pf -> u = y)).

% query and answer: partial function
[Lib] <- PF({OrdPair(1,2), OrdPair(3,2)}, {1,2,3}, {1,2,3}).

Yes
```

Note that since the variables `s1, s2` are both *sets* then these must always be provided. This is true for all sets including the integers. The `Lib` module provides other parts of the Z notation, such as function override, domain and range restriction.

4.4 Structure Simulation: Rules

This section describes a set of rules for animation of Z. The example of the 'small file system' from Chapter 3 is used to illustrate these rules.

$[FileId]$

$MaxFiles : \mathbb{N}_1$

FileSys

$Files : \mathbb{F}\ FileId$
$Count : 0 \mathinner{\ldotp\ldotp} MaxFiles$

$\#Files = Count$

AddFID

$\Delta FileSys$
$NewFile? : FileId$

$Count < MaxFiles$
$NewFile? \notin Files$
$Files' = Files \cup \{NewFile?\}$
$Count' = Count + 1$

Recall from Chapter 3 the type of an expression in Z, which is either *given* or derived using type constructors *Power set*, *Cartesian product*. A further derived type is a *schema*, a typed set each of whose members is a *binding*, a collection of named (bound) type variables. All these features are implemented in a manner which will be described. The Z schemas in the example are interpreted in a single Gödel module `Demo2`, which imports predicates from the `Lib` module.

Schemas are also represented by characteristic predicates, defining the relationship between schema variables. In the main, the animation strategy is to *test* schema variables to see if they satisfy the characteristic predicate for that schema. If possible, other variables are generated associated with bindings for the schema. The representation in Gödel of given sets, schema variables, schemas and schema declarations is presented in a tutorial manner in the next subsection, followed by the small file system example. The complete code of the latter can be found in Appendix C.

4.4.1 Givensets and Bindings

Given Sets : In order to provide a small model environment, elements of the given sets are provided as if they were enumerated free types in Z. The given sets of the specification are declared as one of the base types (in Gödel) and some constants of the base types introduced to model the environment. These constants are coded into extensional sets, for there is no automatic facility for obtaining a set of all elements of a particular type. In the example, `FileId` is declared as a BASE type and constants `F1, F2, F3` declared to be of type `FileId`. In addition each of the constants are declared as a member of the type via the predicate `IsFileId`:

```
PREDICATE   IsFileId : FileId.
```

The declarations `IsFileId(F1), IsFileId(F2), IsFileId(F3)` enable the constants to be collected together via the use of an intentional set, `setfid`

```
[Demo2] <- setfid = { g : IsFileId(g) }.
 setfid = {F1,F2,F3 } ?
```

The result is the same as if *FileId* was defined by the enumerated free type:

$$FileId ::= F1 \mid F2 \mid F3$$

Schema and Variable Names Further BASE types are schema and variables names and these are modelled by `Name, Var` respectively[1]. Note that although it is desirable as far as possible to replicate Z names in Gödel, upper case must be used as the first character since names are constants. For example, `FileSys` is declared as base `Name`, and the state variable `Files`, declared as base `Var`. Gödel functions capture variable and schema "decorations". Thus `DSet` is a function from `Var` to `Var` and primes a variable name. In a similar manner `IN` decorates an input variable, `OUT` an output: functions `IN`, `OUT` have as their argument a type of `Name` and return the same type. Examples of this use of functions are `DSet(Count), IN(NewFile)` which denote *Count′*, *NewFile?* respectively;

Schema and Variable Bindings : A further BASE type is `BindVar`, used to facilitate the binding formation of schemas. *Lists* of type `BindVar` form a binding of a schema. Variable names are bound to their values via functions over the appropriate types. For example:

```
FUNCTION Bind1 : Var * Set(FileId) ->  BindVar.
```

The second argument of `Bind1` can be a ground term of the appropriate type, or a Gödel variable, for example `Bind1(Files, {F1, F2})`. The following list of `BindVar` forms the schema binding of `FileSys`:

```
[Bind1(Files, files), Bind2(Count, count )  ]
```

which can take the value:

```
[Bind1(Files, {F1}), Bind2(Count, 1 )  ].
```

The typing restrictions ensure that Gödel variables

```
files, files1, count, count1,newfile
```

are all appropriately typed.

In the next subsection it will be shown how the binding function device ensures that types and identifiers associated with a given schema are completely fixed by function declarations and schema statement. They also explain how schema declarations and predicates are modelled.

[1] Although, strictly, schema and variable names are the same Z type *Name*, it is convenient here to ascribe them to different types.

4.4.2 Schemas

Schema Bindings : A schema is modelled by:

$$SchemaType(binding, name) \Leftarrow Signature(binding\_variables)$$
$$\wedge\ Predicate(binding\_variables)$$

where *binding* is of type `List(BindVar)` and *name* is of type `Name`. $SchemaType(binding, name)$ is defined by the Gödel predicate:

```
PREDICATE SchemaType: List(BindVar) * Name.
```

For example the schema named *FileSys* has schema clause head as follows:

```
SchemaType( [ Bind1(Files, files ), Bind2(Count, count )], FileSys)
```

Schema names are also decorated via Gödel functions, thus `DSch(FileSys)` primes `FileSys`;

Variable Declarations : Variables are checked for additional constraints via $Signature(binding\_variables)$, for example if they represent a partial function;

Schema Predicate : $Predicate(binding\_variables)$ constrains the declared variables using predicates and functions provided by the system. Existential and universal quantification are both provided by Gödel, as are set operations;

Schema Declaration in a Signature: If a schema B is declared in the signature of schema A, then the clause head for B is conjoined to the signature of A. The bindings are appended. For example the binding for `AddFID` is composed of the binding of `Del(FileSys)` appended to `[ Bind3(IN(NewFile), newfile)]`.

Given the binding function declarations `Bind1, Bind2`, the list inside the predicate models $\theta FileSys$. The set of all schema bindings is $\{FileSys \bullet \theta FileSys\}$ and the answer substitutions for a given query provide a subset of this set.

4.4.3 Testing Strategy for Animation

The testing strategy for animation is similar to the strategy for software testing. However the data required is of a more abstract nature and for every test the data should be checked as part of the typing constraints of the specification. Tests for animation purposes can be divided (as in the case of software testing) into black-box testing and white-box testing. Black-box testing is designed to include tests which check schemas which model top level functions and which therefore accept data from the containing system. Testing can be with regard to *input* in the same manner as software testing, so that for schemas which model each software function tests should include

- different classes of input to schemas;
- boundary values for input;
- how the software will cope with invalid input;
- input with safety or integrity implications.

For state based systems, consideration should also be given to

- any initial and terminal state(s);
- state partitioning regarding equivalence classes and their boundary values as described briefly in Section 2.2.4.2 and more fully in [30, 113, 106].

Equivalence partitioning could be a way of overcoming the problem of infinite sets, with (for example) an equivalence partition class of numerical values greater than a particular value tested by using representatives of that class.

White box testing involves the structure of the specification and includes 'integration testing' involving interfaces between schemas, schemas which call other schemas and 'unit testing' of lower level schemas. Testing should include the criteria that every conjunct in a schema predicate is exercised and should also check

- boundary conditions for in-range data;
- system *states*, that they adhered to any safety and integrity constraints;
- error handling schemas by the use of erroneous data.

Some of the above tests would be in conjunction with proof obligations of the system. It is easier to show by a few well constructed tests that a proof obligation does not hold, than to attempt to discharge it, which can be difficult and costly [85]. In some cases (as for constructive state based specifications), some test variables are instantiated (e.g. input and state variables) and the code allows the rest of the variables to be calculated (e.g. output and post-operation variables). In other cases, most or even all variables require instantiation and the animation provides a check.

4.4.4 Animation of the Small File System

The small file system specification is now animated in order to demonstrate the general method and the simpler of the rules presented in the previous section. The complete code in `Demo2` for the file system is provided in Appendix C. The module is divided into its export part which declares bases, constants, predicates and functions. The local part defines the predicates. In this case there are predicates defining data, *IsFileId*, and predicates defining schemas *FileSys*, *FileSys'*, Δ*FileSys* and *AddFID*. The assumption is that the user of the animator provides a value for *MaxFiles* of 10, so that the required subset of integers for *Count* is $1 \mathrel{..} 10$. The set *FileId* is obtained via its characteristic predicate *IsFileId*.

The local part of the module defines the predicates which model schemas where Gödel is used to check the type of the variable (e.g. `files, files1` are both subsets of `setFID`). The given set *FileId* is modelled in a small way, by the intentional set `{x : IsFileId(x)}`. The schema predicates are modelled by the library code in `Lib` which includes the Sets module. For example

```
files1 = files + {newfile}
```

models $files' = files \cup \{NewFile?\}$.

The following provides a fragment of the code which models the small file system specification.

```
LOCAL Demo2.

% schema for state FileSys
SchemaType( [ Bind1(Files, files ), Bind2(Count, count )], FileSys)
  <-
    setFID = {x : IsFileId(x) } &
    files Subset setFID &
     count In {y : 0 =< y =< 10} &
     Card(files, count).

% schema for state Del FileSys
SchemaType(binding, Del(FileSys)) <-
    SchemaType(b1,  FileSys) &
    SchemaType(b2, DSch(FileSys)) &
          Append(b1, b2, binding) .

% DSch(FileSys) is not shown.
```

```
% schema for operation AddFID
SchemaType( binding, AddFID ) <-
       SchemaType(binding1,  Del(FileSys)) &
   binding1 = [Bind1(Files, files ), Bind2(Count, count ),
     Bind1(DSet(Files), files1 ), Bind2(DSet(Count), count1 )]  &
      Append(binding1, [ Bind3(IN(NewFile), newfile)] , binding) &
     count < 10 &
     setFID = {x : IsFileId(x) } &
     newfile In  setFID &
  ~ ( newfile In files) &
    files1 = files + {newfile} &
     count1 =  count +   1 .
```

The following shows some possible queries. The queries are with respect to the bindings of three schemas, *FileSys*, Δ*FileSys*, *AddFID*. In each case, none of the variables is instantiated for it is possible for them to be determined from the information in the schema declarations and predicate. In this case the schema declarations `files Subset setFID` and `files1 Subset setFID` mean that all possible subsets are generated and the predicate checks their values. The Unix file store case study of Section 4.6 provides an example of data types which are not so amenable to generation of uninstantiated values.

Section 3.1.1 describes how the specification of the small file system is constructive in that the primed variables are expressed explicitly in terms of the unprimed variables. A non-constructive schema *AddFID*1 which expresses the same constraints as *AddFID* has been animated. This produces the same results as the original form and can be seen in Appendix C. This is because `files` and `files1` in *AddFID*1 are also instantiated from their declarations. (This subject is revisited in Section 5.6.7.2.) The output eventually provides *all* schema bindings associated with the three schemas via sets of answer substitutions. The following (commented) output provides examples of the kind of testing required to validate a specification. These include unit testing of component schemas and integration testing of components with each other.

```
% test of schema FileSys
[Demo2] <- SchemaType(b, FileSys).
% the initial state
b = [Bind1(Files,{}),Bind2(Count,0)] ?
% second schema binding - a further state
```

```
b = [Bind1(Files,{F1}),Bind2(Count,1)] ?

Yes
% the following provides information about integration of
% schema FileSys and DSch(FileSys).
[Demo2] <- SchemaType(b, Del(FileSys) ).
  % first schema binding
b = [Bind1(Files,{}),Bind2(Count,0),Bind1(DSet(Files),{}),
              Bind2(DSet(Count),0)] ? ;
 % second schema binding
b = [Bind1(Files,{}),Bind2(Count,0),Bind1(DSet(Files),{F1}),
              Bind2(DSet(Count),1)] ?
Yes

% the following tests provide information about integration of schema
% Del(FileSys) and also about function of AddFID

[Demo2] <-
SchemaType(b, AddFID).
 % first schema binding
b = [Bind1(Files,{}),Bind2(Count,0),Bind1(DSet(Files),{F1}),
           Bind2(DSet(Count),1),Bind3(IN(NewFile),F1)] ? ;
 % second schema binding
b = [Bind1(Files,{}),Bind2(Count,0),Bind1(DSet(Files),{F2}),
           Bind2(DSet(Count),1),Bind3(IN(NewFile),F2)] ? ;

b = [Bind1(Files,{F1}),Bind2(Count,1),Bind1(DSet(Files),{F1,F3}),
           Bind2(DSet(Count),2),Bind3(IN(NewFile),F3)] ?
```

The first schema binding for *AddFID* models

$$\begin{aligned} &< \mathit{Files} \Rrightarrow \{\}, \mathit{Count} \Rrightarrow 0, \mathit{Files}' \Rrightarrow \{F1\}, \\ &\quad \mathit{Count}' \Rrightarrow 1, \mathit{NewFile}? \Rrightarrow F1 > . \end{aligned}$$

The input of error or out of range values can also be modelled:

```
%% invalid input of F5 which is not a file

[Demo2] <- SchemaType(b, AddFID) & b = [Bind1(Files,{}),Bind2(Count,0),
           Bind1(DSet(Files), x), Bind2(DSet(Count),1),Bind3(IN(NewFile),           F5
```

```
Error: undeclared or illegal symbol in term: "F5".

%% out of range value (of 12) for count

[Demo2] <- SchemaType(b, AddFID) & b = [Bind1(Files,{}),Bind2(Count,0),
           Bind1(DSet(Files),x), Bind2(DSet(Count), 12),
           Bind3(IN(NewFile),y)].

No
```

This small animation demonstrates the superiority of Gödel over Prolog, for we have been able to use sets and types here, rather than set-of, as in Prolog. Also the ordering of the code has not prevented the eventual output of *all* the bindings consistent with the original instantiation. Even a simple specification requires 'union' and this is captured in a straightforward way in Gödel and *without* the use of non-declarative Prolog control features such as 'cut'.

All of these features compare favourably with the animation of [119]. The next subsection briefly describes how other features of Z such as schema expressions and axiomatic descriptions are modelled.

4.4.5 Schema Calculus and Schema Referencing

Schema Calculus: Conjunction and disjunction of schemas A, B is modelled by conjunction and disjunction of the Gödel predicates of schemas A, B. The variables are merged; the lists are appended and duplicates removed. Schema composition and piping is accomplished by conjunction, with appropriate adjustment to the variable list in the new schema;

Schema Referencing: As explained the constructed type `List(BindVar)` of *Schema* is equivalent to the binding formation $\theta Schema$. Code is provided to form the set of all bindings for a schema, $\{S \bullet \theta S\}$. An example of its use is given in Section 4.6.2 and the code can be seen in Appendix C;

Axiomatic Descriptions and Generic Definitions: Axiomatic Descriptions are divided into two types, ones defining a single constant (such as *MaxFiles*) and one modelled in the same way as a schema. In the first case, their value would be supplied by the user of the animator (in the same manner as the instantiation of given sets). In the second case, for more complex axiomatic descriptions, suitable *names* must be generated for them `Axiom1, Axiom2`.... They must then be conjoined to the schemas which refer to them, otherwise there is no check on the constraints introduced. Generic definitions are treated in the same way as the parametrised definitions of functions etc, i.e. by using parameters `a,b..` instead of generic sets X_i.

Sequences are an example of a generic definition, and their use is demonstrated in the assembler. They are defined as a partial function from the integers to a set of items: the first predicate tests whether a term is a sequence, the second finds its head. The definitions are as follows:

```
% Example of generic definition- head of sequence.
% First define a sequence
IsSequ(sequ, items, size) <- Size(sequ, size )  &
                     domseq = {n: 1 =< n =< size} &
                    TF(sequ, domseq, items).

% Head of sequence- corresponds to OrdPair(1, item);
% Non-empty sequence
HeadSequ(item, sequ) <- sequ ~= {}  &
                   OrdPair(1, item) In sequ.
```

The two case studies which follow illustrate the rules (and tests).

4.5 Animation Example 1

4.5.1 Assembler

The case study used to illustrate the rules for translation is an assembly process described in [51]. This illustrates how more features of Z are captured in an animation, and uses more of the Gödel library. It also serves as a useful comparator between the Prolog and Gödel Z animators. The Prolog animation of the assembler in [119] contained a critique of the method, and one of these was that Prolog does not properly represent first order logic and so quantification was not properly modelled. Gödel does contain quantification and this was found to be in advantage, as will be seen.

The assembler presented here has been extended to include the two phases contributing to its implementation. None of the schemas involves state change, so the priming convention does not apply. The only decoration is ?, ! for input and output

Assembly Language			Machine Language		
Label	Opcode	Operand	Location	Opcode	Operand
v1		100	1		100
v2		4095	2		4095
loop	load	v2	3	01	2
	subn	8	4	03	8
	store	v2	5	02	2
	compare	v1	6	50	1
	jumple	exit	7	61	9
	jump	loop	8	71	3
exit	return		9	77	

Table 4.1: Example of Translation from Assembly to Machine Language

variables. The schemas model 'constraints between variables' rather than a state machine. The specification is (unlike the small file system) not constructive. For simplicity the computer is regarded as a "one address machine", in that two integers (opcode and operand) are located at a single machine address or location in the machine's memory. It is analysed by means of the following sample of assembly code and its machine equivalent (Table 4.1). As can be seen, an assembly instruction has three fields: label, opcode and operand, and not every instruction has one of each represented. The machine code's two fields are related in the following way: a machine instruction's opcode determines whether its integer operand is treated as a value or as an address. If an assembly instruction is labelled, then that label uniquely determines the instruction, and conversely every referenced label must appear in the label field of some instruction. The operand mnemonics, when they appear in the operand field of an assembly instruction, are directly translated to their numeric value in the corresponding machine operand.

4.5.2 Assembler and Machine Requirements in Z

The "given" sets are of assembly and machine instructions: A and M, and of permissible labels and opcodes: SYM and $OPSYM$.

$$[A, M, SYM, OPSYM]$$

The presence or not of symbol, label and opsym fields is modelled by partial functions

$$lab, op, ref, num,$$

in the following way.

The domains of ref and num are disjoint, and since assembly instructions have either opcode, or operand (the latter being referential or numeric) the union of the domains of op, ref, num equals A. Two axiomatic descriptions model assembly and machine language requirements.

$$
\begin{array}{|ll}
lab : A \pfun SYM & \text{A1} \\
op : A \pfun OPSYM & \text{A2} \\
ref : A \pfun SYM & \text{A3} \\
num : A \pfun \mathbb{N} & \text{A4} \\
\hline
\mathrm{dom}\, ref \cap \mathrm{dom}\, num = \varnothing & \text{A5} \\
\mathrm{dom}\, op \cup \mathrm{dom}\, num \cup \mathrm{dom}\, ref = A & \text{A6}
\end{array}
$$

$$
\begin{array}{|ll}
opcode : M \pfun \mathbb{N} & \text{M1} \\
operand : M \pfun \mathbb{N} & \text{M2} \\
\hline
\mathrm{dom}\, opcode \cup \mathrm{dom}\, operand = M & \text{M3}
\end{array}
$$

Axioms $M1 \ldots M3$ will be referred to as 'Assembly Context2'. Mnemonics for translation of opcode in assembler to opcode in machine are represented by the following axiomatic description.

$$
\begin{array}{|ll}
mnem : OPSYM \pfun \mathbb{N} & \text{M4}
\end{array}
$$

This will be referred to as 'Assembly Context3'.

4.5.3 Assembly Process in Z

Input of assembly instructions and output of machine instructions are sequences $seqa?, seqm!$ respectively. The mathematical modelling of the assembler is by functions such as those described, and is treated fully in Hayes, while there is only sufficient detail in this chapter for the specification and subsequent animation to be understood. For instance, a label must be unique to a position; the composite $seqa? \comp lab$ is "one-one" and therefore the inverse of $seqa? \comp lab$ is also a partial function. The inverse of $seqa? \comp lab$ is $symtab$ and it plays an important role in maintaining the integrity of the assembler, determining the operand of machine instructions when the corresponding assembler operand is referential.

When the assembly operand is numeric, we can form the composite function:

$seqa? \mathbin{;} num$ and it can be seen that the union of $seqa? \mathbin{;} num$ and $seqa? \mathbin{;} ref \mathbin{;} symtab$ equals the operand of the machine output.

$$
\begin{array}{|ll}
\hline
\multicolumn{2}{l}{\textit{Assembly}} \\
seqa? : \mathrm{seq}\, A & \\
seqm! : \mathrm{seq}\, M & \\
\hline
\exists\, symtab : SYM \rightarrowtail\!\!\!\!\!\rightarrow \mathbb{N} \bullet & \mathrm{S2} \\
\quad symtab = (seqa? \mathbin{;} lab)^{-1} & \mathrm{S1} \\
\quad \mathrm{ran}(seqa? \mathbin{;} ref) \subseteq \mathrm{dom}\, symtab & \mathrm{S3} \\
\quad \mathrm{ran}(seqa? \mathbin{;} op) \subseteq \mathrm{dom}\, mnem & \mathrm{S4} \\
\quad seqm! \mathbin{;} operand = (seqa? \mathbin{;} ref \mathbin{;} symtab) \cup (seqa? \mathbin{;} num) & \mathrm{S5} \\
\quad seqm! \mathbin{;} opcode = seqa? \mathbin{;} op \mathbin{;} mnem & \mathrm{S6} \\
\hline
\end{array}
$$

The specification of the assembler in [51] includes a two-phase implementation. The first phase is captured by *Phase*1, where the machine instructions are constructed for inputs with numeric operands. Two tables are built: *rt* which records the positions where symbols are referenced, and *st* which records the values of the symbols. The sequence *core* of machine instructions captures the state of these instructions, which are partly constructed during phase one. The information about this intermediate state is modelled by *IS*.

$$
\begin{array}{|l}
\hline
\textit{IS} \\
st : SYM \leftrightarrow \mathbb{N} \\
rt : \mathbb{N} \rightarrowtail\!\!\!\!\!\rightarrow SYM \\
core : \mathrm{seq}\, M \\
\hline
\end{array}
$$

During Phase 1, *rt*, *st* will be constructed and the opcode and numeric fields of the machine instructions will be provided with their final values in *core*. (The sequence *core* can be thought of as acting as a ‘place-holder’ for the output machine sequence.)

$$
\begin{array}{|ll}
\multicolumn{2}{l}{\underline{\quad Phase1 \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad}} \\
seqa? : \mathrm{seq}\, A & \\
IS & \\
\hline
st = (in \mathbin{;} lab)^{-1} & \text{P1.1} \\
rt = (seqa? \mathbin{;} ref) & \text{P1.2} \\
(\mathrm{dom}\, rt) \lhd (core \mathbin{;} operand) = (seqa? \mathbin{;} num) & \text{P1.3} \\
(core \mathbin{;} operand) = (seqa? \mathbin{;} op \mathbin{;} mnem) & \text{P1.4} \\
\mathrm{ran}(seqa? \mathbin{;} op) \subseteq \mathrm{dom}\, mnem & \text{P1.5} \\
\hline
\end{array}
$$

During Phase 2 the input sequence of assembler code is not accessible and so information about the symbolic and numeric operand fields is obtained only from the reference table. During this phase the construction of the operands for the output machine instructions is completed.

$$
\begin{array}{|ll}
\multicolumn{2}{l}{\underline{\quad Phase2 \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad}} \\
IS & \\
seqm? : \mathrm{seq}\, M & \\
\hline
st \in SYM \nrightarrow \mathbb{N} & \text{P2.1} \\
\mathrm{ran}\, rt \subseteq \mathrm{dom}\, st & \text{P2.2} \\
(seqm! \mathbin{;} opcode) = (core \mathbin{;} opcode) & \text{P2.3} \\
(\mathrm{dom}\, rt) \lhd (seqm! \mathbin{;} operand) = (\mathrm{dom}\, rt) \lhd (core \mathbin{;} operand) & \text{P2.4} \\
(\mathrm{dom}\, rt) \lhd (seqm! \mathbin{;} operand) = (rs \mathbin{;} st) & \text{P2.5} \\
\hline
\end{array}
$$

To obtain the final implementation, the schemas are conjoined and *IS* is hidden for the intermediate state is not normally significant:

$$Implementation \mathrel{\widehat=} Phase1 \land Phase2 \setminus (st, rt, core).$$

In Hayes, *Implementation* is expanded out and it is proved that it is the same as *Assembly*.

4.5.4 Translation of Assembly to Gödel

The three axiomatic definitions:

*AssemblyContext*1	A1 .. A6
*AssemblyContext*2	M1 .. M3
*AssemblyContext*3	M4

are declared as schemas, named as above. They are then treated as if they were declared in the signature of *Assembly*. The bases, constants and binding functions are first declared, followed by the predicates required for data and for the four schemas. Functions `Bind1 .. Bind8` bind the variable names to Gödel variables, as for the previous examples. The full code is in Appendix C, a fragment only is presented here. The following shows the given set 'types' and an example of a binding:

```
% given sets
 Sym, A, M, Opsym, Int, Label, Op.

% example of bindings
FUNCTION    Bind1 :  Var * Set(OP(A, Sym)) ->  BindVar.
```

Decorations on Set names, ie priming, input, output are as before. For example

```
FUNCTION    IN : Var  ->  Var.
```

Predicates `IsSym`, `IsA` etc characterise symbols, assembly etc, for example

```
PREDICATE IsSym : Sym.
```

The data and predicate definitions are presented in the local part of module `Assembly` where, for example symbols and variables are as follows:

```
CONSTANT
% symbols
 V1, V2, Loop, Exit, Label : Sym;
% variable names
 SeqA, SeqM, Lab, Op, Ref, Num, Opcode, Operand, Mnem  : Var;
```

To ensure that data types can be collected together as a set, `IsA, IsM` are predicate definitions. The given sets are modelled by intentional set definitions, for example

```
opsym  =  {x : IsOpSym(x) }
```

models *OPSYM*. The natural numbers, $\mathbb{N}$ are modelled by a subset, $0..5000$. (This is because the partial function predicate requires a set.) Axioms $A1 \ldots A6$, $M1 \ldots M4$ are modelled by the Assembly Context 'schemas', and the code is commented to indicate the correspondence. The header and first few literals of Context 1 are

```
   SchemaType([ Bind1(Lab,lab),  Bind2(Op,op), Bind1(Ref,ref),
        Bind3(Num ,num) ], Assembly_context1) <-
  % given sets associated with declarations
```

```
        a = {x : IsA(x) }  &
        sym =  {x : IsSym(x) } &
        opsym  =  {x : IsOpSym(x) } &
        int = {x : 0 < x < 5000 } &
         .....
  % The rest of the code capture axioms  A5 to A6
```

The head (only) is shown for Assembly Context 2:

```
   SchemaType([Bind4(Opcode , opcode), Bind4(Operand,operand)],
                     Assembly_context2) <-
```

The assembler schema is modelled by conjoining Assembly Context schemas to the variable declarations of the input and output sequences. The sequences are checked by the Gödel predicate `IsSeq`, (which presents the lengths of the sequences as unknowns). Several auxiliary values are required, for example '`DomContents`' and '`RanContents`' for the domains and ranges of the various functions. Other intermediate values are also required, e.g. `seqaop` for $seqa? \mathbin{⨾} op$.

```
% Assembler
SchemaType([Bind6(IN(SeqA),seqa), Bind7(OUT(SeqM),seqm), Bind1(Lab,lab),
        Bind2(Op,op),   Bind1(Ref,ref),
        Bind3(Num ,num), Bind4(Opcode , opcode),
        Bind4(Operand,operand),Bind5(Mnem, mnem)], Assembly) <-
     SchemaType([Bind1(Lab,lab),  Bind2(Op,op), Bind1(Ref,ref),
        Bind3(Num ,num)], Assembly_context1) &             % A1-A6
     SchemaType([Bind4(Opcode , opcode), Bind4(Operand,operand)],
        Assembly_context2) &                               % M1-M3
     SchemaType([Bind5(Mnem, mnem)], Assembly_context3) &  % M4
            ......
                  % The rest of the code capture axioms    S1-S6
```

IS, *Phase*1, *Phase*2 and *Implementation* are modelled in a similar fashion. *Phase*1 and *Phase*2 were conjoined to form *Implementation*, and the values st, rt were hidden, but *core* was necessary for the computation. The head only of *Implementation* is shown:

```
SchemaType([Bind6(IN(SeqA),seqa ), Bind7(OUT(SeqM),seqm), Bind7(Core, core),
        Bind1(Lab,lab),  Bind2(Op,op), Bind1(Ref,ref), Bind3(Num ,num),
Bind4( Opcode , opcode ), Bind4(Operand,operand ),
    Bind5( Mnem, mnem )], Implementation)  <-
```

Queries can be made which interrogate all schemas. In the case of the assembler and its implementation, all (or most) variables are instantiated and would have to be supplied in advance. For example, a query to *Assembly_context*1 has bindings replicating the values of Table 4.1 and can be found (in full) in Appendix C where *operand* has a value represented by

$$operand \Rrightarrow \{M1 \mapsto 100, M2 \mapsto 4095, M3 \mapsto 2, M4 \mapsto 8, M5 \mapsto 2, \\ M6 \mapsto 1, M7 \mapsto 9, M8 \mapsto 3\}.$$

The animation shown exercises all the schema predicates. Truncated forms of queries to *Assembly_context*1 and *Assembly* are presented here. These are commented to indicate their link(s) to software testing:

```
% This tests a schema component and is a type of unit test.
[Assembly] <-
     SchemaType([Bind1(Lab, {OrdPair(A1, V1) .. OrdPair(A9, Exit)}),
                 Bind2(Op, {OrdPair(A3, Load) .. OrdPair(A9, Return)}),
                  ......
                 Bind3(Num, {OrdPair(A1, 100) .. OrdPair(A4,8)})],
                              Assembly_context1).
Yes.

% Query to the 'Assembly' - all data is supplied and
% the response is a check for accuracy.
% This is both a 'functional' test (black-box)
% and a test to examine how the component schemas integrate together.
[Assembly] <-
seqa = {OrdPair(1, A1), OrdPair(2, A2), .. ,OrdPair(8, A8), OrdPair(9, A9)} &
seqm = {OrdPair(1, M1), OrdPair(2, M2), .. ,OrdPair(9, M9)} &
..
op = {OrdPair(A3, Load), OrdPair(A4, Subn),.. OrdPair(A9, Return)} &
opcode = {OrdPair(M3,1), .. ,OrdPair(M8, 3)} &
operand = {OrdPair(M1,100), ..,OrdPair(M8,3)} &
....

mnem = {OrdPair(Load, 1), OrdPair(Subn, 3), .. ,OrdPair(Return, 77)}   &
  SchemaType([Bind6(IN(SeqA),seqa), Bind7(OUT(SeqM),seqm), Bind1(Lab,lab),
                Bind2(Op,op), Bind1(Ref,ref), Bind3(Num ,num),
                Bind4(Opcode , opcode), Bind4(Operand,operand),
```

```
                Bind5(Mnem, mnem)], Assembly).
Yes
```

It can be seen that the queries check that the output assembly stream correctly models the input stream; no new values are calculated (apart from intermediate ones). In each case the answer set is a singleton.

The two phase design of the assembler can also be animated. The code is in Appendix C. For the first phase, $Phase1$, the assembly instructions are input, and also *core* (because its definition cannot be computed using `Lib`, as it is implicit). Two possible values are provided for the machine operand. The first was the intermediate value, where only 'numeric' fields are represented:

$$operand \Rrightarrow \{M1 \mapsto 100, M2 \mapsto 4095, M8 \mapsto 3\}.$$

The second was the final value, equivalent to the binding above.

Each value was successful, so although *operand* need only have numeric fields represented after the first phase, the final value is also consistent with $Phase1$. *rt* and *st* are also computed during Phase 1:

```
rt = {OrdPair(3,V2),OrdPair(5,V2),OrdPair(6,V1),OrdPair(7,Exit),
      OrdPair(8,Loop)}
st = {OrdPair(Exit,9),OrdPair(Loop,3),OrdPair(V1,1),OrdPair(V2,2)}
```

For Phase 2, the values of rt, st computed by Phase 1 were input, together with values as for *Assembly*. As can be seen this phase will only accept the final values for operand, not the intermediate ones. For *Implementation*, the values provided were as for *Assembly*, with the addition of *core*. These values are checked and found to be consistent. In the original case study it is proved that *Implementation* is the same as *Assembly* and these tests provide an alternative demonstration and are complementary to the proof process in that they 'make visible' an assembly process in an intuitive manner. In contrast, proof is able to provide generality.

4.5.5 Comparison of Gödel and Prolog Versions

The Gödel animation can be compared with the Prolog animation. The advantages identified in Section 4.2.2 were all found to exist. In Prolog non declarative features such as 'cut' were used (for example in composition of functions) to aid implementation of much of the set theory code, compromising its correctness, whereas in Gödel no such devices were necessary. Thus the library implementation in Gödel was a straightforward replication in first order logic of the Toolkit definitions. The library implementation in Gödel was also more extensive than the Prolog library. This allowed the phase and implementation schemas to be modelled, whereas this was not possible for the Prolog animation.

The Gödel animation allowed a direct translation of existential quantification via `SOME`. In the Prolog version the expression:

$$\exists\, symtab : SYM \rightarrowtail\!\!\!\!\!\!+\; \mathbb{N} \bullet \ldots$$

was translated by *not* providing *symtab* with a name, but relying on the fact that variables on the right hand side of the implication which do not appear on the left are assumed existentially quantified.

Also in Gödel the 'types' $OPSYM, OP, SYM$ etc. are correctly modelled using 'types' rather than by 'setof' and the Prolog list. The Gödel types also allowed the modelling of the Z bindings by functions $Bind_i$, so that a different binding function was required for each pair of types, thus ensuring the correct replication of the bindings in Z.

4.6 Animation Example 2

The Unix file system case study was chosen because it involves 'state change' (unlike the assembler) and also contains more complex Z constructions such as the θ formation. This allows the coverage of more of the Z syntax.

4.6.1 Unix File System

The second case study involves the animation of a specification of the 'Unix file system'. A full account of the original specification is in [84] and only a fragment is presented here. The file storage system allows files to be stored and retrieved using file identifiers; the set of all file identifiers is called *FID*. Further given sets are channel identifiers *CID*. A channel identifier is a Unix file descriptor.

$$[FID, CID]$$

In order to support random access to files for reading and writing, a channel is defined which remembers a file and the current position in the file.

$$\begin{array}{|l}
\hline \textit{CHAN} \\
\textit{fid} : \textit{FID} \\
\textit{posn} : \mathbb{N} \\
\hline
\end{array}$$

$CHAN$ consists of a file identifier, fid and a position within a file, $posn$. The next schema shows the channel $CHAN'$ after some operation; each variable is primed.

$$\begin{array}{|l}
\hline \textit{CHAN}' \\
\textit{fid}' : \textit{FID} \\
\textit{posn}' : \mathbb{N} \\
\hline
\end{array}$$

The following expresses the property that the fid of a channel is never changed.

$$\begin{array}{|l}
\hline \Delta \textit{CHAN} \\
\textit{CHAN}, \textit{CHAN}' \\
\hline
\textit{fid}' = \textit{fid} \\
\hline
\end{array}$$

A channel storage system allows channels to be stored and retrieved using channel identifiers taken from CID. The channel storage system is denoted $cstore$ and is defined by the schema CS, where $cstore$ denotes a partial function between channel identifiers (the set CID) and the schema $CHAN$ represents a channel.

$$\begin{array}{|l}
\hline \textit{CS} \\
\textit{cstore} : \textit{CID} \nrightarrow \textit{CHAN} \\
\hline
\end{array}$$

The schema $openCS$ denotes the opening of a new channel. The old channel store is updated by the addition of a new channel whose file position is zero, but fid is unconstrained. The expression $\{cid! \mapsto \theta CHAN\}$ denotes a mapping from $cid!$ to the channel binding $\theta CHAN$.

$$\begin{array}{|l}
\hline \textit{openCS} \\
\Delta \textit{CS} \\
\textit{CHAN} \\
\textit{cid}! : \textit{CID} \\
\hline
\textit{cid}! \notin \operatorname{dom} \textit{cstore} \\
\textit{posn} = 0 \\
\textit{cstore}' = \textit{cstore} \oplus \{\textit{cid}! \mapsto \theta \textit{CHAN}\} \\
\hline
\end{array}$$

In the following schema, a channel is *closed*; the channel must have been previously open ($cid? \in \text{dom}\, cstore$) and *cstore* is updated by the removal of *cid*?, the input channel.

$$
\begin{array}{|l}
\hline
\textit{closeCS} \\
\Delta CS \\
cid? : CID \\
\hline
cid? \in \text{dom}\, cstore \\
cstore' = \{cid?\} \lhd cstore \\
\hline
\end{array}
$$

4.6.2 Gödel Code for the Unix File System

The following shows the base types and functions necessary to produce bindings within the types. The module imports `Lib`, the library necessary for functions and relations, a full version of which is in Appendix C. The code is commented to facilitate understanding. Schema (names) are decorated via the use of functions as before. Again, we need extensional sets, for integers form part of the source or target sets of partial functions.

```
% code for schema
% In this case, the second component is a binding
FUNCTION    Bind4 : Var * Set(OP(Cid, List(BindVar))) -> BindVar.

% Schema referencing
PREDICATE SThetaS: Set(List(BindVar)) * Name.
```

The code for schemas *CHAN*, $CHAN'$, $\Delta CHAN$, *CS*, and *openCS* is presented. Schemas which are included in the signatures of other schemas are modelled via their conjunction with other variable bindings (if any). The list of bindings is appended to the list of other variable bindings to become the binding of the schema which includes them. For example the bindings of *CHAN* and $CHAN'$ are appended to become the binding of $\Delta CHAN$. The relationships between variables in Z in the schema predicate is modelled by `Lib`. For example the predicate `FunOveride` represents function overriding, $\oplus$ in Z.

```
 %%%%%%Schema %%%%%%

SchemaType(binding, Chan) <-
      binding = [Bind1(Fid, f), Bind2(Posn,posn) ] &
```

```
% Types of variables, position is a natural number
      fid = {x :  IsFileId(x) } &
      posfile = {x : 0 =< x < 100} &
           posn >= 0 &
           posn In posfile &
           f In fid .

% Forms the set of bindings for a schema in simple cases
% where functions are not involved
SThetaS(val, schname) <-
     val = {binding : SchemaType(binding, schname)}.

SchemaType(binding, Cs) <-
        binding = [Bind4(Cstore, c)] &
         cid = {x :  IsCid(x) } &

         PF(c, cid, schtyp) &
         SThetaS(schtyp, Chan).

% The bindings are expanded out:
% Bind1, Bind2 are from declaration of CHAN
% Bind4 vars are from declaration of Cs and Cs'

SchemaType([Bind1(Fid, f), Bind2(Posn,posn), Bind3(OUT( ChId), outc),
  Bind4(Cstore, cs), Bind4(DSet(Cstore), cs1) ], OpenCs )  <-
% declared schemas
     SchemaType([Bind1(Fid, f), Bind2(Posn,posn)], Chan) &
      SchemaType([Bind4(Cstore, cs),Bind4(DSet(Cstore), cs1)], Del(Cs)) &
% schema predicate
        cid = {x :  IsCid(x) } &
        posfile = {x : 0 =< x < 100} &
          outc In cid &
        % cid not in dom cstore
          DomContents(cs, domcs) &
          ~(outc In domcs) &
           posn In posfile &
           posn = 0 &
 % Old channel store is updated by addition of a new
```

```
% channel whose file position is zero, but FileId is unconstrained
 FunOveride(cs1, cs, {OrdPair(outc, [Bind1(Fid, f), Bind2(Posn,posn)])} ).

SchemaType([Bind3(IN( ChId), inc),
  Bind4(Cstore, cs), Bind4(DSet(Cstore), cs1) ],CloseCs ) <-
       SchemaType([Bind4(Cstore, cs),Bind4(DSet(Cstore), cs1)], Del(Cs)) &
             cid = {x :  IsCid(x) } &
           inc  In cid &
          DomContents(cs, domcs) &
         inc In domcs  &
         DomExclude(cs1, {inc}, cs).
```

4.6.3 Example of Queries to Unix Files

The Unix file system expresses constraints in a constructive manner (like the small file system). However unlike the latter system, there is much less flexibility as to instantiation of variables. This is because many of the state variables are functions, and therefore of a more complex data type than *file*, *file′*. For example if *cstore′* from schema *openCS* is not instantiated in the original query, then it will not be provided with values by its declaration $cstore' : CID \nrightarrow CHAN$ and will require to be constructed via the schema predicate. This is the case here where the schema predicate provides such a construction. However in a non-constructive version of the specification, many or all of the variables would require instantiation in the original query, (as for the assembler) otherwise the query would flounder.

A number of possible queries and the responses are presented. The queries shown here represent black-box testing, of schemas *CS*, *openCS*. Tracing the code (as it is executed) also allows for the structure of the specification to be examined (but that is not shown). The first query investigates the possible bindings for *CS* which is provided with a binding equivalent to

$$cstore = \{cid1 \mapsto \langle fid \Rrightarrow F1, posn \Rrightarrow 2\rangle\}.$$

```
[UnixFiles] <- SchemaType ( binding, Cs ) &
     binding = [ Bind4 ( Cstore, { OrdPair ( Cid1, [ Bind1 ( Fid, F1 ),
                                  Bind2 ( Posn, 2 ) ])})].


binding = [Bind4(Cstore,{OrdPair(Cid1,[Bind1(Fid,F1),Bind2(Posn,2)])})] ? ;
```

There are no other values which satisfy the predicate. A further query shows the effect on the channel store, of the opening of a further channel. As before, the existing channel has file identifier F1 and position 2. The query provides the value of CS', the value after the new channel is opened, and information about the new channel. The new channel is added and can be any of the remaining (unopened) ones and the file identifier can be any. The position in the file remains at 0. In this case the answer set contains more than one binding.

```
[UnixFiles] <- cs = {OrdPair(Cid1,[Bind1(Fid,F1),Bind2(Posn,2)])} &
                 SchemaType([b1,b2,b3,Bind4(Cstore, cs),
                  Bind4(DSet(Cstore), cs1) ], OpenCs ).


b1 = Bind1(Fid,F1),
b2 = Bind2(Posn,0),
b3 = Bind3(OUT(ChId),Cid2),
cs = {OrdPair(Cid1,[Bind1(Fid,F1),Bind2(Posn,2)])},
cs1 = {OrdPair(Cid1,[Bind1(Fid,F1),Bind2(Posn,2)]),
     OrdPair(Cid2,[Bind1(Fid,F1),Bind2(Posn,0)])} ? ;
```

Initiating a back track gives a further answer:

```
b1 = Bind1(Fid,F5),
b2 = Bind2(Posn,0),
b3 = Bind3(OUT(ChId),Cid3),
cs = {OrdPair(Cid1,[Bind1(Fid,F1),Bind2(Posn,2)])},
cs1 = {OrdPair(Cid1,[Bind1(Fid,F1),Bind2(Posn,2)]),
     OrdPair(Cid3,[Bind1(Fid,F1),Bind2(Posn,0)])} ?
Yes
```

Thus amongst many possible values we have:

$$\begin{aligned} &cstore' = \{\, Cid1 \mapsto \langle fid \Rrightarrow F1, posn \Rrightarrow 2 \rangle, \\ &\qquad Cid2 \mapsto \langle fid \Rrightarrow F1, posn \Rrightarrow 0 \rangle \,\}. \end{aligned}$$

The new channel can be any except $Cid1$ and the file remembered by the new channel, any. Further animations are in Appendix C.

The animation was of a fragment of the Unix file system and was successful. Again, the translation was sophisticated; the longest wait for a query was of the order of a second. The perceived disadvantage is the limitation to finite sets. However the authors of the original Z specification note that file size, directory size "Inodes", device capacity and position within a file are *bounded*. Many specifications require only finite sets. Although the specification is small, it covers more advanced features of Z, including a simple occurrence of the binding formation θ. A translation of this feature was never attempted in the case of Prolog animation.

4.7 Conclusion

This chapter has investigated the *structure simulation method* and in particular the use of the LP, Gödel, for Z animation. and covers **Contribution 3**. The two studies demonstrate

1. The practicality of the method: the studies cover some important features of the Z notation, and in particular, the animation reflects the modularity of Z. This means that it will be easier to investigate any flaws in the specification uncovered by the animation;

2. The sophistication of the method: in the particular case of the assembler, the code appeared in the same order as the Z it modelled. There was no problem with floundering on account of the occurrence of non-ground variables. The translation differed from the equivalent Prolog. For successful execution in Prolog, *either* the code required ordering in a particular way *or* non-declarative features such as 'var' were used;

3. The efficiency of the method: interrogation of the code for the Unix file system involved complex queries, but response to these on a current SPARC Ultra is instantaneous;

4. The potential for proving correctness: there is the possibility of proving that the animation correctly represents the Z specification. Gödel is defined as a theory in first order logic and an implementation must be sound with respect to this semantics. Also, Gödel allows the representation of types and sets.

In short, the use of Gödel types replicates the types in Z, and the implementation of sets and first order logic captures the ZF basis for Z and this builds confidence that the information provided by the animator does indeed represent the information which would be provided by the specification if it could be interrogated.

Animation provides a way of testing a specification, and test cases for animation have been suggested which have a similar purpose to test cases for software. How much of the data needs to be instantiated and how much supplied depends on the application. At the one extreme, the small file system constructs all data from its declarations and at the other the assembler requires all data to be supplied. Animation does not replace proof, but complements it in that tests can be used to make visible the structure of the specification (as in the case of the assembler and its implementation). Although this is not shown here, it could also be used to demonstrate that a certain property is *not* true before the time-consuming effort of formal proof is attempted [85].

For the user of the animator to have confidence in the tool, assurance in its correctness is required, and this is the subject of the next chapter. However, for correctness to be proved, a formal framework is required. Since program synthesis presented problems, the formal framework is provided by *abstract approximation*, and this is the next subject of the thesis. The work on abstract approximation was completed after the Prolog animator of [119] was published, and represents a more recent advance.

Chapter 5

Abstract Approximation

5.1 Introduction

Chapter 4 described *structure simulation*, a set of animation rules for the translation of Z to the Gödel logic programming language and this chapter, in formalising these rules covers **Contribution 4** of this thesis. Structure simulation has been applied to two substantial case studies and has a potential for real world applications. The rules were evolved because an alternative method, *program synthesis*, proposed in Chapter 3 was found to have problems. The simulation rules were found to be practical, but lacked the formal framework for proving correctness which exists when synthesising a program.

This chapter provides a different approach to correctness, *abstract approximation*, introduced by Breuer and Bowen [17] to provide a formal framework and some proof rules for the correct animation of Z. In formalising the animation rules of Chapter 4, we both modify and strengthen the approximation method. Abstract approximation is based on the procedures of *abstract interpretation*, formalised by Cousot and Cousot in [23]. In this initial paper, abstract interpretation was used for static analysis of imperative programs. The abstract interpretation of the concrete semantics explicitly exhibits an underlying structure; the structure is implicitly present in the richer concrete structure of program executions. In abstract approximation the approximate semantics of the animations of Z is compared with the concrete semantics of Z associated with ZF set theory. The approximate semantics of the programming language underestimates the concrete semantics of Z.

The chapter commences with a brief discussion of abstract interpretation, supported by simple illustrative examples and this is provided in Section 5.2. Section 5.3 presents the framework and proof rules of [17] in a tutorial fashion and these are compared with the method and procedures of abstract interpretation (in Section 5.4). Although the work of Breuer and Bowen contains a framework for arbitrary specifications in Z, their method does not cover the possible existence of non-integer sets. In their approach, 'given sets' are not included and predicate values are based on $\{0, 1\}$. Their approach is a generic one based on a suitable declarative language and the prototype animator is in a lazy functional language (Miranda).

We present in Section 5.5 an animation based on a logic programming language with types and sets, such as the Gödel language described in Chapter 4. This approach is adopted because the logic programming language implementation reflects in a more natural way the first order logic of Z, and the sets and types can be used to model the data structures of Z. The translation rules from Chapter 4 *do* include the possibility of non-integer (given) sets and furthermore have a potential application to 'real world' examples such as the assembler and Unix file system.

Proof arguments for correctness are presented in Section 5.6 and the differences between our work and the work of Breuer and Bowen is provided where necessary. The chief difference is, as might be expected, in the interpretation of predicates and this is presented in Section 5.6.4. Another difference is that we allow for the structure of schema expressions (for example conjunction and disjunction) to be directly reflected in the animation, and this is presented in Section 5.6.7.

5.2 Brief Description – Abstract Interpretation

This section provides a brief introduction to the concepts of abstract interpretation and we commence by presenting some simple examples which provide the basic intuitions. A more complex example based on the work of the Cousots is described in Section 5.2.2, which also introduces concepts such as ordering which are used later in the chapter.

5.2.1 Description

The notion of relating a concrete semantics to an approximate one is not new. It is used (for example) to speedily check calculations in arithmetic and the correctness of physical formulae and these commonly used examples of abstract interpretation provide the intuitions behind the method.

1. 'Casting out 9' is used to check additions and multiplications. The result of the check is that the calculation is known to be 'wrong' or 'possibly_correct'. In this case the concrete interpretation of the calculation is Peano arithmetic and the abstract interpretation of the calculation is the method of 'casting out 9'. For example when checking the calculation '$1234 \times 32 = 39486$' we take the 'rest' r_1 of $1234 = (1+2+3+4) mod 9 = 1$, the rest r_2 of $32 = (3+2) mod 9 = 5$ and the rest r of result $39486 = (3+9+4+8+6) mod 9 = 3$. If we multiply r_1 and r_2 and take the rest, $p = (r_1 \times r_2) mod 9 = 5$. We should obtain $r = p$ if the calculation is 'possibly_correct'. In this case $r \neq p$ and so the calculation is 'wrong'. We use 'possibly' because even if $r = p$ the calculation could still be incorrect (e.g. '$1234 \times 32 = 5$'). This is a way of ensuring that the interpretation is 'safe'. This means that if a property of the concrete interpretation is promised, then it is guaranteed. In this case the property is that the answer to the calculation is wrong;

2. 'Dimensions in physical formulae' are used as a check that the formula is 'possibly_correct' or is 'wrong', as in the previous example. The abstraction is 'dimension calculus' which involves taking dimensions of both sides of a physical formulae. For example the formula for the period, T of a simple pendulum of length l is

   $$T = 2\pi(l/g)^{1/2}, \text{where } g \text{ is gravity.}$$

   The dimension of length is $[L]$, the dimension of time $[T]$, and the dimensions of acceleration are $[L][T]^{-2}$ and '2π' is a scalar quantity and dimensionless. Thus the dimensions on the right hand side of the formula are

   $$([L] \ / \ [L][T]^{-2})^{\frac{1}{2}}$$

   which evaluates to $[T]$, the dimension of the left hand side. This means that the formula is 'possibly_correct'. In this case, 'safeness' again means that if we decide a formula is incorrect, it is so.

Abstract interpretation was formalised in a seminal paper by Cousot and Cousot [23], who used it for static analysis of imperative programs. The abstract interpretation of the concrete semantics models the structure which is implicitly present in the richer concrete structure of program executions. A useful introduction to the concepts of semantic domains and approximation is contained in [97] and the detailed analysis of the application of abstract interpretation to 'casting out 9' and 'dimensions' can be found in [25]. The latter paper also provides an extension of the dimension calculus to type checking. Further summaries and applications of abstract interpretation are in [4, 24, 78].

Since the original paper, the work of the Cousots has been extended to declarative languages, including the application to groundness analysis in logic programming [24][1]. In this paper, abstract interpretation is applied to a program analysis which can be used to prove that a particular predicate is, or will be, bound to ground terms during program execution. However there may be predicates whose groundness cannot be determined. Abstract interpretation has been applied to many kinds of static analysis for logic programs. For example [9] describes data flow analysis for (constraint) logic programs which includes variable sharing and freeness as well as groundedness. Abstract interpretation is utilised in strictness analysis [4] in functional programming. The objective is to improve program efficiency by detecting when it is possible to pass arguments 'by value' rather than 'by need'. In this case the notion of 'safeness', is that an argument must never be detected as strict when it is not. On the other hand there may be strict arguments which are undetected by the abstraction.

The example in Section 5.2.2 is based on the seminal work in static analysis of imperative programs by Cousot and Cousot [23] where *finite flowcharts* are used to model the program semantics and abstract interpretation is then used for program de-bugging. The flowchart is represented by a directed graph, viz. a set $Nodes$, and a set $Arcs$, where $Arcs$ is a subset of $Nodes \times Nodes$. Each program point, or $arc \in Arcs$, is associated with a possible set of environments, where an environment is a possible binding for each identifier of the program. In order to accommodate incomplete information, sets associated with program executions can be 'lifted' by top and bottom elements $\top, \bot$ respectively. In particular the set of program identifiers is augmented by $\top_{Id}, \bot_{Id}$ to become semantic domain Id, where Id is a lattice

[1]The Cousots have published many papers on abstract interpretation which can be found at http://www.di.ens.fr/c̃ousot/COUSOTpapers/.

with an imposed ordering :

$$\forall\, id \in Id \bullet \bot_{Id} \quad \sqsubseteq \quad id \quad \sqsubseteq \quad \top_{Id}.$$

In future the subscript on $\bot, \top$ will be omitted. The appropriate domain or sub-domain for each top and bottom element will be understood by its context.

If the semantic domain of program values (booleans, integers etc.) is represented by the set *Val*, then the domain of possible environments *Env* is defined:

$$Env == Id \rightarrow Val.$$

The bottom value for *Env* is an environment e where $e(x) = \bot$ for all $x \in Id$, i.e. where all identifiers have undefined values, during the initial state of the program for example. *Env* is a total continuous function for $Env(\bot) = \bot$, $Env(x) = \bot$ for $x \in Id$ undefined in *Env*. The set of environments associated with each program point, or $arc \in Arcs$, is called a *context*. We have $context \in Contexts$, where

$$Contexts == \mathbb{P}\, Env.$$

Also associated with program arcs is an *abstract context*, $abs \in Abs$, which effectively *abstracts* or *approximates* the context associated with the arc in a manner described below. (This can be compared with the much simpler abstract interpretations involved in casting out nine and dimension calculus.) The nodes of a flow chart are associated with program commands and five types of node are associated with flow charts. These are *entries, exits, assignments, tests and junctions.* In general, the information associated with each arc is propagated through the program via concrete and abstract interpretations of program commands. However, the example is confined to an assignment node.

An abstraction function α maps between the concrete and abstract contexts associated with the arcs. For every node, the context (whether abstract or concrete) associated with its output arc(s) depends on the context(s) of its input arc(s). The program command can be interpreted in an abstract or concrete manner. The figure below (Figure 5.1) diagrammatically represents the action of the abstraction α and adjoint γ where

$$\alpha : Contexts \rightarrow Abs$$
$$\gamma : Abs \rightarrow Contexts.$$

The diagram illustrates the fact that abstracting concrete contexts for input arcs

$$\begin{tikzcd}
Abs \arrow[rr, "f_{abs}"] & & Abs \arrow[d, dashed, "\gamma"] \arrow[d, phantom, "\sqsubseteq", shift left=4ex, near end] \\
Contexts \arrow[u, dashed, "\alpha"] \arrow[rr, "f_{conc}"'] & & Contexts
\end{tikzcd}$$

Figure 5.1: Approximation Diagram for Abstract Interpretation

to a node, performing an abstract interpretation of a language construct, then concreting the resulting abstract, results in a loss of information compared with the concrete interpretation of the context. On the other hand, commencing with the abstract contexts on input arcs, concretising, then performing the concrete interpretation, introduces no loss of information, for the result is the same as if an abstract interpretation had been directly performed. α and γ are monotonic adjoint functions:

$$\alpha \circ \gamma = \mathit{IdentityFunction}_{abs},$$
$$\gamma \circ \alpha \sqsupseteq_{conc} \mathit{IdentityFunction}_{conc}.$$

An example is provided to illustrate the figure where, for simplicity, attention is confined to the case where the program has a single integer variable, x. The operation of the program determines the value of an expression at the single node. The example is needed to help explain the procedures of abstract interpretation, for the latter will, in turn, be compared with the abstract approximation, the principal subject of the chapter. Some of the concepts introduced in the example will also be required for abstract approximation. The example illustrates the abstractions of [4], but the assignment example is new and constructed with the intention of explaining the concepts in as simple as manner as possible.

5.2.2 Abstract Interpretation: Example

For this simple example, consider a single node, n, associated with an assignment `x := x*x - 2*x + 10`. In the case of a single integer variable the context associated with the input arc to n will be the set of possible values of the integer on the arc.

We shall consider the abstraction to be the *sign* of the integer values associated with the arc. In this case the concrete contexts are

$$Contexts == \mathbb{P}\,\mathbb{Z}.$$

The subset ordering is the ordering on the concrete contexts. The top element is $\mathbb{Z}$, and the bottom the empty set, $\{\,\}$. For the abstract contexts we have:

$$Abs == \{\perp, plus, zero, minus, \pm\}.$$

The element $\perp$ is associated with entry arcs where identifiers are undefined, thus the abstraction, α, is defined for $C \in Contexts$:

$$\begin{aligned}
&\alpha(C) = \perp, C = \{\,\};\\
&\alpha(C) = zero, C = \{0\};\\
&\alpha(C) = minus, C = \{x : x \leq 0\};\\
&\alpha(C) = plus, C = \{x : x \geq 0\};\\
&\alpha(C) = \pm, \text{otherwise}
\end{aligned}$$

and the concretisation, γ, is defined for $A \in Abs$:

$$\begin{aligned}
&\gamma(A) = \{\,\}, A = \perp;\\
&\gamma(A) = \{0\}, C = zero;\\
&\gamma(A) = \{x : x \leq 0\}, A = minus;\\
&\gamma(A) = \{x : x \geq 0\}, A = plus;\\
&\gamma(A) = \mathbb{Z},\ \text{otherwise.}
\end{aligned}$$

If the input context (possible input values) to n is the set of natural numbers $\mathbb{N}$, then (from elementary algebra) the context associated with the output arc of n is a member of the set of natural numbers greater than or equal to 9. Thus the concrete interpretation of the calculation for the assignment function in the concrete domain is

$$\{x : x \geq 9\}.$$

The abstract interpretation of the calculation is found using the 'rule of signs' as follows:

$$
\begin{array}{l}
\mathit{plus} \times \mathit{plus} = \mathit{plus};\ \mathit{plus} \times \mathit{minus} = \mathit{minus};\ \mathit{plus} \times \mathit{zero} = \mathit{zero};\\
\mathit{plus} + \mathit{plus} = \mathit{plus};\ \mathit{plus} - \mathit{plus} = \pm;\ \pm + \pm = \pm\\
\mathit{plus} \times \bot = \bot;\ \mathit{plus} + \bot = \bot;\ \mathit{plus} - \bot = \bot\\
\mathit{etc.}
\end{array}
$$

Since the input value of x in the abstract domain is *plus*, we see that our output value is

$$(\mathit{plus} \times \mathit{plus}) - (2 \times \mathit{plus}) + 10 = \pm.$$

It can be seen that by performing the abstract interpretation we have lost information, for by concretising the result of the abstract interpretation:

$$\gamma(\pm) = \mathbb{Z}$$

we see that the set is larger than that produced by the concrete interpretation for $\{x : x \geq 9\} \sqsubseteq \mathbb{Z}$.

The above is a simple example which demonstrates the method rather than a practical application. However it provides enough background to understand the similarities and differences of this work to the work described in the rest of the chapter and a comparison will be made in Section 5.4. A more useful and sophisticated version of the method illustrated by this simple example is interval analysis and this is used as a check on array bounds [24, 25].

5.3 Brief Description: Abstract Approximation

This section provides a brief introduction to the notion of *abstract approximation*, posited by [17] to determine the correctness of animations of Z. The section concentrates mainly on the generic aspects of abstract approximation. However for illustrative purposes we shall use logic programming examples, and in particular the animations of Chapter 4.

This is structured as follows. Abstract approximation compares the interpretation of Z syntactical objects in both the execution language (in our case the LP) and in Z. In Section 5.3.1 we present the subset of the Z syntax which is to be covered by the animation rules. The Z interpretation is the interpretation we would expect

Notation	Meaning
$t,\ t_i$	expressions
$x,\ x_i,\ \ x_i^j,\ X,\ X_i,\ X_i^j$	variable names
$P,\ p,\ p_i,\ p^i$	predicates (in general)
$CP,\ CP_i,\ CP^i,\ GCP,\ GCP_i,\ GCP^i$	*schema* predicates
$\tau,\ \tau',\ \tau_i,\ \tau_i^j$	types
$d,\ d_i,\ d^i,\ D,\ D_i,\ D^i$	declarations
$Sch,\ Sch_i,\ Sch^i$	schema names
$S,\ s$	set names

Table 5.1: Notation: Z Syntax

if we had been evaluating the objects using set theoretic (ZF) considerations. In addition, the Z domain has been extended to accommodate non-terminating computations and a description of this extended Z domain is presented in Section 5.3.2. The LP interpretation is according to the implementation of the theory of finite sets in the LP, however the details of the LP domain are deferred until Section 5.5. A comparison between abstract and concrete interpretations is in Section 5.3.3. We next determine the meaning of *loss of information* in abstracting the Z syntax, and the concept of *ordering* is presented in Section 5.3.4. This is in order for a comparison to be made between the abstracted interpretation and the concrete.

Details of this comparison of LP and Z interpretations are presented in Section 5.5. In brief, the comparison is as follows: for finite sets and when the LP terminates, the interpretations are the same, but for infinite sets, or when the LP fails to terminate, the LP interpretation underestimates the Z interpretation in a manner to be explained. Breuer and Bowen have formalised the process of showing that the abstract interpretation of a given piece of Z syntax always underestimates the concrete. The formalisation which involves a structural induction process is presented in Section 5.3.5. In order to aid the reader, a summary of notation used for Z syntax in the remainder of this chapter is provided in Table 5.1.

5.3.1 Z Syntax

The following is an outline summary of the Z syntax to be interpreted and should be taken in conjunction with the description of Z in Chapter 4 and in [105]. It includes those constructs which are *explicitly* covered by our application of abstract approximation. The given sets (and names of enumerated free types) are from the set $GIVEN$, the set of schema names $NAME$, and the set of variable names (within a

$$
\begin{aligned}
expr ::= {} & \mathbb{Z} \mid n \in \mathbb{Z} \text{ the integers and integer values} \\
& \mid t_1 + t_2 \mid t_1 - t_2 \ldots \mid \text{ an integer expression} \\
& \mid G_i \text{ where } G_i \in GIVEN \text{ a given set reference} \\
& \mid x_i \text{ where } x_i \in VAR \\
& \mid \{x_1 \ldots x_n\} \text{ an enumerated set} \\
& \mid t_1 t_2, \text{ function application} \\
& \mid (t_1, \ldots, t_n), \text{ a tuple} \\
& \mid t_1 \cup t_2 \mid t_1 \cap t_2 \mid \bigcup t \mid \ldots \\
& \qquad \text{set union, intersection, distributed union etc.} \\
& \mid \text{“}Enum\_Type ::= x_1 \mid \ldots \mid x_n\text{”} \\
& \qquad \text{where } x_i \in VAR, Enum\_Type \in GIVEN \\
& \qquad\quad \text{an enumerated free type} \\
& \mid \lambda\, d \mid p \bullet t \text{ where } d \in decl
\end{aligned}
$$

Figure 5.2: Z Syntax

schema) are VAR. We consider the following four parts of the Z syntax: Expressions, Predicates, Axiomatic Definitions and Schemas, denoted $expr, pred, axdef, schema$ respectively. Although declarations are not top-level it is convenient to treat them as syntactic objects, as suggested in [17].

A declaration can be made up of both variable declarations and schema references. However in our work, variable declarations and schema references will be interpreted separately, as will be seen. The following is the syntax of *basic declarations*, *basic_decl* and sequences of basic declarations *decl* where $x_i \in VAR, t, t_i \in expr, Sch \in NAME$, and Sch is a schema reference:

$$
\begin{aligned}
basic\_decl &::= x_1, \ldots, x_n : t \mid Sch \\
decl &::= bd_1;\ \ldots;\ bd_n \text{ where } bd_i \in basic\_decl.
\end{aligned}
$$

The rest of the Z syntax is presented in Figure 5.2 where $t,\ t_k \in expr$ (all k) and the expressions within " " are to be treated as complete syntactic objects, where '|' means just that. The syntax includes schema references which are not included in the version of the syntax presented by Breuer and Bowen for it is assumed they are expanded and absorbed into the schemas which reference them. The syntax replicates [17] with the exception that

1. *GIVEN* are not included by Breuer and Bowen;

2. we do not cover the arbitrary free types described in Chapter 4.

Predicates have the following syntax, where $p_1, p_2 \in pred$, $e_1, e_2 \in expr$:

$$\begin{aligned} pred ::= {} & p_1 \vee p_2 \mid p_1 \wedge p_2 \\ & \mid \text{“}\forall\, d \mid p_1 \bullet p_2\text{”} \mid \text{“}\exists\, d \mid p_1 \bullet p_2\text{”} \text{ where } d \in decl \\ & \mid e_1 = e_2 \mid e_1 \subseteq e_2 \mid e_1 \in e_2. \end{aligned}$$

The syntax for schemas, schema bindings, schema expressions and axiomatic definitions is presented below, where predicate p can be absent.

$$\begin{aligned} schema \quad ::= \quad & \text{“}Sch \mathrel{\widehat=} \left[\, d \mid p \,\right]\text{”} \mid \theta Sch \mid \{Sch \bullet \theta Sch\} \\ & \text{where } \; d \in decl, p \in pred, Sch \in NAME \\ & \mid \text{“}Sch \mathrel{\widehat=} Sch^1 \wedge Sch^2\text{”} \mid \text{“}Sch \mathrel{\widehat=} Sch^1 \vee Sch^2\text{”} \\ & \text{where } Sch, Sch^1, Sch^2 \in NAME \\ axdef \quad ::= \quad & \left[\, d \mid p \,\right], \text{ where } \; d \in decl, p \in pred. \end{aligned}$$

5.3.2 The Z Domain

In all that follows we are assuming that the specification we are animating is type-checked, and that (by implication) the animation is also type correct. The contents of the Z domain are driven by the animation rules of Chapter 4 where in order to animate given sets the user of the animator is required to instantiate them with suitable values. For example the set of file identifiers *FileId* might be represented by $\{F1, F2, F3\}$. The understanding is that these are all different values: it is as though *FileId* were defined as the free type:

$$FileId ::= F1 \mid F2 \mid F3.$$

This is similar to the way ‘deferred sets’ are instantiated in B AMN. The set *FileId* is then represented in the LP by the enumerated set `{F1, F2, F3}`. This means that the Z domain D also consists of instantiations of the given sets supplied by the user of the animator. Thus *GIVEN* consists of $\{G^1, \ldots, G^N\}$ and each G^k consists of distinct $g^k_1, \ldots g^k_n$.

The concrete domain consists of (i) the standard sets of Z, (ii) the non-standard sets and (iii) the instantiations of given sets. Requirements (i) - (iii) are formalised in Figure 5.3. The domain D_Z differs from reference [17] which does not include given sets; in addition the authors treat ‘non-standard sets for ZF’ separately from

$$
\begin{aligned}
D_Z ::= \; & n(\in \mathbb{Z}), \; \textit{an integer} \\
& \mid g_i^k, \text{ distinct elements of given sets} \\
& \qquad \text{where each instantiated } g_i^k \in G^k \\
& \mid (a_1, \ldots, a_n) \text{ where } a_k \in D_Z, \text{ a tuple} \\
& \mid \text{“}a_1 \mid \ldots \mid a_n\text{” where } a_k \in \mathit{VAR}, \text{ enumerated free type} \\
& \mid s \text{ where } s \in \mathbb{P}\, D_Z, \text{ a complete set} \\
& \mid s_{\cup\perp} \text{ where } s \in \mathbb{P}\, D_Z, \text{ an incomplete set} \\
& \mid \{x_1 \mapsto a_1, \ldots, x_n \mapsto a_n\} \text{ where } a_k \in D_Z, x_k \in \mathit{VAR} \\
& \qquad \text{(a finite symbol table)}
\end{aligned}
$$

Figure 5.3: The Z Domain

D_Z. The symbol table gives bindings of variable names to values and is used to express the value of a schema. Recall from Chapter 3 that schemas in Z are defined:

$$Sch \mathrel{\widehat{=}} \left[\, D \mid CP_1; \; \ldots; \; CP_m \,\right]$$

where each D is a declaration, associating n variables x_i with their types. The binding for Sch which follows provides values of x_i which satisfy CP_i. and is represented by

$$< x_1 \Rrightarrow a_1, \ldots, x_n \Rrightarrow a_n > .$$

This object is represented more simply by the symbol table

$$\{x_1 \mapsto a_1, \ldots, x_n \mapsto a_n\}$$

which is part of the existing syntax of Z. This is required so that a binding can be regarded as a syntactical object in Z, for a set of such bindings is the output of a schema interpretation in Z and in the LP.

5.3.3 Interpretations of Z Syntax

In abstract approximation, the abstract domain is the domain of the execution, here denoted D_{LP}, and the concrete domain D_Z the domain of the ZF interpretation. Given a syntactic expression, ϵ, of the Z notation, this can be interpreted in the abstract domain, D_{LP}, and in D_Z, as can be seen in Figure 5.4. The interpreted value in both D_{LP}, and in D_Z, in each case depends on the environment. The environments are, respectively, ρ_{LP}, ρ_Z and are assumed to be variables, while the syntactic expression ϵ is a constant. If $\rho_{LP} : VAR \twoheadrightarrow D_{LP}$ is an environment in

$$\begin{tikzcd}
\rho_{LP} \arrow[rr, "\mathcal{E}_{LP}[\![\epsilon]\!]_"] \arrow[d, dashed, "\gamma \circ _"'] & & D_{LP} \arrow[d, dashed, "\gamma"] \\
\rho_Z \arrow[rr, "\mathcal{E}_Z[\![\epsilon]\!]_"'] & & D_Z \arrow[u, phantom, "\sqsupseteq"']
\end{tikzcd}$$

Figure 5.4: Approximation Diagram for LP and Z domains

the execution domain D_{LP}, then $\gamma \circ \rho_{LP} : VAR \twoheadrightarrow D_Z$ is an environment in D_Z. If Σ_1 is the set of syntactic Z set and numeric expressions, then the evaluation in set-theoretic terms of expressions is the interpretation in D_Z, D_{LP} is respectively:

$$\mathcal{E}_Z[\![\epsilon]\!]\rho_Z = a_Z, \quad \mathcal{E}_{LP}[\![\epsilon]\!]\rho_{LP} = a_{LP}$$

where $\epsilon \in \Sigma_1$, $a_Z \in D_Z$ and $a_{LP} \in D_{LP}$, a *term*. For example, the syntactic expression: '$x + y$' in a Z environment

$$\{x \mapsto 2, y \mapsto 4\}$$

evaluates as

$$\mathcal{E}_Z[\![x + y]\!]\{x \mapsto 2, y \mapsto 4\} = 6.$$

In the LP $x + y$ is interpreted as a term and

$$\mathcal{E}_{LP}[\![x + y]\!]\{x \mapsto 2, y \mapsto 4\}$$

is implemented by

```
<- exp = x + y  & (x = 2) & (y = 4)
```

and evaluated by means of the ground substitution $\{x/2, y/4\}$ to '6'.

Where all the components are ground, a similar diagram can be constructed for the evaluators $\mathcal{P}_Z[\![\_]\!]\_$, $\mathcal{P}_{LP}[\![\_]\!]\_$ which interpret syntactic predicates '$p \in \Sigma_2$' in the Z and LP domains where a predicate will interpret its component expressions and return an appropriate boolean value. For example, the syntactic predicate expression: '$y \in \{1,2,3\}$' in a Z environment $\{x \mapsto 2, y \mapsto 3\}$:

$$\mathcal{P}_Z[\![y \in \{1,2,3\}]\!]\{x \mapsto 2, y \mapsto 3\} = y \in \{1,2,3\} \wedge (x = 2) \wedge (y = 3)$$

evaluates to '*true*'. $\mathcal{P}_{LP}[\![y \in \{1,2,3\}]\!]\{x \mapsto 2, y \mapsto 3\}$ becomes the query:

```
<- y In {1,2,3}  & (x = 1) & (y = 3)
```

again resulting in '*true*'. In the above examples all variables are ground; in the case where some variables are not ground, it is possible that predicate evaluation will result in the environment being updated and this is examined in Section 5.6.4. The syntactic objects *schemas, schema expressions* and *axiomatic definitions* also have their interpretation in Z and LP domains and will be treated in a subsequent section.

The implication of Figure 5.4 that the animation is an abstracted interpretation holding less information than the concrete is now made explicit. The concrete interpretation in D_Z *refines* the abstract in D_{LP}. Integers, sets, tuples etc. in D_{LP} correspond to integers, sets, tuples etc. in D_Z and the comparison is made in D_Z. For a particular implementation, a computation (in D_{LP}) can fail to terminate or provide an answer while determining the value of a term or predicate. An example can be seen in Chapter 4 where '$1/x$' was not evaluated for $x = 0$. This has the effect that expressions, predicates, and so on can be incomplete, or contain elements which are themselves incomplete. Thus in order to accommodate incomplete computations, the concrete domain also contains $\perp$. The nature of these incomplete elements and the different manner in which they occur in the work of Breuer and Bowen and in ours will be made apparent later.

5.3.4 Ordering in the Z and Execution Domains

Figure 5.4 supposes an ordering on the elements of the Z and execution domains. This ordering will now be made explicit, recalling that the ordering derives from the possible non-termination of the execution. This ordering is in respect of all types of domain elements. If a, b are integers then the ordering $\sqsubseteq$ is

$$a \sqsubseteq b \Leftrightarrow (a = \perp \ or \ a = b).$$

Sets can be incomplete: if set s belongs to the concrete domain, the incomplete set 'tagged' with subscript $\cup\perp$ denoted $s_{\cup\perp}$ also belongs to the concrete domain. The notion of incomplete sets arises from the fact that a set can be partially output from an execution and then no more values are provided and this possibility needs to be represented in D_Z also. Where the animation is implemented by a functional language, incomplete sets model the occurrence of lazy lists. An example of an incomplete set in a logic programming language which can be seen in Chapter 4 is a set of 'answer substitutions' where a set is only partially output. This will be revisited in Sections 5.6.7 and 5.5.3.

The refinement relation, $\sqsubseteq$ on D_Z is equality on integers (as above) and co-ordinatewise on tuples. It uses the following powerdomain orderings on subsets, as described in [17]. For 'complete sets' $D_1, D_2 \in \mathbb{P}\, D$:

$$D_1 \sqsubseteq D_2 \Leftrightarrow \\ (\forall\, d_1 : D_1 \bullet \exists\, d_2 : D_2 \bullet d_1 \sqsubseteq d_2) \wedge (\forall\, d_2 : D_2 \bullet \exists\, d_1 : D_1 \bullet d_2 \sqsupseteq d_1).$$

For example, $\{1, 2, 3, \perp, 4\} \sqsubseteq \{1, 2, 3, 4, 5\}$.

For 'incomplete sets':

$$(D_1)_{\cup\perp} \sqsubseteq D_2 \Leftrightarrow (D_1)_{\cup\perp} \sqsubseteq (D_2)_{\cup\perp} \Leftrightarrow (\forall\, d_1 : D_1 \bullet \exists\, d_2 : D_2 \bullet d_1 \sqsubseteq d_2).$$

For example, $\{1, 2, 3, 4\}_{\cup\perp} \sqsubseteq \{1, 2, 3, \perp, 4, 5\}$.

Equality is defined:

$$(D_1)_{\cup\perp} \sqsubseteq (D_2)_{\cup\perp}, (D_1)_{\cup\perp} \sqsupseteq (D_2)_{\cup\perp} \Leftrightarrow (D_1)_{\cup\perp} = (D_2)_{\cup\perp}.$$

In general the incomplete sets are 'non-standard' with respect to ZF. For example $\{1, 2, 3, 4\}_{\cup\perp} \sqsubseteq \{1, 2, 3, \perp, 4\}_{\cup\perp}$ and $\{1, 2, 3, 4\}_{\cup\perp} \sqsupseteq \{1, 2, 3, \perp, 4\}_{\cup\perp}$ so the two sets are 'equal' – however they do not have the same elements. (It could be said that the two sets contain the same 'information' – see [47].) The following subsection formalises the approximation condition of Figure 5.4.

5.3.5 Rules for Approximation

The rules which follow are from [17] and they will be used, where appropriate, to prove that the approximation in the LP is correct. Figure 5.4 represents the fact that if ϵ is a syntactic Z expression then condition **AR1** must hold for a correct animation of Z in D_{LP}:

Approximation Rule 1 (AR1)

$$\gamma(\mathcal{E}_{LP}[\![\epsilon]\!]\rho_{LP}) \sqsubseteq \mathcal{E}_Z[\![\epsilon]\!](\gamma \circ \rho_{LP}).$$

The following conditions form the basis of a structural induction rule in which if it can be shown that **AR1** holds for syntactic variable $\epsilon = x$, then it also holds for syntactical expression $\epsilon = fx$. For example, f might be the syntactic operator '$\cup$' on variable tuple $\epsilon = (x_1, x_2)$. We denote by f_Z, f_{LP} the interpretation in the Z domain and LP domain respectively of the syntactic expression fx. Thus if fx is set union, then $f_Z x$ is the set theoretic evaluation of set union and $f_{LP} x$ is the induced operation in D_{LP} of set union. In order to show **AR1**, the following must hold for the operators f of Z on variables x:

Condition 1 In order to prove correctness it is necessary to show that the interpretation in D_{LP} is built recursively for each operator of Z, acting on each syntactic Z expression.

$$f_{LP}(\mathcal{E}_{LP}[\![x]\!]\rho_{LP}) = \mathcal{E}_{LP}[\![fx]\!]\rho_{LP}$$

Condition 2 A further condition is a property of Z, i.e. the manner in which expressions in the Z domain are evaluated:

$$f_Z(\mathcal{E}_Z[\![x]\!]\rho_Z) = \mathcal{E}_Z[\![fx]\!]\rho_Z.$$

However this condition is only true for complete sets and is not in general true for incomplete sets;

Condition 3 The third condition is the key one, which encapsulates the approximating mechanism:

$$\gamma(f_{LP}(\mathcal{E}_{LP}[\![x]\!]\rho_{LP})) \sqsubseteq f_Z(\gamma(\mathcal{E}_{LP}[\![x]\!]\rho_{LP})).$$

Conditions 1–3 provide the basis of a structural induction rule:
Structural Rule for Induction:
If **Conditions 1–3** hold for all $\rho_{LP} : VAR \twoheadrightarrow\!\!\!\!\!\!+ \; D_{LP}, x : \Sigma$, then **AR1** for $\epsilon = x$ implies **AR1** $\epsilon = fx$.
PROOF

$$\gamma(\mathcal{E}_{LP}[\![x]\!]\rho_{LP}) \sqsubseteq \mathcal{E}_Z[\![x]\!](\gamma \circ \rho_{LP}) \qquad \text{[Base Case]}$$
$$f_Z(\gamma(\mathcal{E}_{LP}[\![x]\!]\rho_{LP})) \sqsubseteq f_Z(\mathcal{E}_Z[\![x]\!](\gamma \circ \rho_{LP}))$$
$$[f_Z \text{ Monotone for environment with complete sets}]$$
$$\gamma(f_{LP}(\mathcal{E}_{LP}[\![x]\!]\rho_{LP})) \sqsubseteq f_Z(\mathcal{E}_Z[\![x]\!](\gamma \circ \rho_{LP})) \qquad [\text{cond. 3}, \sqsubseteq \text{transitive}]$$
$$\gamma(f_{LP}(\mathcal{E}_{LP}[\![x]\!]\rho_{LP})) \sqsubseteq \mathcal{E}_Z[\![fx]\!](\gamma \circ \rho_{LP}) \qquad [\text{cond. 2}, \rho_Z = \gamma \circ \rho_{LP}]$$

$$\gamma(\mathcal{E}_{LP}[\![fx]\!]\rho_{LP}) \sqsubseteq \mathcal{E}_Z[\![fx]\!](\gamma \circ \rho_{LP}) \qquad \text{[cond. 1]}$$

$\square$

The base types for induction include integers, instantiations of given sets, sets of integers and variables. However since **Condition 2** is only true for complete sets, then **AR1** can only be used for complete sets. In order to encompass incomplete sets, we need to extend ZF operations. A further induction rule is presented, as in [17]: **AR2** is implied by **AR1**, provided that we interpret f_Z for incomplete sets in such a way that is is monotonic in the refinement relation for incomplete sets.

Approximation Rule 2 (AR2)

Recall $\rho_{LP} : VAR \nrightarrow D_{LP}$ is an environment in the execution domain D_{LP}, then $\gamma \circ \rho_{LP} : VAR \nrightarrow D_Z$ is an environment in D_Z and write $\rho_Z = \gamma \circ \rho_{LP}$. Consider ρ'_Z environment in D_Z which refines ρ_Z, viz. $\rho_Z \sqsubseteq \rho'_Z$. Then **AR2** is

$$\rho_Z \sqsubseteq \rho'_Z \Rightarrow \gamma(\mathcal{E}_{LP}[\![\epsilon]\!]\rho_{LP}) \sqsubseteq \mathcal{E}_Z[\![\epsilon]\!]\rho'_Z.$$

AR2 is stronger than **AR1**, for if we take $\rho_Z = \rho'_Z$, then **AR2** becomes **AR1**. Conversely, assuming the monotonicity of f_Z, then **AR1** implies **AR2**.

If **Conditions 1–3** hold for all $\rho_{LP} : VAR \nrightarrow D_{LP}, x : \Sigma$, and f_Z is monotone, then **AR2** for $\epsilon = x$ implies **AR2** for $\epsilon = fx$.

5.4 Comparison of Abstract Interpretation and Abstract Approximation

The resemblances between abstract interpretation and abstract approximation can be seen in Figure 5.5, which contains the approximation diagrams for each concept.

Section 5.2 describes the use of abstract interpretation in static analysis of imper-

$$\begin{tikzcd}
Abs \arrow[r, "f_{abs}"] & Abs \arrow[d, dashed, "\gamma"] \\
Contexts \arrow[u, dashed, "\alpha"] \arrow[r, "f_{conc}"'] & Contexts \arrow[r, phantom, "\sqsubseteq"] & {}
\end{tikzcd}
\qquad
\begin{tikzcd}
\rho_{LP} \arrow[r, "\mathcal{E}_{LP}[\![\epsilon]\!]_"] \arrow[d, dashed, "\gamma \circ _"'] & D_{LP} \arrow[d, dashed, "\gamma"] \\
\rho_Z \arrow[r, "\mathcal{E}_Z[\![\epsilon]\!]_"'] & D_Z \arrow[r, phantom, "\sqsupseteq"] & {}
\end{tikzcd}$$

Figure 5.5: (i) Abstract Interpretation and (ii) Abstract Approximation

ative programs and diagram **(i)** of Figure 5.5 pictures the abstract interpretation of program environments at a program node. The abstractions are of the contexts associated with a program node. The figure indicates the loss of information when concretising the result of an abstract interpretation. Diagram **(ii)** of Figure 5.5 represents loss of information when interpreting a piece of Z syntax in an abstract execution domain as compared with the concrete 'Z' domain.

Diagrams **(i)** and **(ii)** are similar in that they both represent an abstract and concrete interpretation of a piece of syntax. However for abstract interpretation, the abstraction is a *set descriptor*, whereas for abstract approximation integers, sets, tuples in the abstract correspond to integers, sets, tuples in the concrete. However the abstract object may not be as well defined as the associated concrete object. Abstract approximation also represents loss of information, for the reason that a program may terminate, or only provide a partial answer. There is also a difference in the manner in which both represent lack of information. In abstract interpretation the *top* element corresponds to the *least* precise information, the set $\mathbb{Z}$. The *most* precise information is given by the empty set, and corresponds to $\perp$. The abstract interpretation is an *upper* approximation. As pointed out by [78], the ordering is opposite to the ordering of domain theory; the *top* element corresponds to total lack of information. Abstract approximation has the ordering of domain theory; the *bottom* element corresponds to total lack of information and it is a *lower* approximation.

Abstract interpretation involves two comparisons. First of all the concrete context inputs are abstracted (via α) and an abstract interpretation of a language construct performed. The resulting abstraction is concreted (via γ) then compared with the concrete interpretation of the context and a loss of information is found. The second comparison involves commencing with the abstract contexts on input arcs, concretising, via γ then performing the concrete interpretation. The result is the same as if an abstract interpretation had been directly performed: the second comparison results in no loss of information. On the other hand abstract approximation explicitly involves only *one* comparison: a syntactic object is interpreted in an environment in the abstract (execution) domain and the value concreted (via γ). This value is compared to that found by concreting an execution environment and interpreting the same syntactic object in the resulting concrete environment in a concrete (Z) domain. In that case the first result underestimates the second. For abstract approximation the notion of 'safeness' is that the abstract approximation will always provide correct information. However there may be little or no information provided if the program fails to terminate. The principle of safeness means that we do not want to output the *wrong* information, for it may mislead.

The remainder of this chapter is restricted to the application of the generics of abstract approximation to the logic programming domain. Although the examples supplied are in Gödel, the framework is intended to apply to any logic programming language with sound semantics and with sets and types.

5.5 Formalising Structure Simulation

This section formalises the translation of Z syntax to a logic program. The assumption is that the specification has been translated to a logic program and that the user queries this program, as in the case of the simple example and case studies of Chapter 4. Section 5.5.1 outlines how a specification might be parsed in order to apply the rules and explains how a more detailed, formal set of rules are required for the structure simulation rules of Chapter 4 to be proved correct. The formal rules are based on the syntax of Z. Section 5.5.2 presents an overview of the animation approach in initialising and querying the specification. Section 5.5.3 describes the representation of the expressions and sets of Z in the LP domain and Section 5.5.4 defines the concretisation function γ in mapping between the abstract and concrete domains.

5.5.1 Parsing the Specification and Applying the Translation Rules

The rules for structure simulation presented in Chapter 4 are of a general nature and a more detailed set are presented in Section 5.6. In order to translate a Z specification automatically it needs to be *parsed*. A report which suggests a suitable grammar script for Z is [18]. The script is suitable for input to the publically available PRECC[2] utility. The authors suggest that PRECC parsers are particularly useful for Z in that the same concrete symbols can be read by PRECC in different ways according to their context (as a schema name or as another identifier for example). The FUZZ type checker [105] also includes a parser.

Both the formal detailed rules for structure simulation and correctness proofs are based on the syntax of Z. The correctness proofs use structural induction, which is not based on any particular order of rule application, and hence the translation

[2] A PREttier Compiler-Compiler

process is *confluent*. The parsing and translation would depend on the 'normal' syntax of Z. If it was decided to re-define parts of the Z syntax (such as operator precedence for example) then this would have to be allowed for.

5.5.2 Overview

The user of the animator supplies some values for the constants and sets of the specification, followed by an initial imposed environment ρ^o. If the initial environment is consistent with the constraints of the specification, an answer set is output which provides a set of values for the other schema variables. For example, in Chapter 4 schemas are represented in the LP by characteristic predicates which define the relationship between schema variables. A schema may have other schemas as references or be defined in terms of schema expressions (schema conjunction for example). A top level call to a schema will provide the answer substitution set, which models the appropriate set of schema bindings, where a single binding is of the form:

$$< x_1 \Rrightarrow a_1, \ldots, x_n \Rrightarrow a_n >$$

or

$$[Bind_1(X_1, a_1), \ldots, Bind_n(X_n, a_n)]$$

in the LP. The characteristic predicate for the schema (in the LP) is from the *interpreted* syntax of the schema definition and includes syntactical objects such as set expressions, predicates, declarations and references to other schemas. Suppose $Env_{LP} == VAR \twoheadrightarrow\!\!\!\!\!\!\shortmid\;\; D_{LP}$ is the set of all possible environments associated with a specification, then $\rho^o_{LP} \in Env_{LP}$ for the characteristic predicates to succeed. The user is then supplied with the set of answer substitutions satisfying each schema, which will have been constrained by its environment $\rho^o_{LP} \in Env_{LP}$. At the end of the execution the environment ρ^o_{LP} will have been enhanced to one of a set of environments $\rho_{LP} \in Env_{LP}$ and each answer substitution will conform to ρ_{LP}.

Note that for terminating computations of expressions we are interested *either* in a value which terminates with success (and there may be many such values contributing to many schema bindings) *or* in a finite failure. In Nicholson et al. [88], the continuation semantics of Prolog is presented. However this is not our approach – we are not interested in the computational processes (for example backtracking) which accompanies such a search. We assume that some variables are initially instantiated, and some are initially undefined and are subsequently computed. (Whereas

Nicholson et al. are concerned with the possible *changing* values of variables as the program executes.) Non-termination or floundering occurs when it is not possible to compute the undefined values.

This section describes how the integers, given sets and derived expressions of Z are abstracted by the *declarative semantics* of a logic programming language. The integers, $\mathbb{Z}$, and instantiated given sets G^k form the basis of domains D_{LP} and D_Z. Syntactic expressions of Z considered are schemas, predicates and expressions such as arithmetical and set expressions of Figure 5.3. The *output* is confined to schema bindings, as in Chapter 4. Thus evaluations of arithmetical, set and expressions *other than* these, take place *as part of a program execution to determine or check schema bindings.* A description is presented in the next subsection, of how the Z syntax is interpreted in the LP, and of how the outputs can be ordered.

5.5.3 The Logic Programming Domain

Recall that the given sets are denoted $GIVEN$, the set of schema names $NAME$, and the set of variable names (within a schema) are VAR. The variable and schema names will subsequently be interpreted as constants in D_{LP}, which means confining them to allowed constant names in the programming language. However, in most of what follows, $x_1, \ldots x_n$ etc. will be used to 'stand for' the variable names.

The proposed abstract domain, D_{LP}, includes representations of integer values, instantiated values, tuples, bindings and sets. n-Tuples are represented by functions of arity n and sets are represented both as *terms* and as answer sets. $GIVEN$ is captured by declaring $\{G^1, \ldots, G^N\}$ as bases of the program and for each base G^k is declared the constants $g_1^k, \ldots g_n^k$. In order to 'collect together' the constants to form a set, for each base a predicate is constructed denoting membership. For example predicate `IsFileId` is applied thus: `IsFileId(F1) IsFileId(F2)` etc. The *set* of file identifiers is formed by set comprehension as in Section 5.2.2.

In the execution domain D_{LP}, the refinement ordering is generated by:

$$\forall x : D_{LP} \bullet \perp \sqsubseteq x$$

and by recursion on the representation of sets and tuples. The partial element $\perp$ of D_{LP} represents an incomplete computation, when the program fails to terminate. The formalisation involves *set terms* as previously described. It also involves *sets of answer (substitutions)*, where the latter are only considered in the case of schema outputs.

5.5.3.1 Set Objects in the LP

During execution, one or more computations may fail to terminate and this has the following effect on set objects:

1. Set terms: recall that (finite) set terms are represented by

   $$\{a_1, a_2, \ldots, a_n\}, \text{ or } a_1 \circ (\ldots (a_n \circ \varnothing))))$$

   where each a_i is itself a term. The following set contains an incomplete element:

   $$(\perp \circ (a_1 \circ (\ldots (a_n \circ \varnothing))))$$

   and when a computation of a set term fails to terminate, in an attempt to evaluate an infinite set for example, we obtain the set:

   $$(a_1 \circ (\ldots (a_n \circ \perp))).$$

   In both cases the set evaluates to '$\perp$' since functions are strict. This bottom element is designated $\perp \circ \perp$ to distinguish it as a set and the equivalent of $\varnothing_{\cup\perp}$ in D_Z. We have, for all set a:

   $$\begin{aligned} a \cup (\perp \circ \perp) &= (\perp \circ \perp) \cup a = (\perp \circ \perp) \\ a \cap (\perp \circ \perp) &= (\perp \circ \perp) \cap a = (\perp \circ \perp). \end{aligned}$$

   Note that the above applies to terms in an execution which fails to terminate, rather than to terms in their initial program state, for these may very well be undefined;

2. Sets of answer substitutions: recall that for some schema Sch, a binding is denoted in the LP:

   $$\begin{aligned} &\theta Sch = [Bind_1(X_1, a_1), \ldots, Bind_n(X_n, a_n)] \\ &\qquad \text{where } a_k \in D_{LP}, X_i \in VAR, Sch \in NAME \end{aligned}$$

   and that the answers to a query concerning the characteristic predicate of Sch provide a subset of

   $$\{Sch \bullet \theta Sch\}$$

which depends on the values instantiated. However an answer set can output some results then fail with an error message. An example would be a schema with

```
(x = 1) \/ ({x, 1, 2, 3} = {1, 2, 3, 4})
```

in the predicate as in Chapter 4. Whether the answer set contains some or indeterminate answers depends on the way it is evaluated (generally it echoes the code order). Thus there is no way of knowing, from the output, the nature of the rest of the set. This set is an example of the incomplete set of Section 5.3.4 where, if $b_i, (i = 1 \ldots k)$ is a schema binding the incomplete set of answers can be denoted

$$\{b_1, \ldots b_k\}_{\cup\perp}$$

where no more answers are provided after the k^{th} which is followed by the output of an error message, as in the example.

Figure 5.6 contains the abstract (or LP) representation of Z expressions which is defined recursively via *terms* in the logic programming language. D_{LP} is sup-

$D_{LP} ::= m, m \textit{ an integer}$
$\quad | \; g_i^k,$ where each g_i^k is base G^k
$\quad | \; Tn(a_1, \ldots, a_n)$ where $a_k \in D_{LP}$, a tuple
$\quad | \; \{a_1, \ldots, a_n\}$ where $a_k \in D_{LP}$, enumerated free type
$\quad | \; \{a_1, \ldots, a_n\}$ where $a_k \in D_{LP}, a_k \neq \perp$, an complete set *term*
$\quad | \; \{a_1, \ldots, \perp, \ldots a_n\} (= \perp \circ \perp)$ where $a_k \in D_{LP}$
$\quad | \; \{a_1, \ldots, a_n\}_{\cup\perp} (= \perp \circ \perp)$ where $a_k \in D_{LP}$,
$\quad | \; Bind_i(X_i, a_i)$ where $a_i \in D_{LP}, X_i \in VAR$
$\qquad$ a single variable binding
$\quad | \; [\; Bind_1(X_1, a_1), \ldots, Bind_n(X_n, a_n)\;]$, where $a_k \in D_{LP}, X_i \in VAR$,
$\qquad$ a schema type $- -$ a single schema binding

Figure 5.6: The Interpretation of Expressions in the LP Domain

plemented by the following answer set for a schema, of complete and incomplete schema bindings:

$$D_{LP} ::= \{b_1 \ldots b_n\} \mid \{b_1 \ldots b_n\}_{\cup\perp}$$

where each b_i is of the form:

$$[\ Bind_1(X_1, a_1^i), \ldots, Bind_N(X_n, a_n^i)\ ].$$

In addition, for each base type G^k there is a unary predicate IsG^k which is applied to each constant, g_i^k, of the base[3]. Bindings of variable names to values, $Bind_i(X_i, a_i)$ provide an environment, and also interpret schema types, as described.

5.5.4 Concretisation Function γ

Expressions of D_{LP} are mapped to expressions of D_Z, and predicates of D_{LP} to predicates of D_Z. $\perp$ of D_{LP} maps to $\perp$ of D_Z. A concretisation function $\gamma : D_{LP} \rightarrow D_Z$ is constructed which maps to D_Z from an abstract domain D_{LP} recursively as follows: integers in D_{LP} map to integers in D_Z, instantiated values of given sets map to instantiated values in D_Z, set terms map to sets, and functions, $Tn(a_1, a_2, \ldots, a_n)$, map to n-tuples. If a_i is a term in the LP and $X_i \in VAR$, G a 'typical' base type representing a given set and g a 'typical' member. The mapping γ for *terms* is defined in Figure 5.7. We now define how γ maps the *answer substitutions* which

$$
\begin{array}{lcl}
\gamma(m) & = & m, m \text{ an integer} \\
\gamma(g) & = & g, \text{ constant of base type } G \text{ is} \\
 & & \text{mapped to instantiated element } g \\
\gamma(\{a_1, \ldots, a_n\}) & = & \{\gamma(a_1), \ldots, \gamma(a_n)\}, \\
\gamma(\perp \circ (a_1 \circ (\ldots (a_n \circ \varnothing)))) & = & \gamma(\perp \circ \perp) = \varnothing_{\cup\perp} \\
\gamma(a_1 \circ (\ldots (a_n \circ \perp))) & = & \gamma(\perp \circ \perp) = \varnothing_{\cup\perp} \\
\gamma(Tn(a_1, \ldots, a_n)) & = & Tn(\gamma(a_1), \ldots, \gamma(a_n)), \\
\gamma([Bind_1(X_1, a_1), \ldots, Bind_n(X_n, a_n)]) & = & \{X_1 \mapsto \gamma(a_1), \ldots, X_n \mapsto \gamma(a_n)\} \\
 & & \text{a single schema binding} \\
\gamma(\perp) & = & \perp
\end{array}
$$

Figure 5.7: γ: LP terms

model schema bindings. A complete (incomplete) set of schema bindings maps to a complete (incomplete) set of schema bindings. If binding b_i is an answer substitution for $i = 1 \ldots m$ then Figure 5.8 shows the mapping of a complete and an incomplete

[3]Where the meaning is apparent we shall in future remove super and subscript from instantiated elements and given sets and use g, G, respectively.

$$\begin{array}{lcl} \gamma(\{b_1 \ldots b_m\}) & = & \{c_1 \ldots c_m\} \\ \gamma(\{b_1 \ldots b_k\}_{\cup\perp}) & = & \{c_1 \ldots c_k\}_{\cup\perp} \end{array}$$

where

$$b_i = [\ Bind_1(X_1, a_1^i), \ldots, Bind_n(X_n, a_n^i)\ ],$$
$$c_i = \{X_1 \mapsto \gamma(a_1^i), \ldots, X_n \mapsto \gamma(a_n^i)\},$$

for $i = 1 \mathrel{..} m$

Figure 5.8: γ: Answer Substitutions

set of b_i. Where the set is incomplete we obtain the set up to element k, $0 \leq k < m$, then no more values.

5.6 Correctness: Proof Arguments

The structural induction process is intended to show that the answer set output from the LP for a given query *abstracts* or underestimates the answer set expected from the Z interpretation. We need to determine how a given piece of Z syntax will be interpreted in the LP and Z domains in a given environment. The basis for the induction is the integers and given sets of the specification.

5.6.1 Structural Induction: Strategy

Evaluation functions in the LP are $\mathcal{E}_{LP}[\![\_]\!]$ for numerical and set expressions, $\mathcal{P}_{LP}[\![\_]\!]$ for predicates, $\mathcal{D}_{LP}[\![\_]\!]$ for declarations, and $\mathcal{S}_{LP}[\![\_]\!]$ for schemas. Evaluation functions are provided in the Z domain in a similar manner. The proofs that these correctly approximate the Z domain are outlined. They can be summarised as follows:

- For any computation, if the memory bounds are exceeded, the computation results in $\perp$ and underestimates the Z interpretation;

- For integers and integer expressions, such as addition, subtraction, two cases are considered, the case where all integers are within the *MinInt*, *MaxInt* of the LP implementation, and the case where they are not. In the case where the integers exceed these bounds, the LP evaluates to '$\perp$' and always underestimates the Z interpretation. The details are in Section 5.6.2;

- Sets are 'non-standard' for Z if they are incomplete, for they do not adhere to the ZF axioms. Sets are 'non-standard' for the LP if they contain incomplete elements or are infinite. For 'sets as terms' rule **AR1** is sufficient. Where a program terminates the approximation is exact and the details are contained in Section 5.6.3. In many cases, it is more convenient to use **AR1** directly, rather than **Conditions 1-3**;

- Section 5.6.3 also contains the cases where sets have incomplete elements or are incomplete or are infinite. When this refers to sets and set expressions involving *terms* and not to sets of answer substitutions, we treat these together. This is because in all of these cases they evaluate to $\perp \circ \perp$ in the LP and always underestimate the ZF interpretation.

  In the case of set union and distributed union, two methods are pursued, the method where the interpretations of Figure 5.4 are interpreted *directly*, and the method utilising **AR2**. This is for illustrative purposes and because the interpretation of schemas is formalised in such a way that it depends on distributed union. However for subsequent proofs, only the direct method is pursued;

- Predicates can both provide a boolean answer *and* update the environment, from ρ_{LP}, to ρ'_{LP} (say). It can happen that the environment can be updated in a number of different ways, thus providing a set of answer substitutions. The update is extended to all literals conjoined to the literal being evaluated. This is the result of the *resolution inference rule* of logic programming (Section 3.3.1). Resolution is presented in a novel way, in the form of three constraint satisfaction rules. The interpretation of predicates is not completed until set comprehension is completed, because of the occasional non-determinism of the updates. However if a computation fails to terminate, the predicate evaluates to $\perp$, which underestimates the Z interpretation. This is detailed in Section 5.6.4;

- The proofs for set comprehensions and variable declarations are presented in Section 5.6.5. Variable declarations are treated in a similar manner as predicates, where it is possible for a value to provided for a variable by declaring it to be (for example) a member of a known set. The interpretation of these predicates allows the environment to be updated in both Z and the LP;

- For schema bindings output by the LP which form an answer set, it is possible to obtain an incomplete set. In that case we cannot assume standard ZF operations and we cannot use **AR1** directly. Thus a potential answer substitution might fail to terminate and in that case the answer set is incomplete and rule **AR2** is used.

The base types for the induction are (i) integers, (ii) sets of integers, (iii) given sets and their instantiated elements and (iv) variables, so the first task is to show how their interpretation in the LP underestimates the interpretation in Z. Induction is over each Z construct and is shown in Appendix D. Only the novel or most salient parts are presented in this chapter. The induction takes place in the following order:

1. Numeric expressions;
2. Set expressions (union and distributed union);
3. Predicate expressions: infix;
4. Set comprehension and variable declarations;
5. Predicates: quantified expressions (which depend on declarations);
6. Schemas and Schema Expressions.

5.6.2 Base Types

(i) Integers

The assumption is that there are largest positive and negative integers available in the system, *MaxInt*, *MinInt*, which cannot be exceeded. Any attempt to do so may cause the computation to terminate. Thus for $m \in \mathbb{Z}$:

$$\mathcal{E}_{LP}[\![\mathrm{m}]\!]\rho_{LP} = m = \mathcal{E}_Z[\![m]\!]\rho_Z, \quad -\mathit{MinInt} \leq m \leq \mathit{MaxInt}$$
$$\mathcal{E}_{LP}[\![\mathrm{m}]\!]\rho_{LP} = \perp, m < -\mathit{MinInt} \text{ or } m > \mathit{MaxInt}.$$

($\perp$ may be implemented by the output of an error message, or alternatively to the character ∞. The latter is suggested by the IEEE floating point standard.) Thus since $\gamma(\perp) = \perp$:

$$\gamma(\mathcal{E}_{LP}[\![m]\!]\rho_{LP}) \sqsubseteq \mathcal{E}_Z[\![m]\!]\rho_Z, m \in \mathbb{Z}.$$

(ii) Sets of integers s

Suppose s is a subset of $\{i : \mathbb{N} \mid -MinInt \leq i \leq MaxInt\}$ and assuming that the memory bounds are not exceeded, then the abstract interpretation is *exact*.

Where $MinInt, MaxInt$ are exceeded, (for example $s = \mathbb{Z}$) then s is interpreted as $\bot \circ \bot$ in the LP, and therefore underestimates its interpretation in the Z domain.

$$\gamma(\mathcal{E}_{LP}[\![\{i : -MaxInt \leq i \leq MaxInt\}_{\cup\bot}]\!]\rho_{LP}) = \gamma(\bot \circ \bot) = \varnothing_{\cup\bot} \sqsubseteq \mathcal{E}_Z[\![\mathbb{Z}]\!]\rho_Z.$$

(iii) Given sets and their instantiated elements

Suppose G, g is a given set and typical element. These are interpreted in the LP by base type G, associated constant g and predicate IsG. In each case the abstract interpretation is exact for:

$$\gamma(\mathcal{E}_{LP}[\![g]\!]\rho_{LP}) = g$$
$$\gamma(\mathcal{E}_{LP}[\![G]\!]\rho_{LP}) = \gamma(\{x : IsG(x)\}) = G.$$

(iv) Variables

In section 5.5.2 we explained that schema bindings of variable names to values are constructed as $Bind_i(X_i, x_i)$, where variable 'names' are linked to variables in the LP. The value of a variable x_i can be obtained as a 'lookup' in the environment, where env_{LP} interprets the LP environment ρ_{LP} as in Andrews [6]. Assuming that the variable has a defined value, the trivial interpretation in the LP is

$$(x_i = a_i) \longleftarrow env_{LP}$$

which can be denoted:

$$\mathcal{E}_{LP}[\![x_i]\!]\rho_{LP} = a_i \Leftrightarrow (x_i = a_i) \ \& \ true \quad (a_i \neq \bot).$$

If the variable is undefined because of finite failure, the answer returned is *false*:

$$\mathcal{E}_{LP}[\![x_i]\!]\rho_{LP} = \bot \Leftrightarrow (a_i = \bot) \ \& \ false.$$

If the variable is associated with a schema binding, there are *no* values which satisfy the instantiated variables and the schema predicate, so no answer substitutions. For further discussion see Section 5.6.7.

If the variable is undefined because of non-termination or floundering, the answer returned is $\bot$:

$$\mathcal{E}_{LP}[\![x_i]\!]\rho_{LP} = \bot \Leftrightarrow (a_i = \bot)\ \&\ \bot.$$

In either case this is an exact approximation of the interpretation in D_Z, since $\rho_Z = \gamma \circ \rho_{LP}$.

5.6.3 Numeric and Set Expressions

Details about numerical expressions are in Appendix D where if fx is a numerical expression:

$$\gamma(\mathcal{E}_{LP}[\![fx]\!]\rho_{LP}) \sqsubseteq \mathcal{E}_Z[\![fx]\!]\rho_Z,$$

so the abstract evaluation underestimates the concrete.

We next apply the rules to set expressions, beginning with set union. Set operators such as intersection and power set are a special case of set comprehensions and will be treated in Section 5.6.5.

5.6.3.1 Set Union

For tutorial reasons we consider set union directly rather than commencing from $\bigcup$ and specialising. The syntactic expression '$x_1 \cup x_2$' is interpreted via an equivalent 'term' in the LP, denoted '$x_1 \cup_{LP} x_2$'. (In Gödel, 'union' is provided by a function '+'.) Consider, first, sets which are complete and with complete elements and suppose:

$$x_1 \mapsto a_1, x_2 \mapsto a_2 \in \rho_{LP}.$$

The expression $x_1 \cup_{LP} x_2$ is evaluated using the LP ground substitution $\{x_1/a_1, x_2/a_2\}$ so that $(x_1 \cup_{LP} x_2)\{x_1/a_1, x_2/a_2\}$ evaluates to $(a_1 \cup_{LP} a_2)$. We assume that $\cup_{LP}$ is set-theoretic and implements $\cup$ for finite sets in the same manner as $\cup$ for ZF. (See Appendices B, C.) The interpretation of ZF operations on sets is built recursively.
Condition 1 becomes:

$$f_{LP}(\mathcal{E}_{LP}[\![(x_1, x_2)]\!]\rho_{LP}) = a_1 \cup_{LP} a_2 = \mathcal{E}_{LP}[\![x_1 \cup x_2]\!]\rho_{LP},$$

which will hold for set operations for terminating computations.
Condition 2
If x_1, x_2 are complete sets, $\gamma(x_1, x_2)$ in D_Z evaluates in the expected way to $(\gamma(a_1), \gamma(a_2))$

and

$$f_Z(\mathcal{E}_Z[\![(x_1, x_2)]\!]\rho_Z) = \gamma(a_1) \cup_Z \gamma(a_2) = \mathcal{E}_Z[\![x_1 \cup x_2]\!]\rho_Z.$$

Since $\cup_{LP}$ is set-theoretic then $\gamma(a_1 \cup_{LP} a_2) = \gamma(a_1) \cup_Z \gamma(a_2)$ and **Condition 3** becomes:

$$\begin{aligned}&\gamma\ (f_{LP}\ (\mathcal{E}_{LP}[\![(x_1, x_2)]\!]\rho_{LP}))\\ &\quad = \gamma(a_1 \cup_{LP} a_2) = \gamma(a_1) \cup_Z \gamma(a_2) = f_Z(\gamma(a_1, (a_2))) = f_Z(\gamma(\mathcal{E}_{LP}[\![(x_1, x_2)]\!]\rho_{LP})).\end{aligned}$$

In other words the computation is exact for terminating computations. There are two ways of extending the result to non terminating computations.

1. The first way is to provide an extension of union to incomplete sets and use **AR2** as the proof rule. This is the rule used by [17] since their implementation is in Miranda, a functional programming language, which incorporates 'lazy lists'. If

   $$x_1 = a_{\cup\perp}, x_2 = b,$$

   we define the extension for union:

   $$(a_{\cup\perp} \cup_Z b) = (a \cup_Z b_{\cup\perp}) = (a \cup_Z b)_{\cup\perp}$$

   which is pointwise monotonic in the Z domain for non-standard sets. **Conditions 1–3** thus hold when 'f' is '$\cup$', thus: since **AR2** holds for x_1, x_2, then **AR2** holds for $\epsilon = x_1 \cup x_2$, where $\cup$ is pointwise monotonic for both standard and non-standard sets;

2. The second way is to interpret figure 5.4 directly and we see that in D_{LP}, if x_1 is not a standard finite set it can only have the value $x_1 = \perp \circ \perp$ and the left hand side of **AR1** for $\epsilon = x_1 \cup x_2$ is

   $$\gamma\ ((\perp \circ \perp) \cup_{LP} b) = \gamma\ (\perp \circ \perp) = \varnothing_{\cup\perp},$$

   since all set terms in the LP involving $\perp \circ \perp$ evaluate to $\perp \circ \perp$. Then since

   $$x_1 \mapsto \varnothing_{\cup\perp}, x_2 \mapsto \gamma(b)$$

are both members of environment $\rho_Z(= \gamma \circ \rho_{LP})$, the right hand side of the ordering relationship becomes:

$$\varnothing_{\cup\perp} \cup_Z \gamma(b)$$

which will, in any case always exceed $\varnothing_{\cup\perp}$, whatever its value, provided that it is still type correct. Since we have established conditions (1 – 3) for complete sets and **AR1** *directly* for incomplete or infinite sets, then **AR1** holds when 'f' is '$\cup$'.

5.6.3.2 Distributed Union

The interpretation of distributed union $\bigcup x$ is provided in the example in Section 5.2.2. $y = \bigcup x$ is modelled by `DUnion(arga, argb)` where the first argument is a set of sets (input) and the second their union (output). We denote `DUnion(_, _)` by $\bigcup_{LP}$ and it is defined so that it implements $\bigcup$ for finite sets in the same manner as $\bigcup$ for ZF. The argument that the LP interpretation underestimates the Z interpretation follows in a similar fashion to the argument for $\cup$.

As will be seen, the set of schema bindings (where non-empty incomplete sets can occur) is expressed as a distributed union. For this reason we *are* interested in the details of incomplete sets and need to look at **AR2**. Consider, first, the evaluation of $\bigcup x$ where $x = \{a_1 \ldots a_n\}$ is a complete, finite set in the LP environment. Then x is $\gamma(\{a_1 \ldots a_n\}) = \{\gamma(a_1) \ldots \gamma(a_n)\}$ in the Z environment.

$$f_{LP}(\mathcal{E}_{LP}[\![(x)]\!]\rho_{LP}) = \bigcup_{LP}(\{a_1 \ldots a_n\}) = \mathcal{E}_{LP}[\![\bigcup x]\!]\rho_{LP} = \mathcal{E}_{LP}[\![fx]\!]\rho_{LP}.$$

Thus **Condition 1** will hold for set operations for terminating computations.
Condition 2
If x is complete and involves only complete sets, the interpretation in D_Z evaluates in the expected way, $\gamma(x) = \{\gamma(a_1) \ldots \gamma(a_n)\}$ and we have

$$f_Z(\mathcal{E}_Z[\![(x)]\!]\rho_Z) = \bigcup_Z\{\gamma(a_1) \ldots \gamma(a_n)\} = \bigcup_Z \gamma(\{a_1 \ldots a_n\}) = \mathcal{E}_Z[\![\bigcup x]\!]\rho_Z.$$

Since $\bigcup_{LP}$ is set-theoretic then: $\gamma(\bigcup_{LP}(\{a_1 \ldots a_n\}) = \bigcup_Z(\{\gamma(a_1) \ldots \gamma(a_n)\})$ and **Condition 3** becomes:

$$\begin{aligned}&\gamma\ (f_{LP}\ (\mathcal{E}_{LP}[\![(x)]\!]\rho_{LP}))\\ &\quad = \gamma(\bigcup_{LP}\{a_1 \ldots a_n\}) = \bigcup_Z\{\gamma(a_1) \ldots \gamma(a_n)\} = f_Z(\gamma(\mathcal{E}_{LP}[\![(x)]\!]\rho_{LP})).\end{aligned}$$

Since it is equivalent to the set-theoretic definition then `DUnion(arga, argb)` approximates exactly in its interpretation of $\bigcup x$ in the case where x is complete. Thus **AR1** is true for complete sets.

When the induction process involves only set *terms* we can use **AR1** for incomplete or infinite sets, for the left hand side evaluates to $\varnothing_{\cup\perp}$, which always exceeds the right hand side. However we shall subsequently use $\bigcup$ as part of the induction process, for sets of bindings. Since it is possible to obtain an incomplete set of schema bindings, we need to extend the operation $\bigcup_Z$ to incomplete sets or sets containing incomplete sets. The only restriction is that the extension must be monotonic and the following extension:

$$\bigcup_Z \{u_{\cup\perp}, v, w\} = \bigcup_Z \{u, v, w\}_{\cup\perp},$$
$$\bigcup_Z (t_{\cup\perp}) = (\bigcup_Z t)_{\cup\perp}$$

is monotonic. It is then necessary to use **AR2** as the proof rule. Thus by **AR2**, the LP interpretation underestimates the Z interpretation for $\bigcup$ for incomplete sets. Thus $\bigcup$ is interpreted exactly for complete sets and underestimates where sets involved are infinite or non-standard.

5.6.4 Predicate Expressions

The evaluator $\mathcal{P}_{LP}$ interprets syntactic predicates p in the LP domain in the manner expected, where a predicate evaluates to $\perp$ when a program flounders or fails to terminate during its evaluation. Thus if

$$Bool_Z = \{tt, \mathit{ff}, \perp\}$$
$$Bool_{LP} = \{true, false, \perp\}$$

then

$$\gamma(true) = tt, \gamma(false) = \mathit{ff}, \gamma(\perp) = \perp.$$

We also have:

$$\mathcal{P}_{LP}[\![P_1 \wedge P_2]\!]\rho_{LP} = (\mathcal{P}_{LP}[\![P_1]\!]\rho_{LP} \;\&\; \mathcal{P}_{LP}[\![P_2]\!]\rho_{LP} = true) \Leftrightarrow$$
$$(\ (\mathcal{P}_{LP}[\![P_1]\!]\rho_{LP} = true)\ \&\ (\mathcal{P}_{LP}[\![P_2]\!]\rho_{LP} = true)\ ).$$

The approximation requirement, **AR1** for predicates becomes:

$$\gamma(\mathcal{P}_{LP}[\![\epsilon]\!]\rho_{LP}) \sqsubseteq \mathcal{P}_Z[\![\epsilon]\!](\gamma \circ \rho_{LP}).$$

5.6.4.1 Infix Predicates

In this subsection we treat *infix* predicates only: $=$, $\subseteq$, $\in$. Quantification predicates $\forall\, d \mid p \bullet q$ and $\exists\, d \mid p \bullet q$, where d is a declaration will be treated after we have treated declarations in Section 5.6.5.1.

In an LP, infix predicates $p \in \Sigma_2$ of the form $p(x_1, x_2),\ x_1, x_2 \in \Sigma_1$ are interpreted in such a way that they potentially provide *enhancements* to the existing environment as well as evaluating to boolean values. They differ fundamentally from the functional implementation suggested by [17]. Furthermore, if there are predicates conjoined to the infix predicates their environments are also enhanced in a manner which will be described. In order to compare the interpretations in the LP and in Z, we need to extend the Z interpretation so that it also encompasses constraint satisfaction. There are three constraint properties associated with predicate evaluation. Suppose $\mathcal{I}$ is an infix predicate, standing for equality, subset or membership. Then if either (or both) x_1 or x_2 is undefined or only partially defined they can become ground through resolution. We call this property:

Constraint Property 1:

$$\mathcal{P}_{LP}[\![x_1 \mathcal{I} x_2]\!]\rho_{LP} = \mathcal{P}_{LP}[\![x_1 \mathcal{I} x_2]\!]\rho'_{LP} = \mathit{true}$$

where $\rho'_{LP} = \rho_{LP} \oplus \{x_1 \mapsto a_1, x_2 \mapsto a_2\}$.

The environments of predicates conjoined to the infix predicates are also enhanced:

Constraint Property 2:

$$\mathcal{P}_{LP}[\![P \wedge (x_1 \mathcal{I} x_2)]\!]\rho_{LP} = \mathcal{P}_{LP}[\![(x_1 \mathcal{I} x_2) \wedge P]\!]\rho_{LP} = \mathcal{P}_{LP}[\![P]\!]\rho'_{LP}$$

where $\rho'_{LP} = \rho_{LP} \oplus \{x_1 \mapsto a_1, x_2 \mapsto a_2\}$.

An example which illustrates both properties is

```
[Lib] <- ([1, 2, 3, y] = [1, x, 3, 4]) & z = x + y.
x = 2,
y = 4,
z = 6 ? ;
```

The same constraint properties can be extended to Z.

An extension of these properties is the case where x_1 can take many values. The different values contribute to different answer substitutions. Examples are subset and membership. We call this:

Constraint Property 3

$$\mathcal{P}_{LP}[\![x_1 \mathcal{I} x_2]\!]\rho_{LP} = \mathcal{P}_{LP}[\![x_1 \mathcal{I} x_2]\!]\rho'_{LP} = \mathit{true}$$
$$\mathcal{P}_{LP}[\![P \wedge (x_1 \mathcal{I} x_2)]\!]\rho_{LP} = \mathcal{P}_{LP}[\![(x_1 \mathcal{I} x_2) \wedge P]\!]\rho_{LP} = \mathcal{P}_{LP}[\![P]\!]\rho'_{LP}$$

where

$$\rho'_{LP} = \rho_{LP} \oplus \{x_1 \mapsto a_1\} \vee \rho'_{LP} = \rho_{LP} \oplus \{x_1 \mapsto a_2\} \vee \ldots$$
$$\vee\ \rho'_{LP} = \rho_{LP} \oplus \{x_1 \mapsto a_n\}$$

where $\rho'_{LP} \in \mathit{Env}_{LP}$.

The proofs for all the infix predicates are in Appendix D. Where there is only one way the environment can be enhanced, then we can consider **AR1**. However where there is more than one way of enhancing the environment, the values contribute to a set expression. In that case the comparison between the Z and LP domains will be deferred to Section 5.6.5.

Thus assuming that the execution terminates, and x_1, x_2 take unique values, we can summarise, thus. There are three cases for x_1, x_2, depending on whether or not x_1, x_2 are defined prior to execution of equality function and in each case **AR1** is true.

Equality: If 'f' is the syntactic predicate $=$ for variable (x_1, x_2): **AR1** holds for $(x_1 = x_2)$;

Subset If 'f' is the syntactic predicate $\subseteq$ for variable (x_1, x_2): **AR1** holds for $(x_1 \subseteq x_2)$;

Membership **AR1** is true where 'f' is the syntactic predicate $\in$ for variable (x_1, x_2); the LP interpretation of $\in$ underestimates the Z interpretation as required.

5.6.5 Set Comprehension and Variable Declarations

Set Comprehension is defined in terms of declarations $D_1;\ \ldots;\ D_n$, a constraining predicate p and an expression t involving the declared variables:

$$x_1 : \tau_1;\ x_2 : \tau_2;\ \ldots x_n : \tau_n \mid p \bullet t.$$

We first present the interpretation of declarations singly, then within the context of a set declaration. An example of how this works for the simple schema *FileSys* is given later, in Section 5.6.7.2.

5.6.5.1 Variable Declarations

Variable declarations occur within bound expressions with structure: $\_\ d \mid p \bullet t\ \_$ where d is a declaration, p is a predicate and t a term. d is of the form:

$$x_1 : \tau_1;\ x_2 : \tau_2;\ \ldots x_n : \tau_n.$$

These include *set comprehensions, quantified expressions, lambda expressions* and *schemas*. The declaration results in a single tuple of values $(x_1, \ldots x_n)$ being generated (or tested in the case of schemas). Each value is constrained by p and used to evaluate t.

An evaluation function $\mathcal{D}_{LP}$ gives the interpretation in D_{LP} of syntactic declarations $x : \tau$, where x is a variable and τ is set-valued with value provided by ρ_{LP}. The evaluation function is built recursively and interprets in a similar manner to the infix predicates of Section 5.6.4.1, for variable values generated by the declarations will update the environment. Since this is so, the declarations are treated as predicates. The declarations considered in this section do not include schema references, for these are treated separately.

1. τ is a *set:*

   $$\mathcal{D}_{LP}[\![x : \tau]\!]\rho_{LP} = \mathcal{P}_{LP}[\![x \in \tau]\!]\rho_{LP}.$$

   '$x : \tau$' has the effect of either testing a value or updating the environment as in the case of the membership predicate;

2. τ is a *Power Set*, $\tau = \mathbb{P}\,\tau'$ say:

   $$\mathcal{D}_{LP}[\![x : \mathbb{P}\,\tau']\!]\rho_{LP} = \mathcal{P}_{LP}[\![x \subseteq \tau']\!]\rho_{LP}.$$

   '$\tau : \mathbb{P}\,\tau'$ ' uses a 'subset' test rather than a 'membership of power set' test for reasons of efficiency. It has the same effect on the environment as the subset predicate;

3. τ is a *Cartesian Product*, $\tau_1 \times \tau_2$:

$$\mathcal{D}_{LP}[\![x : \tau_1 \times \tau_2]\!]\rho_{LP} = \mathcal{P}_{LP}[\![x = (x_1 \mapsto x_2)]\!]\rho_{LP}$$
$$\& \ \mathcal{P}_{LP}[\![x_1 \in \tau_1]\!]\rho_{LP}) \ \& \ \mathcal{P}_{LP}[\![x_2 \in \tau_2]\!]\rho_{LP}).$$

'T2' captures a representation of ordered pair (as an example of a tuple) in the LP. In our Gödel library this is 'OrdPair'. The following shows the implementation of $\nrightarrow$, which illustrates the interpretation of cartesian product.

```
PF(pf, s1, s2) <- ALL [z,x,y] (z In pf &
                                  (z = OrdPair(x,y))
                       ->  (x In s1) & (y In s2)  &
                       ALL [u] (OrdPair(x, u) In pf -> u = y)).
```

Thus a single declaration (such as $x : \tau$) has the effect of enhancing the environment as for the membership predicate:

$$\mathcal{D}_{LP}[\![x : \tau]\!]\rho_{LP} = \mathcal{P}_{LP}[\![x \in \tau]\!]\rho_{LP} = \mathcal{P}_{LP}[\![x \in \tau]\!]\rho'_{LP}$$

where if $\tau = \{a_1 \dots a_n\}$ then

$$\rho'_{LP} = \rho_{LP} \oplus \{x \mapsto a_1\} \vee \rho'_{LP} = \rho_{LP} \oplus \{x \mapsto a_2\}, \dots \vee \rho'_{LP} = \rho_{LP} \oplus \{x \mapsto a_n\}.$$

In general, if $x : \tau$ is a declaration, then

$$\mathcal{D}_{LP}[\![x : \tau]\!]\rho_{LP} = \mathcal{P}_{LP}[\![x \in \tau]\!]\rho'_{LP}$$

where it is possible for ρ'_{LP} to take many values determined by the nature of the type τ.

A sequence of declarations is evaluated in the LP as a conjunction:

$$\mathcal{D}_{LP}[\![D_1; \ \dots; \ D_n]\!]\rho_{LP} = \mathcal{D}_{LP}[\![D_1]\!]\rho_{LP} \ \& \ \dots \ \& \ \mathcal{D}_{LP}[\![D_n]\!]\rho_{LP}.$$

Declarations can be represented in a simpler manner in Z, where again values are chosen from some set-valued τ. However in this case τ is a type constructor thus

$$\mathcal{D}_Z[\![x : \tau]\!]\rho_Z = \mathcal{P}_Z[\![x \in \tau]\!]\rho'_Z$$

where τ is a set, or a power set, $\mathbb{P}\,\tau'$, or a cartesian product $\tau' \times \tau''$ and ρ'_Z takes its values from τ.

5.6.5.2 Interpretation of Set Comprehension

A set comprehension is

$$\{x_1 : \tau_1;\ x_2 : \tau_2;\ \ldots x_n : \tau_n \mid p \bullet t\}$$

where each $x_i : \tau_i$ provides a value which contributes to the tuple $(x_1, \ldots x_n)$ which is used to evaluate t. Thus if $s = \{d \mid p \bullet t\}$ is a syntactical set comprehension it is interpreted in D_{LP} as $\mathcal{E}_{LP}[\![s]\!]\rho_{LP}$ and in D_Z as $\mathcal{E}_Z[\![s]\!]\rho_Z$. Each of these interpretations is respectively dependent on its constituent $\mathcal{D}_{LP}, \mathcal{D}_Z$ where the declarations act as generators for s. Since declarations in the LP are treated as predicates, then the set comprehension of s is interpreted in the LP:

$$\mathcal{E}_{LP}[\![s]\!]\rho'_{LP} = \{\mathcal{D}_{LP}[\![d]\!]\rho'_{LP}\ \ \&\ \ \mathcal{P}_{LP}[\![p]\!]\rho'_{LP} \bullet \mathcal{E}_{LP}[\![t]\!]\rho'_{LP}\}.$$

(This way of writing a set comprehension in the LP is chosen so that it resembles set comprehension in Z. It differs from the way it would be coded in Gödel `s = {x : p(x)}`.) The environment ρ'_{LP} inside the comprehension is the variable which acts as a set generator, for recall that $\mathcal{D}_{LP}[\![x : \tau]\!]\rho_{LP} = \mathcal{P}_{LP}[\![x \in \tau]\!]\rho'_{LP}$. A similar interpretation is true for D_Z.

Recall from Section 5.5.3.1 that there are two kinds of sets, *set terms* and *sets of answer substitutions*. However for terminating executions they are treated the same. We assert that for terminating computations, **AR1** is true, since the interpretation of set comprehension is exact. In order to establish this correctness result, the approach is to initiate an induction process over the set generators (as in [17]). Thus a set with *one* generator will be expressed in terms of a set with *no* set generators, and we first need to define this object. A set with *no* set generators, can be defined in such a way that it evaluates to a singleton when predicate p evaluates to *true* and to an empty set when the predicate evaluates to false. In the LP domain:

$$\mathcal{E}_{LP}[\![\{\mid p \bullet t\}]\!]\rho'_{LP} = \{\mathcal{E}_{LP}[\![t]\!]\rho'_{LP}\} \text{ where } \mathcal{P}_{LP}[\![p]\!]\rho'_{LP} = \mathit{true}$$
$$\mathcal{E}_{LP}[\![\{\mid p \bullet t\}]\!]\rho'_{LP} = \{\} \text{ where } \mathcal{P}_{LP}[\![p]\!]\rho'_{LP} = \mathit{false}.$$

In the Z domain

$$\mathcal{E}_{Z}[\![\{\mid p \bullet t\}]\!]\rho'_{Z} = \{\mathcal{E}_{Z}[\![t]\!]\rho'_{Z}\}, \text{ where } \mathcal{P}_{Z}[\![p]\!]\rho'_{Z} = \mathit{tt}$$
$$\mathcal{E}_{Z}[\![\{\mid p \bullet t\}]\!]\rho'_{Z} = \{\} \text{ where } \mathcal{P}_{Z}[\![p]\!]\rho'_{Z} = \mathit{ff}.$$

Induction is based on the equivalence:

$$\{x_1 : \tau_1;\ x_2 : \tau_2;\ \ldots x_n : \tau_n \mid p \bullet t\} = \bigcup\{x_1 : \tau_1 \bullet \{x_2 : \tau_2;\ \ldots x_n : \tau_n \mid p \bullet t\}\}.$$

An example of a set with *one* generator illustrates the equivalence:

$$\{x : \tau \mid 1 < x < 4 \bullet x^3\}$$

where $\tau = \{1, 2, 3\}$. If x is respectively $1, 2, 3$, then $p = 1 < x < 4$ is $false, true, true$ and $\{\mid p \bullet t\}$ is $\{\}, \{8\}, \{27\}$. We have

$$\{x : \tau \mid 1 < x < 4 \bullet x^3\} = \{8, 27\} = \{\} \cup \{8\} \cup \{27\}$$

or

$$\{x : \tau \mid 1 < x < 4 \bullet x^3\} = \bigcup\{x : \tau \bullet \{\mid 1 < x < 4 \bullet x^3\}\}.$$

We first consider terminating computations where the interpretation is proposed as exact. The induction process depends on showing that if we assume that **AR1** holds for

$$\{x_1 : \tau_1;\ x_2 : \tau_2;\ \ldots x_n : \tau_n \mid p \bullet t\}$$

then it holds for

$$\{x_1 : \tau_1;\ x_2 : \tau_2;\ \ldots x_n : \tau_n;\ x_{n+1} : \tau_{n+1} \mid p \bullet t\}.$$

For the induction process we first consider the base case:

Base Case: no set generators

We consider **AR1** for the base case where there are no set generators. **Conditions 1–2** are true for standard sets since the interpretation is built recursively in both the LP and Z domains:

$$\begin{aligned}
&\mathcal{E}_{LP}[\![\{\mid p \bullet t\}]\!]\rho'_{LP} = \{\mathcal{E}_{LP}[\![t]\!]\rho'_{LP}\} \text{ where } P_{LP}[\![p]\!]\rho'_{LP} = true \\
&\mathcal{E}_{LP}[\![\{\mid p \bullet t\}]\!]\rho'_{LP} = \{\} \text{ where } \mathcal{P}_{LP}[\![p]\!]\rho'_{LP} = false \\
&\mathcal{E}_{Z}[\![\{\mid p \bullet t\}]\!]\rho'_{Z} = \{\mathcal{E}_{Z}[\![t]\!]\rho'_{Z}\}, \text{ where } \mathcal{P}_{Z}[\![p]\!]\rho'_{Z} = tt \\
&\mathcal{E}_{Z}[\![\{\mid p \bullet t\}]\!]\rho'_{Z} = \{\} \text{ where } \mathcal{P}_{Z}[\![p]\!]\rho'_{Z} = ff.
\end{aligned}$$

Assuming that the calculation of p, t for environment ρ'_{LP} terminates, the approximation of '$\{\mid p \bullet t\} = f(p, t)$' in the LP domain is exact, since **Condition 3** becomes:

$$\gamma(f_{LP}(\mathcal{E}_{LP}[\![(p,t)]\!]\rho'_{LP})) = f_Z(\gamma(\mathcal{E}_{LP}[\![(p,t)]\!]\rho'_{LP})).$$

If the calculation of p, t fails to terminate, then the left hand side of the approximation evaluates to $\gamma(\perp \circ \perp) = \varnothing_{\cup\perp}$ and thus underestimates the right hand side, however it is evaluated.

$$\begin{aligned} \text{LHS} &= \gamma(\mathcal{E}_{LP}[\![f(p,t)]\!]\rho_{LP}) = \gamma(\perp \circ \perp) = \varnothing_{\cup\perp} \\ \text{RHS} &= \mathcal{E}_Z[\![f(p,t)]\!](\gamma \circ \rho_{LP}) \end{aligned}$$

Set Comprehension – Induction on Declaration Sequence

Induction is based on the equivalence:

$$\{x_1 : \tau_1;\ x_2 : \tau_2;\ \ldots x_n : \tau_n \mid p \bullet t\} = \bigcup\{x_1 : \tau_1 \bullet \{x_2 : \tau_2;\ \ldots x_n : \tau_n \mid p \bullet t\}\}$$

for values of $\tau_1, \ldots \tau_n$ in the environment. Write the interpretation of $\bigcup$ in Z domain and LP domains as $\bigcup_Z$ and $\bigcup_{LP}$ as in Section 5.6.3.

The equivalence means that the set comprehension with one generator, '$\{x : \tau \mid p \bullet t\}$', can be evaluated in the LP environment:

$$\mathcal{E}_{LP}[\![\{x : \tau \mid p \bullet t\}]\!]\rho_{LP} = \bigcup\nolimits_{LP} \mathcal{E}_{LP}[\![\{\mid p \bullet t\}]\!]\rho'_{LP}$$

where τ is $\{a_1 \ldots a_n\}$ in ρ_{LP} and $\rho'_{LP} = \rho_{LP} \oplus \{x \mapsto a_i\}$. The interpretation is similar for D_Z.

For n generators, $x_1 : \tau_1;\ x_2 : \tau_2;\ \ldots x_n : \tau_n$, if $\tau_1 \mapsto s \in \rho_{LP}$, then $\tau_1 \mapsto \gamma(s) \in \rho_Z$ and the set comprehensions in both the Z and LP domains can be represented as the distributed union of a family of sets indexed by i where $a_i \in s$, $b_i \in \gamma(s)$ respectively:

$$\begin{aligned} &\mathcal{E}_{LP}[\![\{x_1 : \tau_1;\ \ldots x_n : \tau_n \mid p \bullet t\}]\!]\rho_{LP} = \\ &\qquad \bigcup\nolimits_{LP}\{a_i \in s \bullet \mathcal{E}_{LP}[\![\{x_2 : \tau_2;\ \ldots x_n : \tau_n \mid p \bullet t\}]\!](\rho_{LP} \oplus \{x_1 \mapsto a_i\})\} \\ &\mathcal{E}_Z[\![\{x_1 : \tau_1;\ x_2 : \tau_2;\ \ldots x_n : \tau_n \mid p \bullet t\}]\!]\rho_Z = \\ &\qquad \bigcup\nolimits_Z\{b_i \in \gamma(s) \bullet \mathcal{E}_Z[\![\{x_2 : \tau_2;\ \ldots x_n : \tau_n \mid p \bullet t\}]\!](\rho_Z \oplus \{x_1 \mapsto b_i\})\}. \end{aligned}$$

For finite sets, the approximation of **Condition 3** is exact. Thus since **AR1** holds for the empty sequence and assuming it holds for the sequence

$$\{x_2 : \tau_2;\ \ldots x_n : \tau_n \mid p \bullet t\}$$

in environments

$$\rho_Z \oplus \{x_1 \mapsto b_i\},\ \rho_{LP} \oplus \{x_1 \mapsto a_i\}$$

it then holds for

$$\{x_1 : \tau_1;\ x_2 : \tau_2;\ \ldots x_n : \tau_n \mid p \bullet t\}$$

in environments ρ_Z, ρ_{LP}. Thus **AR1** holds for $\{x_1 : \tau_1;\ x_2 : \tau_2;\ \ldots x_n : \tau_n \mid p \bullet t\}$ since it holds for each of its components τ_i, p, t.

For infinite sets, or if any set is non-standard in the LP, the induction process depends on whether we are addressing set terms or sets of answer substitutions:

- For set terms the LP interpretation of s evaluates to $\perp \circ \perp$ and underestimates the Z interpretation in the same manner as the 'no set generator case';
- For the 'answer set' the result depends on distributed union, where incomplete sets are involved. This is because we can equivalently express a set comprehension as a distributed union. Since distributed union underestimates for incomplete sets, then set comprehension underestimates for incomplete sets.

Thus set comprehension in the LP is an exact interpretation for finite or complete sets. For infinite or incomplete sets the LP interpretation is an underestimation. This is true for either set terms or sets of answer substitutions.

5.6.5.3 Set Operations Power Set, Set Intersection

Other set operations $\epsilon = f(x_1, x_2, \ldots, x_n)$ can be expressed via set comprehensions. Examples are set intersection and power set:

Set Intersection $s = x_1 \cap x_2$ in the LP is part of the library of set operations. However $\cap$ can be expressed as

```
s = {x : (x In x1) & (x In x2 )}.
```

where we are assuming that x_1, x_2 are appropriately typed. In Z this last condition is expressed explicitly so

$$s = x_1 \cap x_2 = \{x : X \mid (x \in x_1 \wedge x \in x_2) \bullet x\}.$$

This is treated as a set comprehension where $p = x \in x_1 \wedge x \in x_2$. Thus for terminating computations, the interpretation in the LP approximates exactly, and for non-terminating computations, the LP interpretation underestimates;

Power Set in the LP, $s = \mathbb{P}\, x$ can be be expressed in Gödel as

```
s = {z : z Subset x }.
```

Its generic 'LP form' is as a set comprehension with predicate *true*:

$$s = \mathcal{E}_{LP}[\![\{z \subseteq x \mid true \bullet z\}]\!]\rho_{LP}.$$

Since 'power set' is a type in Z, there is no specific definition for it (see Chapter 3). The power set axiom of ZF (from Appendix A) provides a definition in Z, for $s = \mathbb{P}\, x$ is such that

$$\forall\, z \bullet (z \in s \Leftrightarrow z \subseteq x)$$

and this set and the interpretation in the LP can be shown to be equal. Thus the interpretation of power set is exact for finite sets. For infinite sets, the LP interpretation is $\perp \circ \perp$ which always underestimates.

The interpretation is exact if the computations terminate. For infinite or incomplete sets, the interpretation in the LP evaluates to $\perp \circ \perp$ and so underestimates the interpretation in Z.

5.6.5.4 Quantifiers

Universal Quantification

The syntactic predicate '$\forall\, x : s \mid p \bullet q$' is interpreted in the LP :

```
ALL [x] (x In s <-> p => q )
```

and in Z as

$$\forall\, x : s \mid p \Rightarrow q$$

and is evaluated for finite sets s on an element by element basis for values of s. Its interpretation can be denoted in the LP as $\mathcal{P}_{LP}[\![fx]\!]\rho_{LP}$, where x is the tuple s, p, q

and 'f' the syntactic '$\forall$'. For terminating computations, **Condition 1–2** hold in the LP and in Z. If '$\forall\, x : s \mid p \Rightarrow q$' is *true* then **Condition 3** becomes:

$$\begin{aligned} \textbf{LHS} = &\ \gamma(f_{LP}(\mathcal{P}_{LP}[\![(s,p,q)]\!]\rho_{LP})) \\ = &\ \gamma(true) = tt \\ \textbf{RHS} = &\ f_Z(\gamma(\mathcal{P}_{LP}[\![(s,p,q)]\!]\rho_{LP})) = tt. \end{aligned}$$

and is thus exact for each p, q in an environment containing s. The result follows similarly if '$\forall\, x : s \mid p \bullet q$' is *false*.

For infinite sets the truth value in the LP will be $\perp$ i.e. it will fail to terminate and the left hand side of Condition 3 will evaluate to $\perp$. For infinite sets, the Z interpretation of the quantification will result in the value tt or $f\!f$, and $\gamma(\perp) = \perp$ and $\perp \sqsubseteq f\!f, \perp \sqsubseteq tt$. For cases where s is incomplete, or not fully defined, then the LP interpretation results in $\perp$, which either underestimates the interpretation in Z, if it is $f\!f, tt$, or is exact, if the interpretation in Z is $\perp$. Thus in all cases, the interpretation in the LP of universal quantification adheres to **AR1**.

Existential Quantification

A similar interpretation is true for $\exists$ where '$\exists\, x : s \mid p \bullet q$' is interpreted in the LP by

```
SOME [x] (x In s <-> p & q ).
```

and in Z: '$\exists\, x : s \mid p \wedge q$'. The LP interpretation evaluates to $true, false, \perp$, which always underestimates its interpretation in Z, as for $\forall$.

5.6.6 Function Application and Lambda Expressions

Function application of t_1 to t_2 assumes that t_1 is appropriately typed, as a set of pairs. It is interpreted in the LP by

$$\mathcal{E}_{LP}[\![t_1 t_2]\!]\rho_{LP} = a \Leftrightarrow t_2 \mapsto a \in t_1.$$

It is mapped in a similar way in Z. For terminating computations, where set t_1 is finite, the interpretation is exact. Where t_1 is infinite or incomplete, the LP underestimates the Z interpretation for

$$\mathcal{E}_{LP}[\![t_1 t_2]\!]\rho_{LP} = \perp.$$

Lambda expressions require evaluation individually:

$\lambda\, x_1 : \tau_1;\ \ldots x_n : \tau_n \mid p \bullet t$ where t is a term can be expressed (in Z) as a set of maplets $x \mapsto a$ where the x is a tuple $(x_1, \ldots, x_n)$ and a is the term t evaluated at $(x_1, \ldots, x_n)$:

$$\{x_1 : \tau_1;\ \ldots x_n : \tau_n \mid p \bullet (x_1, \ldots, x_n) \mapsto t)\}.$$

It is interpreted as the equivalent set expression in the LP:

$$\mathcal{E}_{LP}[\![\lambda\, x_1 : \tau_1;\ \ldots x_n : \tau_n \mid p \bullet t]\!]\rho_{LP}$$

which is

$$\begin{aligned}&\mathcal{E}_{LP}[\![\{x_1 : \tau_1;\ \ldots x_n : \tau_n \mid p \bullet (x_1, \ldots, x_n) \mapsto t)\}]\!]\rho_{LP} = \\ &\quad \{\mathcal{D}_{LP}[\![x_1 : \tau_1;\ \ldots x_n : \tau_n]\!]\rho_{LP}\ \&\ \mathcal{P}[\![p]\!] \\ &\qquad \mid T2(Tn(x_1, \ldots, x_n) \mapsto t)\}.\end{aligned}$$

An example can be seen in Appendix C. The approximation is exact for terminating computations and underestimates for the rest.

5.6.7 Interpretation of Schemas and Schema Expressions

Suppose that the syntactic objects *schema*, *axdef* $\in \Sigma_3$ are interpreted in the LP and in Z by $\mathcal{S}_{LP}$, $\mathcal{S}_Z$ respectively. A schema can be represented (in its horizontal form) by the following syntactic object:

$$Sch \mathrel{\widehat{=}} \left[\, D_1;\ \ldots;\ D_n \;\middle|\; CP \,\right]$$

where $D_i = X_i : \tau_i$, and $CP\ ::=\ CP_1 \wedge \ldots \wedge CP_m$.

Sch evaluates to a set expression, of bindings of variable name(s) to values. The bindings are constrained by the variable declarations and by the schema predicate. Constants are used to capture variable names in the schema declaration and these are linked to local variable names via the binding. Suppose *GCP* is defined as *CP* where all the free occurrences of $X_1 \ldots X_n$ are replaced by $x_1 \ldots x_n$

$$GCP(x_1, \ldots, x_n) = CP(X_1/x_1, \ldots, X_n/x_n)$$

and any bound variables replaced by arbitrary local variables. *A set of schema bindings* of *Sch* can be represented in Z (as suggested in [17]) by a set expression:

$$\{x_1 : \tau_1;\ \ldots;\ x_n : \tau_n \;\middle|\; GCP \bullet \{X_1 \mapsto x_1, \ldots, X_n \mapsto x_n\}\}.$$

There is a similar representation in the LP where $[Bind_1(X_1, x_1), \ldots, Bind_n(X_n, x_n)]$ replaces $\{X_1 \mapsto x_1, \ldots, X_n \mapsto x_n\}$. If we assume that the set of bindings is constrained by an initial imposed environment ρ^o then the interpretation of the schema $Sch \mathrel{\widehat=} [\, D \mid CP \,]$ is the interpretation of a set expression:

$$\begin{aligned}&\mathcal{S}_{LP}[\![[\, X_1 : \tau_1;\ \ldots;\ X_n : \tau_n \mid CP \,]]\!]\rho^o_{LP}\\ &= \mathcal{E}_{LP}[\![\{x_1 : \tau_1;\ \ldots;\ x_n : \tau_n \mid GCP \bullet [Bind_1(X_1, x_1), \ldots, Bind_n(X_n, x_n)]\}]\!]\rho^o_{LP},\end{aligned}$$

where ρ^o_{LP} can contain defined values of all the schema variables (as in the case of the assembler in Chapter 4) or just some of them (as in the case of the Unix file system in Chapter 4). The interpretation of schemas and schema expressions is in terms of a *characteristic predicate*, providing a single binding for a schema expression.

5.6.7.1 Characteristic Predicate for a Schema Expression

A schema binding is obtained by providing the schema with some initial environment, ρ^0_{LP}. After the schema has been interpreted and if the computation is successful, this environment is updated to ρ_{LP}, where ρ_{LP} is such that all schema variables $x_1, \ldots x_n$ have defined values ($\neq \bot$). In its initial state a schema is interpreted by:

$$\mathcal{S}_{LP}[\![[\, X_1 : \tau_1;\ \ldots;\ X_n : \tau_n \mid CP \,]]\!]\rho^o_{LP}$$

and this evaluates in the LP to bindings of variable names to values. During the execution the environment has been enhanced by the provision of additional variable values other than ‘undefined’. The values have been provided by the interpretation of schema declarations and predicate. This binding is a member of the set defined previously:

$$\begin{aligned}&\mathcal{S}_{LP}[\![[\, X_1 : \tau_1;\ \ldots;\ X_n : \tau_n \mid CP \,]]\!]\rho^o_{LP}\\ &= \mathcal{E}_{LP}[\![\{x_1 : \tau_1;\ \ldots;\ x_n : \tau_n \mid GCP\\ &\qquad \bullet [Bind_1(X_1, x_1), \ldots, Bind_n(X_n, x_n)]\}]\!]\rho^o_{LP}\end{aligned}$$

where each enhanced environment $\rho_{LP} \in Env_{LP}$ satisfies

$$\begin{aligned}&\mathcal{D}_{LP}[\![D_1;\ \ldots;\ D_n]\!]\rho^0_{LP}\ \&\ \mathcal{P}_{LP}[\![GCP]\!]\rho^0_{LP} =\\ &\qquad \mathcal{D}_{LP}[\![D_1;\ \ldots;\ D_n]\!]\rho_{LP}\ \&\ \mathcal{P}_{LP}[\![GCP]\!]\rho_{LP}.\end{aligned}$$

The characteristic schema predicate of Sch is as follows:

$$\begin{aligned}&\mathit{SchemaType}(\mathit{binding}, \mathit{Sch}) \Leftrightarrow\\ &\quad (\mathit{binding} = [\mathit{Bind}_1(X_1, x_1), \ldots, \mathit{Bind}_n(X_n, x_n)]) \ \&\\ &\qquad \mathcal{D}_{LP}[\![D_1;\ \ldots;\ D_n]\!]\rho_{LP} \ \&\ \mathcal{P}_{LP}[\![GCP]\!]\rho_{LP},\end{aligned}$$

where each x_i satisfies

$$\mathcal{D}_{LP}[\![D_1;\ \ldots;\ D_n]\!]\rho_{LP} \ \&\ \mathcal{P}_{LP}[\![GCP]\!]\rho_{LP}.$$

The values $x_1 \mapsto a_1 \ldots x_n \mapsto a_n$ which satisfy $\mathit{SchemaType}$ have either been generated or were part of the initial environment. In either case this has been achieved by means of the interpretation of $D_1;\ \ldots;\ D_n$ and $GCP(x_1, \ldots x_n)$. The generated values have been obtained via the application of **Constraint Properties 1 - 3** in Section 5.6.4.1. Note that although the schema definition in the LP uses 'if' ($\leftarrow$), by the CWA, this has the same effect as 'if an only if' ($\Leftrightarrow$).

The Z interpretation can similarly be represented by a set of bindings where

$$\mathit{binding} = \{X_1 \mapsto \gamma(x_1), \ldots, X_n \mapsto \gamma(x_n)\}.$$

The values $\gamma(x_i) \in \operatorname{ran} \rho_Z$ satisfy

$$\mathcal{D}_Z[\![D_1;\ \ldots;\ D_n]\!]\rho_Z \wedge \mathcal{P}_Z[\![GCP]\!]\rho_Z.$$

It is worth investigating how the above would apply to a schema, and the one chosen is *FileSys* from Chapter 4 Section 4.4.

5.6.7.2 Interpretation of *FileSys*

This can be written horizontally as:

$$\mathit{FileSys} \mathrel{\widehat{=}} \left[\, \mathit{Files} : \mathbb{F}\, \mathit{FileId};\ \mathit{Count} : 0 \mathrel{..} \mathit{MaxFiles} \;\middle|\; \#\mathit{Files} = \mathit{Count} \,\right].$$

Suppose that $\mathit{MaxFiles} = 10$ is a value provided by the animation user, and that *FileId* is instantiated as $\{F1, F2, F3\}$. Then

$$\begin{aligned}&\mathcal{D}_{LP}[\![\mathit{Files} : \mathbb{F}\, \mathit{FileId};\ \mathit{Count} : 0 \mathrel{..} \mathit{MaxFiles}]\!]\rho^o_{LP}\\ &= \mathcal{P}_{LP}[\![\mathit{files} \subseteq \{F1, F2, F3\}]\!]\rho^o_{LP} \ \&\ \mathcal{P}_{LP}[\![\mathit{count} \in \{1 \mathrel{..} 10\}]\!]\rho^o_{LP}.\end{aligned}$$

If we substitute these values, a binding for *FileSys* is given by:

$$\begin{array}{l} SchemaType(binding, FileSys) \Leftrightarrow \\ \quad (binding = [Bind_1(Files, files), Bind_2(Count, count]) \ \& \\ \quad \mathcal{P}_{LP}[\![files \subseteq \{F1, F2, F3\}]\!]\rho^o_{LP} \ \& \ \mathcal{P}_{LP}[\![count \in \{1\,..\,10\}]\!]\rho^o_{LP} \ \& \\ \quad \mathcal{P}_{LP}[\![\#files = count\ ]\!]\rho^o_{LP}. \end{array}$$

Suppose that initially, $\rho^o_{LP} = \{files \mapsto \perp, count \mapsto \perp\}$. During the execution, *files* is of type $\mathbb{F}$ so becomes evaluated through the interpretation of its declaration:

$$files \subseteq \{F1, F2, F3\}.$$

Similarly, *count* becomes evaluated through its declaration as a member of $1\,..\,10$. A binding of *FileSys* can be expressed:

$$\begin{array}{l} binding = [Bind_1(Files, files), Bind_2(Count, count] \ \& \\ \quad \mathcal{P}_{LP}[\![files \subseteq \{F1, F2, F3\}]\!]\rho^o_{LP} \ \& \ \mathcal{P}_{LP}[\![count \in \{1\,..\,10\}.]\!]\rho^o_{LP} \ \& \\ \qquad \mathcal{P}_{LP}[\![\#files = count\ ]\!]\rho^o_{LP}. \end{array}$$

Thus if *files* evaluates to $\{F1\}$ (say), then in order to satisfy the schema predicate *and* its declaration, *count*, evaluates to '1' since

$$\#files = count \ \& \ count \in \{1\,..\,10\}.$$

Substituting these values yields:

$$\begin{array}{l} binding = [Bind_1(Files, \{F1\}), Bind_2(Count, 1] \ \& \\ \quad \mathcal{P}_{LP}[\![\{F1\} \subseteq \{F1, F2, F3\}]\!]\rho_{LP} \ \& \ \mathcal{P}_{LP}[\![1 \in \{1\,..\,10\}]\!]\rho_{LP} \ \& \\ \qquad \mathcal{P}_{LP}[\![1 = 1]\!]\rho_{LP}, \end{array}$$

where $\rho_{LP} = \{files \mapsto \{F1\}, count \mapsto 1\}$ is the enhanced value of the environment and

$$binding = [Bind_1(Files, \{F1\}), Bind_2(Count, 1)]$$

which was one of the values actually obtained. The full set of bindings can be obtained from the full set of answer substitutions, as was indicated in Chapter 4.

For the 'complete' assembler, the variables were initially all ground, so that ρ^o_{LP} contains no undefined values and the environment is unaltered: $\rho_{LP} = \rho^o_{LP}$, where similarly $\rho_Z = \rho_Z^o$. For the Unix file system, and for the two-phase assembler, for each of the schemas considered some variables are ground in ρ_{LP}^0, and some are determined by the execution.

5.6.7.3 Approximation for Schemas

AR1 can now be considered for schemas and is worth restating. If ϵ is a syntactic Z expression for a set of schema bindings then condition **AR1** must hold for a correct animation of Z in D_{LP}:

Approximation Rule 1 (AR1)

$$\gamma(\mathcal{S}_{LP}[\![\epsilon]\!]\rho_{LP}) \sqsubseteq \mathcal{S}_Z[\![\epsilon]\!](\gamma \circ \rho_{LP}).$$

where

$$\epsilon = \{X_1 : \tau_1 \ldots X_n \tau_n \mid CP \bullet \{X_1 \mapsto x_1, \ldots X_n \mapsto x_n\}\}.$$

The structural induction rule states that if it can be shown that **AR1** holds for syntactic variable $\epsilon = x$, then it also holds for syntactical expression $\epsilon = fx$, where in this case, f is a syntactic operator which forms a schema from tuple $\epsilon = (D, CP)$, where D is a declaration and CP is a predicate. We denote by f_Z, f_{LP} the interpretation in the Z domain and LP domain respectively of the syntactic expression fx. Thus the left hand side of **AR1** is

$$\begin{aligned}
&\gamma(\mathcal{S}_{LP}[\![\,[\, X_1 : \tau_1;\ \ldots;\ X_n : \tau_n \mid CP\,]\,]\!]\rho_{LP}^o) \\
&= \gamma(\mathcal{E}_{LP}[\![\{x_1 : \tau_1;\ \ldots;\ x_n : \tau_n \mid GCP \bullet [Bind_1(X_1, x_1), \ldots, Bind_n(X_n, x_n)]\}]\!]\rho_{LP}^o).
\end{aligned}$$

The right hand side of **AR1** is

$$\begin{aligned}
&\mathcal{S}_Z[\![\,[\, X_1 : \tau_1;\ \ldots;\ X_n : \tau_n \mid CP\,]\,]\!]\rho_Z^o \\
&= \mathcal{E}_Z[\![\{x_1 : \tau_1;\ \ldots;\ x_n : \tau_n \mid GCP \bullet \{X_1 \mapsto x_1, \ldots, X_n \mapsto x_n\}\}]\!]\rho_Z^o.
\end{aligned}$$

These are *set comprehension*, which have been treated in Section 5.6.5. These interpret exactly where components are finite and complete (as in the case of *FileSys*).

An example of a schema with an incomplete set of bindings is

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{UnDef}} \\
\hline
X, Y : \mathbb{N} \\
\hline
Y \in \{1,2,3\} \\
((X = 1) \vee (X = 3) \vee (\{X, 1, 2, 3\} = \{1, 2, 3, 4\})) \\
\hline
\end{array}$$

The Gödel code and queries for *UnDef* is in Appendix C.1.2. The bindings of *UnDef* are interpreted in the LP

$$\{\{X \mapsto 1,\ Y \mapsto 1\}, \{X \mapsto 3,\ Y \mapsto 1\}\}_{\cup\perp}$$

for the program flounders and the set is incomplete. This is because of the inadequate constraint satisfaction for the Gödel implementation.

The set of bindings of *UnDef* is interpreted in Z:

$$\{\{X \mapsto 1,\ Y \mapsto 1\}, \{X \mapsto 3,\ Y \mapsto 1\}, \{X \mapsto 4,\ Y \mapsto 1\}, \{\{X \mapsto 1,\ Y \mapsto 2\}, \ldots\}$$

which demonstrates how the LP interpretation underestimates the Z. In general, where the answer sets is incomplete the LP interpretation underestimates.

5.6.7.4 Schema Conjunction and Disjunction

We now interpret syntactical objects such as $Sch \mathrel{\widehat{=}} Sch^1 \wedge Sch^2$ and $Sch \mathrel{\widehat{=}} Sch^1 \vee Sch^2$. Provided that Sch^1, Sch^2 have compatible declarations their conjunction and disjunction can be defined. These are modelled by conjunction and disjunction of the LP predicates of Sch^1, Sch^2 with lists of bindings appended. This does cause duplication but has not (so far) been found a practical problem. Suppose Sch^1 has predicate CP^1 and declaration sequence D^1 where

$$D^1 = X_1^1 : \tau_1^1;\ \ldots;\ X_n^1 : \tau_n^1$$

modelled by Gödel list b_1:

$$[\ Bind_1^1(X_1^1, x_1^1), \ldots, Bind_n^1(X_n^1, x_n^1)\ ]$$

and Sch^2 has a predicate CP^2 and compatible declaration sequence D^2 where

$$D^2 = X_1^2 : \tau_1^2;\ \ldots;\ X_n^2 : \tau_n^2$$

modelled by Gödel list b_2. Given that the characteristic predicates of Sch^1, Sch^2 are respectively

$$SchemaType(binding, Sch^1), SchemaType(binding, Sch^2)$$

then the characteristic predicate of Sch is

$$\begin{aligned}&SchemaType(binding, Sch) \Leftrightarrow (binding = b_1 \frown b_2)\ \&\\ &\qquad \mathcal{D}_{LP}[\![D^1;\ D^2]\!]\rho^o_{LP}\ \&\ \mathcal{P}_{LP}[\![GCP^1 \wedge GCP^2]\!]\rho^o_{LP}.\end{aligned}$$

We can show that this interprets exactly the Z interpretation where the program terminates, and provides an incomplete set of answer substitutions when the program fails to terminate.

First 'expand out' the version of $Sch \mathrel{\widehat=} Sch^1 \wedge Sch^2$ which is interpreted in Z as

$$\begin{aligned}&\mathcal{S}_Z[\![Sch^1 \wedge Sch^2]\!]\rho_Z\\ &\qquad = \mathcal{S}_Z[\![[\ D^1;\ D^2 \mid CP^1 \wedge CP^2\ ]]\!]\rho^0_Z.\end{aligned}$$

The above represents the declaration sequences before they are merged, so some repetitions would be expected. Schema Sch is then expressed as a set comprehension in the usual way:

$$\begin{aligned}&\mathcal{E}_Z[\![\{x^1_1 : \tau^1_1;\ \ldots;\ x^1_n : \tau^1_n;\quad x^2_1 : \tau^2_1;\ \ldots;\ x^2_n : \tau^2_n \mid GCP^1 \wedge GCP^2\\ &\qquad \bullet \{X^1_1 \mapsto x^1_1, \ldots, X^1_n \mapsto x^1_n;\quad X^2_1 \mapsto x^2_1, \ldots, X^2_n \mapsto x^2_n\}\}]\!]\rho^o_Z.\end{aligned}$$

Then $Sch^1 \wedge Sch^2$ in the LP is

$$\mathcal{S}_{LP}[\![Sch^1 \wedge Sch^2]\!]\rho_{LP}$$

where given that the characteristic predicates of Sch^1, Sch^2 are respectively

$$SchemaType(binding, Sch^1), SchemaType(binding, Sch^2)$$

then the characteristic predicate of Sch evaluates to

$$\begin{aligned}&SchemaType(binding, Sch) \Leftrightarrow\\ &\qquad (binding = [\ Bind^1_1(X^1_1, x^1_1), \ldots, Bind^1_n(X^1_n, x^1_n),\\ &\qquad Bind^2_1(X^2_1, x^2_1), \ldots, Bind^2_n(X^2_n, x^2_n)\ ])\ \&\\ &\qquad\qquad \mathcal{D}_{LP}[\![D^1;\ D^2]\!]\rho^o_{LP}\ \&\ \mathcal{P}_{LP}[\![GCP^1 \wedge GCP^2]\!]\rho^o_{LP}\end{aligned}$$

which is the same as if the expression had been expanded first. Since this is so, then the interpretations are exact or not according to the values of their components, as for *Sch* defined without conjunction. The criteria for exactness or underestimation for each of these interpretations has already been discussed. In general, where each component of an expression is exact, the whole expression is exact, but where one component underestimates, the whole underestimates.

Schema disjunction is defined in a similar manner. If $Sch \mathrel{\widehat=} Sch^1 \vee Sch^2$ then the bindings are appended and the LP interpretations of the schema predicates are disjoined. In Chapter 4, the convention for naming variables is further refined, so that priming, input, output becomes apparent. The formalism is not explored here. However the naming convention enables schema composition and piping to be accomplished. An outline is presented in [119].

5.6.7.5 Schema Reference in a Declaration

A declaration can contain a schema reference. If $x_i \in VAR, t, t_i \in expr, Sch \in NAME$, then recall from Section 5.3.1:

$$basic\_decl ::= x_1, \ldots, x_n : t \mid Sch$$

where *Sch* is a schema reference. The interpretation of this in Z is that its declarations are merged with the declarations of the schema which reference it and its predicate is conjoined. Thus if

$$Sch_1 \mathrel{\widehat=} \left[\, X_1 : \tau_1;\ \ldots X_n : \tau_n;\ Sch \;\middle|\; predicate\ \ of\ \ Sch_1 \right]$$

then this is equivalent to $Sch_1 \mathrel{\widehat=} Sch \wedge Sch_2$ where

$$Sch_2 \mathrel{\widehat=} \left[\, X_1 : \tau_1;\ \ldots X_n : \tau_n \;\middle|\; predicate\ \ of\ \ Sch_1 \right].$$

Thus if a schema *Sch* appears as a reference in the declarations of schema Sch_1, then this is treated as for schema conjunction above: *Sch* is removed from the declarations and conjoined to the predicate of *Sch* and its remaining declarations.

5.6.7.6 Binding Formation θ

The value of the binding formation θSch depends on its context. However we interpret it here in the same context as in the Unix file system case study. In that case study $\{Sch \bullet \theta Sch\}$ was constructed first and θSch was interpreted as a member of that set. The set $\{Sch \bullet \theta Sch\}$ in the LP is the 'same' set as $\mathcal{S}_{LP}[\![Sch]\!]\rho_{LP}$ however in this case it is a set *term* and not an answer set. It is the set comprehension S defined by

$$S = \{SchemaType(binding, Sch) \bullet binding\}$$

so that the binding formation $\theta Sch \in S$, where the code can be found in Appendix C.

This means that if the computation terminates its interpretation is exact, and if one of the members of s fails to terminate then the output of the whole computation is $\perp$.

5.6.7.7 Axiomatic and Generic Definitions

Axiomatic and Generic Definitions require individual definitions; they are interpreted in such a way that they are exact where the computations terminate.

Axiomatic Definitions are modelled in the same way as schemas, and suitable *names* must be generated for them `Axiom1, Axiom2`.... They must then be conjoined to the schema which refer to them, as in the assembly case study in Chapter 4. Their interpretation is the same as for schemas,

Generic definitions are treated in the same way as the parametrised definitions of partial function etc, ie by using parameters `a,b..` They are instantiated when the set is instantiated, and are defined by a predicate in the LP, as for Sequence in Appendix C.

5.7 Summary

This chapter has presented an extension of the work of Breuer and Bowen [17] in determining a formal framework for the correct animation of Z. In formalising the structure simulation rules of Chapter 4, we have completed **Contribution 4** of this thesis. The contents can be summarised:

1. It has provided a tutorial introduction to the formal framework and proof rules of abstract approximation. Abstract approximation is based on the procedures of *abstract interpretation*, formalised by the Cousots in [23]. A comparison has been made with the more established procedures of abstract approximation;

2. It formalises the animation rules of Chapter 4 by applying the correctness criteria of abstract approximation to the rules. Since these rules are practical, it is now possible to use *correct* rules which have been effective for two case studies;

3. In order to fit the logic program semantics into the correctness criteria it frames the resolution inference rules in a novel way and models the 'incomplete set structure' using the formalisms of Finite Sets in Appendix B;

4. The rules from Chapter 4 are shown to adhere to the correctness criteria. Since these rules *do* include the possibility of non-integer (given) sets and include important features of Z, the results have a potential application to 'real world' examples such as the assembler and Unix file system. Although the possibility that the program might not terminate was explored in great detail, this very rarely happened and the examples had to be contrived. The existence of proven rules for animation provide the user with confidence that the results of the animation represent Z.

Chapter 6

Summary and Further Work

6.1 Summary

Delivered computer systems are at worst, unusable by their purchasers, and at best often do not satisfy requirements of functionality and/or safety. The use of formal methods in the development of computer based systems is seen as addressing these problems. When a specification is written in a formal mathematical notation, formal reasoning can be used to ensure that the specification has desirable functionality but with no undesired side-effects. However it is often difficult for the specifier to communicate the functionality of the specification to the user of the implemented system and a proposed method is to *animate* the specification. It is thought that a non-executable specification is best for the specifier since it allows the concise expression of *what* the system should do with no unwanted details as to *how* it should be implemented. Thus for animation, it is necessary that a tool be used to translate the specification to some executable form and this thesis addresses the lack of usable tools for the animation of the Z notation. Contributions are as follows:

Contribution 1 is presented in Chapter 2 which contains a survey of current tools and weaknesses in these tools which have been identified. These are, principally, that the current tools are not usable (in terms of the ability to be used on 'real' examples) or that their translation rules are not correct, or both. The advisability of using a declarative language (such as Haskell or Prolog) is then argued, and in particular the use of a logic programming language is recommended. This is because of its desirable property of being used for 'what if' queries, where the program can be used to test possible differences between real and expected test results. The resulting program is then amenable to analysis via meta-interpreters and techniques of inductive logic as in [79];

Contribution 2 is presented in Chapter 3, in which a formal link between finite set theory and ZF is investigated. A 'partially correct' program is one which is derived logically from a specification, and this way of obtaining a correct specification is presented in Chapter 3. Since there were certain shortcomings in this approach, the work took a different direction and this is presented in Chapter 4;

Contribution 3 In Chapter 4 previous work by ourselves and other researchers in the animation of Z by Prolog using structure simulation is presented first [119]. However, it was noted that while the animation was successful in some respects, several disadvantages were discovered. We argue next that a better technique is to use the Gödel language as described in [117]. Gödel has support for sets, and a program in Gödel is defined as a theory in first order logic and an implementation must be sound with respect to this semantics. Two case studies were presented in Chapter 4, the assembler case study (originally animated in Prolog) and the Unix file system. A test case strategy for animation was presented, of systematic functional and structural testing similar to that for testing of software. Animation can be used to make visible the structure of the specification and, in certain cases, show that a property is *not* present. This makes animation complementary to proof, for formal proof can be expensive and time consuming. Animation tests specific cases and can be made understandable to a customer or a user of the implemented system. This means that potential misunderstandings as to the functionality of the implemented system can be avoided. However testing can seldom be exhaustive so that proof provides for the general case.

The animator proved practical and the method sophisticated is enough to handle the animation of the non-trivial specifications such as the assembler and the Unix file system;

Contribution 4 is the formalisation of structure simulation and Chapter 5 of this thesis utilised more recent criteria for correctness for animations of Z first proposed by [17], viz. *abstract approximation.* The latter paper presented a functional language (Miranda) for implementation, wherein sets are represented as lazy lists. Breuer and Bowen suggest that a 'correct' interpretation of Z syntax in the executable language will approximate from below its interpretation via ZF. Thus for example where a set is implemented in a manner which exceeds memory bounds, non-termination means that the output ($\perp$) is an under-interpretation thus the answer is 'correct'. Chapter 5 contains a tutorial presentation of these proof criteria and they are applied to the rules for animation presented in Chapter 4 for the animation of the assembler and the Unix file system were proved correct, and in order to do so the logic program semantics and resolution rules were framed in a novel way. The formal arguments are presented in a tutorial fashion, with illustrative examples throughout.

6.2 Further Work

Further work includes

1. The extension of the investigation to include more of the Z notation and the completion of proofs;

2. The development of meta-interpreters and techniques of inductive logic to trace and correct flaws in the specification discovered when animation returns an unexpected result, as in [79];

3. Automation of the rules: in order to provide an effective validation technique, the rules require automation with a suitable interface for user instantiation and input of queries;

4. Further investigation into a strategy for selecting test cases for animation, including (where possible) the automatic generation of test cases;

5. The investigation of a functional logic programming language for animation: an interesting area of work would be the investigation of a functional logic language for animation purposes as suggested in [124]. A functional logic language can take advantage of both the logic and functional paradigms. A first step would be to look at the use in practice, by applying the suggested rules to a substantial case study, as in the case of Gödel;

6. An 'embedding' of Z in HOL is described in [16], where the theorem prover HOL is used to support proof in Z. A suggestion is to embed Gödel in HOL so that reasoning can be applied to link the two formalisms.

6.3 Impact of the Work

The impact of the work is that a set of rules for the animation of the Z notation has been presented. The potential of the rules and the animating language, Gödel for contributing to an effective tool have been demonstrated (Chapter 4). Furthermore, the rules have been proved correct using correctness criteria developed by others and techniques developed by ourselves (Chapter 5). The IMPRESS tools has been cited, as having successfully supported validation of a large system using a logic programming language. These tools have potential implications on the further development of our own tool, in that techniques of inductive logic and machine learning could be used to make the animation tool more effective, by facilitating the search, and correction of, flaws in the Z specification being animated.

Bibliography

[1] DEFSTAN 00-55. The Procurement of Safety Critical Software in Defence Equipment. Technical Report Interim Defence Standard 00-55, Ministry of Defence, UK, 1991.

[2] Draft IEC 1508. *Functional Safety – Safety-Related Systems (7 parts).* IEC Draft Standard. IEC SC65A (Secretariat) 123, June 1995.

[3] A. E. Abdallah, A. Barros, J. B. Barros, and J. P. Bowen. Deriving correct prototypes from formal Z specifications. Technical Report SBU-CISM-00-27, SCISM, South Bank University, London, UK, October 2000.

[4] S. Abramsky and C. Hankin. *Abstract Interpretation of Declarative Languages.* Ellis Horwood Limited, Chichester, West Sussex, PO19 1EB, England, 1987.

[5] J. R. Abrial. *The B Book: Assigning Programs to Meanings.* Cambridge University Press, 1996.

[6] J. H. Andrews. Proof-Theoretic Characterisations of Logic Programming. In *Proceedings of the 14th International Symposium on the Mathematical Foundations of Computer Science, vol. 379 of LNCS*, pages 145–154. Springer, 1990.

[7] K.R. Apt and R.N. Bol. Logic programming and negation: A survey. *Journal of Logic Programming*, 19(20):9–71, 1994.

[8] S. Austin and G. I. Parkin. *Formal Methods: A Survey (NPL).* DITC Office, Teddington, Middlesex, TW11 0LW, UK, March 1993.

[9] R. Bagnara, E. Zaffanella, and P. M. Hill. Enhanced sharing analysis techniques: A comprehensive evaluation. In M. Gabbrielli and F. Pfenning, editors, *Proceedings of the 2nd International ACM SIGPLAN. Conference on*

Principles and Practice of Declarative Programming, pages 103–114, Montreal, Canada, 2000. ACM Press.

[10] J. C. Bicarregui, editor. *Proof in VDM: case studies.* Springer-Verlag FACIT Series, 1998.

[11] J.C. Bicarregui, D. Clutterbuck, G. Finnie, H. Haughton, K. Lano, H. Lesan, D. W. R. M. Marsh, B. M. Matthews, M. R. Moulding, A. R. Newton, B. Ritchie, T. G. A. Rushton, and P. N. Scharbach. Formal Methods Into Practice: case studies in the application of the B method. *IEE Proceedings on Software Engineering*, 144(2):119–133, April 1997.

[12] R. E. Bloomfield and P. K. D. Froome. The Application of Formal Methods to the Assessment of High Integrity Software. *IEEE Transactions on Software Engineering*, SE-12(9):988–993, 1986.

[13] S. Blythe. Object oriented design of a Z-to-Prolog translator. Undergraduate Project Report (BSc Honours degree), University of Leeds, 1995.

[14] T. Bolognesi and E. Brinksma. Introduction to the ISO specification language LOTOS. *Computer Networks and ISDN Systems*, 14(1):25–59, 1987.

[15] J. Bowen and V. Stavridou. Safety-Critical Systems, Formal Methods and Standards. *Software Engineering Journal*, 8(4):189–209, July 1993.

[16] J. P. Bowen and M. J. C. Gordon. Z and HOL. In *Z User Workshop, Cambridge, June 1994*, pages 141–167. Springer-Verlag, 1994.

[17] P. T. Breuer and J. Bowen. Towards Correct Executable Semantics for Z. In *Z User Workshop, Cambridge, June 1994*, pages 185–209. Springer-Verlag, 1994.

[18] P. T. Breuer and J. Bowen. A concrete grammar for z. Technical Report PRG-TR-22-95, Oxford University Computing Laboratory, Oxford UK, September 1995.

[19] A. Bryant, A. S. Evans, L. Semmens, R. Milovanovic, S. Stockman, M. Norris, and C. Selley. Using Z to rigorously review a specification of a network management system. In *ZUM'95 – 9th International Conference of Z User's, September 1995, Limerick, Ireland*, pages 423–433. Lecture Notes in Computer Science 967, Springer-Verlag, Heidelberg, 1995.

[20] R. M. Burstall and J. Darlington. A Transformation System for Developing Recursive Programs. *Journal of the ACM*, 24(1):44–67, 1977.

[21] K. L. Clark, F. G. McCabe, and S. Gregory. IC-Prolog Language Features. In K. L. Clark and S. A. Tarnlund, editors, *Logic Programming*, pages 253–266. Academic Press, London, 1982.

[22] I. M. Copi. *Symbolic Logic.* Macmillan, New York, 1979.

[23] P. Cousot and R. Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fix-points. In *Proc. 4th ACM Symposium on the Principles of Programming Languages*, pages 238–252, 1977.

[24] P. Cousot and R. Cousot. Abstract Interpretation and Application to Logic Programs. *The Journal of Logic Programming*, 13:103–179, 1992. The editor of Journal of Logic Programming has mistakenly published the unreadable galley proof. For a correct version of this paper, see http://www.di.ens.fr/c̃ousot.

[25] P. Cousot and R. Cousot. Comparing the Galois connection and widening/narrowing approaches to abstract interpretation. Invited paper. In M. Bruynooghe and M. Wirsing, editors, *Proceedings of the International Workshop Programming Language Implementation and Logic Programming, PLILP '92,*, Leuven, Belgium, 13–17 August 1992, Lecture Notes in Computer Science 631, pages 269–295. Springer-Verlag, Berlin, Germany, 1992.

[26] Dan Craigen, Irwin Meisels, and Mark Saaltink. Analysing Z Specifications with Z/EVES. In *Industrial-Strength Formal Methods in Practice*. Springer-Verlag, September 1999.

[27] R. E Davies. Runnable Specification as a Design Tool. In K. L. Clark and S. A. Tarnlund, editors, *Logic Programming*, pages 141–152. Academic Press, London, 1982.

[28] Y. Deville and K. K. Lau. Logic Program Synthesis. *Journal of Logic Programming*, 20:321–350, July 1994.

[29] A. J. Dick, P. J. Krause, and J. Cozens. Computer Aided Transformation of Z into Prolog. In *Proceedings of the 4th Annual Z Users Meeting, Oxford University Computing Laboratory PRG*, pages 71–85. Springer-Verlag, December 1989.

[30] J. Dick and A. Faivre. Automating the Generation and Sequencing of Test Cases from Model-Based Specifications. In J. C. P. Woodcock and P. G. Larsen, editors, *FME '93: Industrial-Strength Formal Methods, First International Symposium of Formal Methods Europe (Proceedings), April 19-23 1993, Odense, Denmark*, pages 268–284. Lecture Notes in Computer Science 670, Springer-Verlag, Germany, 1993.

[31] A. Diller. *Z: An Introduction to Formal Methods.* John Wiley, UK, 1990.

[32] V. Doma and R. Nicholl. EZ: A System for Automatic Prototyping of Z Specifications. In S. Prehn and W. J. Toetnel, editors, *VDM '91, Formal Software Development Methods, Fourth International Symposium of VDM Europe, October 21-25 1991, Noordwijkerhout, The Netherlands*, volume 1, pages 189–203. Springer-Verlag, Lecture Notes in Computer Science 551, Germany, 1991.

[33] G. Dondossola. Formal methods in the development of safety critical knowledge-based components. In Frank van Harmelen, editor, *Proceedings of the KR'98 European Workshop on Validation and Verification of Knowledge-Based Systems (CEUR Workshop Proceedings, Vol-16)*, 1998.

[34] E.H. Durr and J. van Katwijk. VDM++ – a formal specification language for object-oriented designs. In *Computer Systems and Software Engineering: Proceedings of CompEuro'92*, pages 214–219. IEEE Computer Society press, 1992.

[35] S. Easterbrook and J. Callahan. Independent Validation of Specifications: A coordination headache. In *Proceedings of the Fifth Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE'96)*, pages 232– 237. IEEE Computer Science Press, 1996.

[36] H. B. Enderton. *Elements of Set Theory.* Academic Press, New York, 1977.

[37] M. Flynn, T. Hoverd, and D. Brazier. Formaliser – an interactive support tool for Z. In J. E. Nicholls, editor, *Z User Workshop, Oxford 1989*, Workshops in Computing, pages 128–141. Springer-Verlag, 1990.

[38] N. E. Fuchs. Specifications are (Preferably) Executable. *Software Engineering Journal*, pages 323–334, September 1992.

[39] K. Futatsugi, J. A. Goguen, J. P. Jouannaud, and J. Meseguer. Principles of OBJ2. *Proceedings of the 12th ACM Symposium on Principles of Programming Languages (New Orleans)*, pages 52–66, 1985.

[40] G. R. Gladden. Stop the Life-Cycle, I Want to Get Off. *ACM SIGSOFT Software Engineering Notes*, 7(2):35–39, 1982.

[41] H. S. Goodman. The Z-into-Haskell Tool-kit: An illustrative case study. In J. P. Bowen and M. G. Hinchey, editors, *ZUM'95 – 9th International Conference of Z User's, September 1995, Limerick, Ireland*, pages 374–388. LNCS 967, Springer-Verlag, 1995.

[42] A. Gravell and P. Henderson. Executing formal specifications need not be harmful. *Software Engineering Journal*, 11(2):104 –110, March 1996.

[43] Wolfgang Grieskamp. A Computation Model for Z Based on Concurrent Constraint Resolution. In *Proceedings of the ZB 2000 Conference, York, UK*, pages 414–432, 2000.

[44] STARTS Public Purchaser Group. *The STARTS Purchasers' Handbook: Procuring Software-Based Systems*. NCC publications, UK, 1989.

[45] The VDM Tool Group. VDM-SL Toolbox User Manual. Technical report, IFAD, October 2000. ftp://ftp.ifad.dk/pub/vdmtools/doc /userman_letter.pdf.

[46] The VDM Tool Group. VDM++ Toolbox User Manual. Technical report, IFAD, October 2000. ftp://ftp.ifad.dk/pub/vdmtools/doc /usermanpp_letter.pdf.

[47] C. A. Gunter and D. S. Scott. Semantic domains. In J. van Leeuwen, editor, *HandBook of Theoretical Computer Science: Formal Models and Semantics (Vol B)*, pages 635 – 674. Elsevier, 1990.

[48] J. V. Guttag and J. J. Horning. The Algebraic Specification of Abstract Data Types. *Acta Informatica*, 10:27–52, 1978.

[49] J. V. Guttag and J. J. Horning. Formal Specification as a Design Tool. *Proceedings of the 7th ACM Symposium, Principles of Programming Languages*, pages 251–261, 1980.

[50] A. G. Hamilton. *Numbers, Sets and Axioms. The Apparatus of Mathematics.* Cambridge University Press, UK, 1982.

[51] I. Hayes, editor. *Specification Case Studies (Second Edition).* Prentice Hall International (UK) Ltd, 1993.

[52] I. J. Hayes and C. B. Jones. Specifications are not (Necessarily) Executable. *Software Engineering Journal*, 4(6):330–338, 1989.

[53] M. P. E. Heimdahl and N. G. Leveson. Completeness and consistency in hierarchical state-based requirements. *IEEE Transactions on Software Engineering*, 22(6):363 – 377, 1996.

[54] M. C. Henson. Program Development in the Constructive Set Theory TK. *Formal Aspects of Computing*, 1(2):173–192, 1989.

[55] M. A. Hewitt, C. M. O'Halloran, and C. T. Sennett. Experiences with PiZA, an Animator for Z. In *ZUM'97: the Z Formal Specifation Notation , 10th International Conference of Z Users, Reading, UK*, pages 37 – 51. LNCS 1212, Springer-Verlag, 1997.

[56] P. M. Hill and J. W. Lloyd. *The Gödel Programming Language.* MIT Press, 1994.

[57] C. A. R. Hoare. *Communicating Sequential Processes.* Prentice Hall International, 1986.

[58] C. A. R. Hoare. An Overview of Some Formal Methods for Program Design. *IEEE Computer*, 20(9):85 – 91, September 1987.

[59] C. Hogger. *Introduction to Logic Programming.* Academic Press, London, 1984.

[60] D. Jackson. Structuring Z specifications with views. *ACM Transactions on Software Engineering and Methodology*, 4(4):365 – 398, October 1995.

[61] P. H. Jesty, T. F. Buckley, and M. M. West. The Development of Safe Advanced Road Transport Telematic Software. *Microprocessors and Microsystems*, 17(1):37–46, 1993.

[62] Xiaoping Jia. A Tutorial of ZANS – A Z Animation System. Internal Report, Division of Computer Science, Telecomminication, and Information Systems, DePaul University, Chicago, Illinois, USA, 1995.

[63] M. Johnson and P. Sanders. From Z specifications to functional implementations. In J. E. Nicholls, editor, *Z User Workshop, Oxford 1989*, Workshops in Computing, pages 86–112. Springer-Verlag, 1990.

[64] C. B. Jones. *Systematic Software Development Using VDM.* Prentice Hall International, London, 1986.

[65] F. Kluzniak and M. Milkowska. Spill - a logic language for writing testable requirements specifications. *Science of Computer Programming*, 28((2-3)):193–223, April 1997.

[66] G. T. Kneebone. *Mathematical Logic and the Foundations of Mathematics.* Van Nostrand, 1963.

[67] R. D. Knott and P. J. Krause. An Approach to Animating Z Using PROLOG. Technical Report A1.1, Alvey Project SE/065, University of Surrey, 1988.

[68] R. D. Knott and P. J. Krause. LIBRARY SYSTEM: An Example of the Rapid Prototyping of a Z Specification in PROLOG. Technical Report A1.2, Alvey Project SE/065, University of Surrey, 1988.

[69] R. D. Knott and P. J. Krause. On the Derivation of an Effective Animation: Telephone Network Case Study. Technical Report A1.3, Alvey Project SE/065, University of Surrey, 1988.

[70] R. D. Knott and D. Pitt. Implementation of Z Specifications Using Program Transformation. Technical report, University of Surrey, 1987.

[71] R. A. Kowalski. *Logic For Problem Solving.* North Holland Artificial Intelligence Series, New York, 1979.

[72] K. K. Lau and S. D. Prestwich. Top-down Synthesis of Recursive Logic Procedures fom First-order Logic Specifications. In *7th International Conference on Logic Programming*, pages 668–684, 1990.

[73] J. W. Lloyd. *Foundations of Logic Programming (Second, Extended Edition).* Springer-Verlag, Berlin, 1987.

[74] R. R. Lutz. Analyzing software requirements errors in safety-critical, embedded systems. In *IEEE International Symposium on Requirements Engineering*, pages 126–133, San Diego, CA, 1993. IEEE Computer Society Press.

[75] K. C. Mander and F. Polack. Rigorous specification using structured systems analysis and Z. *Information and Software Technology*, 37(5–6):285–291, May–June 1995.

[76] Z. Manna and R. Waldinger. *The Logical Basis for Computer Programming, Vol 1: Deductive Reasoning.* Addison-Wesley, USA, 1985.

[77] J. Marcus and K. Nuthall. Researchers Join War on Terror. Times Higher Educational Supplement, January 4th, 2002.

[78] K. Marriot and H. Søndergaard. Bottom-up Dataflow Analysis of Normal Logic Programs. *The Journal of Logic Programming*, 13:181–204, 1992.

[79] T. L. McCluskey and M. M. West. The automated refinement of a requirements domain theory. *Journal of Automated Software Enginnering, Special Issue on Inductive Programming*, 8(2):193 – 216, 2001.

[80] T.L. McCluskey, J.M. Porteous, Y. Naik, C.N. Taylor, and S. Jones. A requirements capture method and its use in an air traffic control application. *Software - Practice and Experience*, 25(1):47–71, 1995.

[81] I. Meisels and M. Saaltink. The Z/EVES reference manual (for version 1.3). Technical Report TR-96-5493-03b, ORA Canada, Ontario K1Z 6X3, 1996. (July 2000: version 2.1 of Z/EVES is now available and includes a graphical user interface).

[82] R. Milner. *Communication and Concurrency.* Prentice-Hall International, Eaglewood Cliffs, NJ, US, 1989.

[83] Richard Moore and Peter Froome. MURAL and SpecBox. In *VDM '91: Formal Software Development Methods*, pages 672–674. Lecture Notes in Computer Science 551, Springer-Verlag, October 1991.

[84] C. Morgan and B. Sufrin. Unix Filing system. In I. Hayes, editor, *Specification Case Studies (Second Edition)*, pages 45–78. Prentice Hall International (UK) Ltd, 1993.

[85] P. Mukherjee. Computer-aided validation of formal specifications. *Software Engineering Journal*, 10(4):133–140, July 1995.

[86] G Myers. *The Art of Software Testing.* Wiley, 1979.

[87] P. Neumann. ACM forum on risks to the public in computers and other related systems. Software Engineering Notes.

[88] T. Nicholson and N. Foo. A denotational semantics for prolog. *ACM Trans. on Programming Languages and Systems*, 11(4):650–665, 1989.

[89] P. G. Larsen and B. S. Hansen and H. Brunn N. Plat and H. Toetenel and D. J. Andrews and J. Dawes and G. Parkin and others. Information technology — Programming languages, their environments and system software interfaces — Vienna Development Method — Specification Language — Part 1: Base language, December 1996.

[90] Z Standards Panel. Z Notation (final committee draft, cd 13568.2). Technical report, ISO Panel JTC1/SC22/WG1 (Rapporteur Group for Z), August 1999.

[91] D. L. Parnas. 'Formal methods' technology transfer will fail. *Journal of Systems and Software*, 40(3):195–198, 1998.

[92] A. Pettorossi and M. Proietti. Synthesis and transformation of logic programs using unfold/fold proofs. *Journal of Logic Programming*, 41:197–230, 1999.

[93] R. S. Pressman. *Software Engineering, A Practitioners Approach, Fifth Edition.* Mc Graw Hill, 2000. Adapted by Darrel Ince.

[94] M Roper. *Software Testing.* McGraw-Hill, 1994.

[95] J. Rushby. Formal Methods and the Certification of Critical Systems. Technical Report CSL-93-7, Computer Science Laboratory, SRI International, Menlo Park CA 94025 USA, December 1993.

[96] Mark Saaltink. Domain checking Z specifications. In C. M. Holloway and K. J. Hayhurst, editors, *Fourth NASA Langley Formal Methods Workshop (LFM '97), NASA Conference Publication 3356*, 1997. Also obtainable from ORA, Canada, Report CP-97-6018-85.

[97] D. A. Schmidt. *Denotional Semantics, A Methodology for Language Development.* Allyn and Bacon, Inc, 7 Wells Avenue, Newton, Massachusetts, US, 1986.

[98] S. Schneider. *The B-Book.* Palgrave-Macmillan, 2001.

[99] Shapiro. *Algorithmic Program Debugging.* MIT Press, Cambridge, MA, 1983.

[100] L. B. Sherrell and D. L. Carver. FunZ: An intermediate specification language. *The Computer Journal*, 38(3):193–206, 1995.

[101] L.B. Sherrell and D.L. Carver. Funz designs - a bridge between Z specifications and Haskell implementations. In *Nineteenth Annual International Computer Software and Applications Conference (COMPSAC'95)*, pages 12–17. IEEE, Computer Soc Press, Los Alamitos, 1995.

[102] Gerrard Software. *ObjEx User Reference Manual.* UK, 1988.

[103] J. M. Spivey. *Understanding Z, A Specification Language and its Formal Semantics.* Cambridge University Press, UK, 1988.

[104] J. M. Spivey. *The Z Notation: A Reference Manual.* Prentice Hall International (UK) Ltd, UK, 1989.

[105] J. M. Spivey. *The Z Notation: A Reference Manual (2nd ed.).* Prentice Hall International (UK) Ltd, UK, 1992.

[106] S. Stepney. Testing as abstraction. In J. P. Bowen and M. G. Hinchey, editors, *ZUM'95 – 9th International Conference of Z User's, September 1995, Limerick, Ireland*, pages 137 – 151. LNCS 967, Springer-Verlag, 1995.

[107] S. Stepney. New horizons in formal methods. *The Computer Bulletin*, pages 24–26, January 2001.

[108] S. Stepney and S. P. Lord. Formal specification of an access control system. *Software - Practice and Experience*, 17(9), 1987.

[109] Bill Stoddart, Steve Dunne, and Andy Galloway. Undefined expressions and logic in Z and B. *Formal Methods in System Design: An International Journal*, 15(3):201–215, November 1999.

[110] P. Suppes. *Axiomatic Set Theory, (2nd Ed).* Dover, New York, 1972.

[111] Martyn Thomas. Development Methods for Trusted Computers. *Formal Aspects of Computing*, 1(1):5–18, 1989.

[112] Muffy Thomas. The story of the Therac-25 in LOTOS. *High Integrity Systems Journal*, 1:13–17, 1994.

[113] Helen Treharne, Jonathan Draper, and Steve Schneider. Test case preparation using a prototype. In Didier Bert, editor, *B'98 : The 2nd International B Conference, Recent Advances in the Development and Use of the B Method*, volume 1393 of *Lecture Notes in Computer Science (Springer-Verlag)*, pages 293–311, Montpellier, April 1998. LIRRM Laboratoire d'Informatique, de Robotique et de Micro-électronique de Montpellier, Springer Verlag.

[114] Mark Utting. Data structures for Z testing tools. In *FM-TOOLS 2000, Germany, July 2000*, 2000. Obtainable as TR 2000-07, Information Faculty, University of Ulm.

[115] S. H. Valentine. The Programming Language Z $^{--}$. *Information and Software Technology*, 37(5):293 – 301, May 1995.

[116] M. M. West. Safety and Social Aspects of Intelligent Vehicle Highway Systems. Technical Report 94.13, School of Computer Studies, University of Leeds, April 1994. ftp://ftp.comp.leeds.ac.uk/scs/doc/reports/1994/.

[117] M. M. West. Types and Sets in Gödel and Z. In J. P. Bowen and M. G. Hinchey, editors, *ZUM'95 – 9th International Conference of Z User's, September 1995, Limerick, Ireland*, pages 389–407. Lecture Notes in Computer Science 967, Springer-Verlag, Heidelberg, 1995.

[118] M. M. West, T. F. Buckley, P. H. Jesty, and K. M. Hobley. *SG7: Development of a Case Study Leading to Demonstrators*. DRIVE Project *DRIVE Safely* (V1051), DRIVE Central Office, CEC, Brussels, April 1991.

[119] M. M. West and B. M. Eaglestone. Software Development: Two Approaches to Animation of Z Specifications Using Prolog. *Software Engineering Journal*, 7(4):264–276, July 1992.

[120] M. M. West, K. M. Hobley, and P. H. Jesty. The Use of Formal Methods in ATT Systems, a Safety-critical Case Study. Technical Report 95.29, School of Computer Studies, University of Leeds, October 1995. ftp://ftp.comp.leeds.ac.uk/scs/doc/reports/1995/.

[121] M. M. West and T. L. McCluskey. The application of machine learning tools to the validation of an air traffic control domain theory. *International Journal on Artificial Intelligence Tools*, 10(4):613 – 637, December 2001.

[122] B. A. Wichmann. A Development Model for Safety-Critical Software. In B. A. Wichmann, editor, *Software in Safety-Related Systems (IEE/BCS Joint Study Report)*, pages 211–223. John Wiley and Sons, UK, 1992. ISBN 0471-93474-7.

[123] B. A. Wichmann, editor. *Software in Safety-Related Systems (IEE/BCS Joint Study Report)*. John Wiley and Sons, UK, 1992.

[124] M. Winikoff, P. Dart, and E. Kazmierczak. Rapid prototyping using formal specifications. In *Proceedings of the 21st Australasian Computer Science Conference, ACSC'98*, pages 279–293. Springer, 1998.

[125] J. B. Wordsworth. *Software Engineering with B.* Addison Wesley, UK, 1996.

Appendix A

ZF SET THEORY

A.1 Introduction

The axioms of the Zermelo-Fraenkel (ZF) system of axiomatic set theory are necessary for the main part of the thesis and they are outlined here. ZF set theory was developed because 'naive' set theory, introduced by Cantor had given rise to contradictions. Cantor's paper begins[1]:

> "By a 'set'(Menge) we are to understand any comprehension (Zusammenfassung) into a whole M of definite and separate objects m of our intuition or our thought"

This 'comprehension' was eventually replaced by the abstract idea that every property (predicate) gives rise to the set of all things which possess the property. Nevertheless this still gives rise to contradictions, and the ZF system of axiomatic set avoids these by considering every object to be a set.

ZF is a first order theory with equality where the only terms are variables; there are no constant or function letters. ZF has primitive relationship $\in$ denoting membership so that $x \in y$ is the wff denoting the fact that 'x is a member of y'. The system shown here is 'ZF-without-replacement' and is taken from [50]. The replacement axiom is one which can construct a new set as an image of an old, and this is not included for Z and so not included here.

[1] The quotation is from [66].

A.2 ZF Axioms

A.2.1 ZF1 Axiom of Extensionality (Set Equality)

$$y = w \Rightarrow \forall\, x \bullet (x \in y \Leftrightarrow x \in w)$$

ZF1 defines equality between two sets, that they are equal if they have the same members. The definition of subset follows from ZF1.

Definition 1: Subset ($\subseteq$)

$$x \subseteq y \Leftrightarrow (\forall\, w \bullet w \in x \Rightarrow w \in y).$$

A.2.2 ZF2 Null Set Axiom

This can be deduced from the other axioms, but is included as it is standard.

$$\exists\, x\ \ \forall\, y \bullet (\neg\, y \in x)$$

The axiom defines an empty set or $\varnothing$, which can be shown to be unique by ZF1.

A.2.3 ZF3 Pairing Axiom

$$\forall\, x\ \ \forall\, y\ \ \exists\, w\ \ \forall\, z \bullet (z \in w \Leftrightarrow (z = x \vee z = y))$$

Given any sets x and y there is a set w whose elements are x and y. This ‘unordered pair’ is usually denoted $\{x, w\}$, which is the same as $\{y, x\}$ from ZF1. By setting $x = y$, the set $\{x, x\}$ (i.e. the singleton set $\{x\}$) is obtained.

A.2.4 ZF4 Union Axiom

$$\forall\, x\ \ \exists\, y\ \ \forall\, w \bullet (w \in y \Leftrightarrow \exists\, z \bullet (z \in x \wedge w \in z))$$

Given any set x whose elements are sets, there is a set y which has as its elements all elements of elements of x. The definition of ‘distributed union’ and ‘union’ follows.

Definition 2: Distributed Union ($\bigcup$) **and Union** ($\cup$)

The set y from ZF4 is denoted $\bigcup x$; $\cup$ is then defined: $(a_1 \cup a_2) = \bigcup\{a_1, a_2\}$ or

$$(z \in x \cup y) \Leftrightarrow z \in x \vee z \in y.$$

A.2.5 ZF5 Power Set Axiom

$$\forall x \ \exists y \ \forall w \bullet (w \in y \Leftrightarrow w \subseteq x)$$

Given any set x there is a set y, denoted $\mathbb{P}(x)$, the power set of x, which has as its elements all subsets of x.

A.2.6 ZF6 The Axiom of Separation

Given any wff of ZF (as defined above), denoted $\mathcal{A}(z)$ with free variable z,

$$\forall x \ \exists y \ \forall z \bullet (z \in y \Leftrightarrow z \in x \wedge \mathcal{A}(z)).$$

In other words set y is a subset of x containing those members z of x for which $\mathcal{A}(z)$ holds. This replaces the stronger ZF7 (or replacement axiom), which is omitted. The definition of intersection of two sets x, y follows from ZF6, in that the wff $\mathcal{A}(z)$ becomes $z \in w$.

Definition 3: Intersection ($\cap$)

$$x \cap w = y \Leftrightarrow \forall z \bullet (z \in y \Leftrightarrow z \in x \wedge z \in w)$$

A.2.7 ZF8 Infinity Axiom

$$\exists x \bullet (\varnothing \in x \wedge \forall y \bullet (y \in x \Rightarrow y \cup \{x\} \in x)).$$

Taken with the previous axioms it can be proved that this constructs the natural numbers . ZF theory has all sets derivable from the empty set $\varnothing$, and the natural numbers $0, 1, 2, \ldots$ are

$$\varnothing, \ \{\varnothing\}, \ \{\varnothing, \{\varnothing\}\}, \ \ldots$$

In other words each number is the set of all smaller natural numbers.

Appendix B

Theory of Finite Sets – Key Axioms

The following key axioms are taken from [76]. There are two unary predicate symbols $element(u), set(x)$[1] and $(u \circ x)$ denotes a binary insertion function: the result of inserting element u in set x. The constant symbol $\varnothing$ denotes the empty set. A binary predicate symbol $u \in x$ denotes membership. The text uses a distinctive notation; for example quantification is expressed $\forall\, set(x) : p(x)$ and logical and is *and*. We retain the quantification notation, but we write & for conjunction, for compatibility with later work. The following are examples of axioms in the theory, for set generation, membership, set equality and set union. There set equality axioms are additional to the usual equality axioms.

Generation **Empty Set** $set(\varnothing)$

General Case $\forall\, element(u) : \forall\, set(x) : set(u \circ x)$.

Membership $\neg\ \exists\, element(z) : (z \in \varnothing)$

$\forall\, element(z) : \forall\, element(u) : \forall\, set(x) : z \in (u \circ x) \Leftrightarrow z = u \vee z \in x.$

Set equality The following axioms apply to sets:

element multiplicity $\forall\, element(u) : \forall\, set(x) : u \circ (u \circ x) = (u \circ x)$

[1]an element can also be a set

element exchange $\forall\, element(u) : \forall\, element(v) : \forall\, set(x) : u \circ (v \circ x) = v \circ (u \circ x)$.

Induction Principle For each sentence $\mathcal{F}(x)$, u not free in x, the following is an axiom:

$$\begin{array}{l} \text{if} \;\; (\mathcal{F}(\{\} \;\& \\ \quad \forall\, element(u) : \forall\, set(x) : \mathcal{F}(x) \Rightarrow \mathcal{F}(u \circ x)) \\ \text{then} \;\; \forall\, set(x) : \mathcal{F}(x) \end{array}$$

Union with empty set $\forall\, set(x) : \varnothing \cup x = x$

Union: insertion $\forall\, element(u) : \forall\, set(x) : \forall\, set(y)(u \circ x) \cup y = u \circ (x \cup y)$

Set constructor The general form of the set constructor (unary) function for each $\mathcal{F}(u)$ is $\{u : u \in x \;\&\; \mathcal{F}(u)\}$

The axioms are as follows:

Empty Set

$$\{u : u \in \varnothing \;\&\; \mathcal{F}(u)\} = \varnothing$$

General Case

$$\begin{array}{l} \forall\, element(v)\, \forall\, set(x) : \\ \quad \{u : u \in v \circ x \;\&\; \mathcal{F}(u)\} = \\ \qquad \mathit{if}\;\; \mathcal{F}(v) \mathit{then}\;\; v \circ \{u : u \in x \;\&\; \mathcal{F}(u)\} \\ \qquad \mathit{else}\;\; \{u : u \in x \;\&\; \mathcal{F}(u)\} \end{array}$$

This yields the set of all elements u of set x such that $\mathcal{F}(u)$ is true.

Sets can be conceptualised as lists where multiplicity and order of elements is irrelevant, and there are many representations of the same set. Set operations resembling those of ZF can be derived from the axioms using the induction principle. For example 'intersection' can be obtained from set comprehension. A sample follows:

Set Intersection $\forall\, set(x) : \forall\, set(y) : x \cap y = \{u : u \in x \;\&\; u \in y\}$

Subset $\forall\, set(x) : \forall\, set(y) : x \subseteq y \Leftrightarrow \forall\, element(w) : (w \in x \Rightarrow w \in y)$

Union $\forall\, element(u)\, \forall\, set(x) : \forall\, set(y) : u \in (x \cup y) \Leftrightarrow u \in x \vee u \in y$

Set equality - derived $\forall\, set(x) : \forall\, set(y) : x = y \Leftrightarrow x \subseteq y \;\&\; y \subseteq x$

Appendix C

Library of Set Code

C.1 Introduction

This appendix includes the Gödel code discussed in the main part of the thesis. The first element of each module is the EXPORT part, and the second is the LOCAL part.

C.1.1 Library code

This section contains the library code for sets and relations.

Library Code: EXPORT

```
EXPORT Lib.
IMPORT   Sets, Lists, Integers.

CONSTRUCTOR OP/2.

FUNCTION     OrdPair : x * y  ->  OP(x,y).


%% ZF Sets and Relations

%% Distributed union

PREDICATE  DUnion:  Set(Set(a)) * Set(a).

% Cardinality of a set
PREDICATE    Card: Set(a) * Integer.
```

```
%Relation
PREDICATE   Rel : Set(OP(a,b)) * Set(a)  * Set(b).

%Partial Function
PREDICATE   PF  : Set(OP(a,b)) * Set(a)  * Set(b).

PREDICATE   TF  : Set(OP(a,b)) * Set(a)  * Set(b).

PREDICATE   PINJ  : Set(OP(a,b)) * Set(a)  * Set(b).

PREDICATE   TINJ  : Set(OP(a,b)) * Set(a)  * Set(b).

PREDICATE   PSURJ : Set(OP(a,b)) * Set(a)  * Set(b).

PREDICATE   TSURJ : Set(OP(a,b)) * Set(a)  * Set(b).

PREDICATE   TONF  : Set(OP(a,b)) * Set(a)  * Set(b).

PREDICATE  DomContents  : Set(OP(a,b))  * Set(a) .

PREDICATE  RanContents  : Set(OP(a,b))  * Set(b) .

PREDICATE  Composition : Set(OP(a,b)) * Set(OP(a,c)) * Set(OP(c,b)).

PREDICATE  Inverse : Set(OP(a,b))  * Set(OP(b,a)).

PREDICATE DomRestrict : Set(OP(a,b)) * Set(a) * Set(OP(a,b)).

PREDICATE RanRestrict : Set(OP(a,b)) * Set(b) * Set(OP(a,b)).

PREDICATE DomExclude : Set(OP(a,b)) * Set(a) * Set(OP(a,b)).

PREDICATE RanExclude : Set(OP(a,b)) * Set(b) * Set(OP(a,b)).

PREDICATE FunOveride : Set(OP(a,b)) * Set(OP(a,b)) * Set(OP(a,b)).

PREDICATE LambdaSquareNo : Set(OP(Integer,Integer)) * Set(Integer).

PREDICATE IsSequ          : Set(OP(Integer,a)) * Set(a) * Integer.

PREDICATE  HeadSequ       : a   * Set(OP(Integer,a)).
```

%% GCD - taken from Hill and Lloyd

PREDICATE Gcd : Integer * Integer * Integer.

PREDICATE CommonDivisor : Integer * Integer * Integer.

Library Code: LOCAL

```
LOCAL Lib.
IMPORT  Sets, Lists, Integers.
```

% ZF Sets and Relations

% Distributed union

```
        DUnion(x, y)  <- y = {z :  SOME [w] (w In x & z In w) }.
```

% Cardinality of a set

```
        Card({}, 0).
        Card(set, sub_total + 1)
          <- x In set & Card(set\{x}, sub_total).
```

% Relation

```
        Rel(rel, s1, s2) <- ALL [z,x,y]
                        ( z In rel & (z = OrdPair(x,y))
                          ->  (x In s1) & (y In s2)).
```

% partial function

```
        PF(pf, s1, s2) <- ALL [z,x,y]
              ( z In pf & (z = OrdPair(x,y))
                 ->  (x In s1) & (y In s2)  &
                  ALL [u] (OrdPair(x, u) In pf -> u = y)).
```

% total function

```
        TF(tf, s1, s2) <- ALL [z,x,y]
              ( z In tf &  (z = OrdPair(x,y))
                 ->  (x In s1) & (y In s2)  &
                    DomContents(tf, s1) &
                    ALL [u] (OrdPair(x, u) In tf -> u = y)).
```

```
% partial injection
        PINJ(pinj, s1, s2) <- ALL [z,x,y]
                   (z In pinj & (z = OrdPair(x,y))
            ->  (x In s1) & (y In s2)  &
                 ALL [u1] (OrdPair(x, u1) In pinj -> u1 = y)
                & ALL [u2] (OrdPair(u2, y) In pinj ->  u2 = x)).

% total injection
       TINJ(tinj, s1, s2) <- PINJ(tinj, s1, s2) & DomContents(tinj,s1).

% partial surjection
PSURJ(psurj, s1, s2) <- PF(psurj, s1, s2) & RanContents(psurj, s2).

       DomContents(x, y) <- y = { dom: OrdPair(dom,_) In x}.

       RanContents(x, y) <- y = { ran: OrdPair(_,ran) In x}.


       Composition(x, y, z) <- x = {OrdPair(a,b):
           SOME [u] (OrdPair(a,u) In y & OrdPair(u,b) In z )}.


       Inverse (x, y) <- x = {OrdPair(a,b): OrdPair(b,a) In y}.

% Relation x is Relation z with  domain restricted to (set) y.
       DomRestrict(x, y, z) <-
           x = {OrdPair(a,b): OrdPair(a,b) In z & a In y}.

% Relation x is Relation z with  range restricted to (set) y.
       RanRestrict(x, y, z) <-
           x = {OrdPair(a,b): OrdPair(a,b) In z & b In y}.


% Relation x is Relation z with (set) y deleted from its domain.
       DomExclude(x, y, z) <-
             x = {OrdPair(a,b): OrdPair(a,b) In z & ~(a In y)}.


% Relation x is Relation z with (set) y deleted from its range.
       RanExclude(x, y, z) <-
             x = {OrdPair(a,b): OrdPair(a,b) In z & ~(b In y)}.

% Function x is function y overidden by function z
```

```
        FunOveride(x, y, z) <- w = {OrdPair(a,b): OrdPair(a,b) In y &
                              ~SOME[u] (OrdPair(a,u) In z)} &
                                 x = w + z.

% Example of lambda expression, x is set of squares of numbers in y
        LambdaSquareNo(x, y) <- x = {OrdPair(n, n*n): n In y}.

% Example of generic definition- head of sequence.
% First define a sequence
        IsSequ(sequ, items, size) <- Size(sequ, size )  &
                       domseq = {n: 1 =< n =< size} &
                         TF(sequ, domseq, items).

% Head of sequence- corresponds to OrdPair(1, item);
% Non-empty sequence
        HeadSequ(item, sequ) <- sequ ~= {}  &
                      OrdPair(1, item) In sequ.

%% GCD - taken from Hill and Lloyd
%% This works for positive and negative integers, i, j.

Gcd(i,j,d)
    <-  CommonDivisor(i, j, d) &
        ~ SOME [e] (CommonDivisor(i, j, e) & e > d).

 CommonDivisor(i, j, d)  <-
                       IF (i = 0 \/ j = 0)
                       THEN
                          d = Max(Abs(i), Abs(j))
                       ELSE
                          1 =< d =< Min(Abs(i), Abs(j)) &
                          i Mod d = 0 &
                          j Mod d = 0.

%% [Lib] <-  Gcd(9,3, x).

%% x = 3 ? ;
%% No
%% [Lib] <-  Gcd(18, 5, y).

%% y = 1 ? ;
%% No
%% [Lib] <- Gcd(18, 2, z).
```

```
%% z = 2 ? ;
%% No
[%% Lib] <- Gcd(-9, 3, x).

%% x = 3 ? ;
%% No
%% [Lib] <- Gcd(9, -3, x).

%% x = 3 ? ;
%% No
%% [Lib] <-  Gcd(-9, -3, x).

%% x = 3 ? ;
%% No
```

C.1.2 Small File System Code

This section contains code for the small file system, *FileSys*, its unconstructive form and queries to both. It also contains code for *UnDef*.

Small File System: EXPORT

```
EXPORT  Demo2.
IMPORT  Lib.
BASE

%%meta variables
  Name, Var, FileId, BindVar.
%% given sets
CONSTANT
      Files, Count, NewFile, X, Y : Var;
      F1, F2, F3  : FileId;
      FileSys,  AddFID, AddFID1, UnDef : Name.

% AddFID1 is an unconstructive form of AddFID

% An example of a schema with undefined bindings is also included.
% variables are X, Y, name is UnDef.


% This function interprets the binding of a variable name to
% an integer value
```

```
FUNCTION Bind1 : Var * Set(FileId) -> BindVar.
FUNCTION Bind2 : Var * Integer -> BindVar.
FUNCTION Bind3 : Var * FileId -> BindVar.
%% defines variable type of schema is set of BindVar

%%Decorations
%% on Set names, ie priming, input, output
FUNCTION    DSet : Var  ->  Var.
FUNCTION    OUT : Var  ->  Var.
FUNCTION    IN : Var  ->  Var.
%%on Schema names, ie priming, del
FUNCTION    DSch : Name  ->  Name.
FUNCTION    Del : Name  ->  Name.


PREDICATE SchemaType: List (BindVar ) * Name.
PREDICATE   D1 : Integer .
PREDICATE   IsFileId : FileId.
```

Small File System: LOCAL and code for *UnDef*

```
LOCAL Demo2.

IsFileId(F1). IsFileId(F2). IsFileId(F3).

%% schema for state FileSys
SchemaType( [ Bind1(Files, files ), Bind2(Count, count )], FileSys)
  <-
    setFID = {x : IsFileId(x) } &
    files Subset setFID &
     count In {y : 0 =< y =< 10} &
     Card(files, count).

%% schema for state FileSys'
SchemaType( [ Bind1(DSet(Files), files1 ), Bind2(DSet(Count), count1 )],
     DSch(FileSys) )  <-
    setFID = {x : IsFileId(x) } &
    files1 Subset setFID &
     count1 In {y : 0 =< y =< 10} &
     Card(files1, count1).

%% schema for state Del FileSys
```

```
SchemaType(binding, Del(FileSys)) <-
    SchemaType(b1,  FileSys) &
    SchemaType(b2, DSch(FileSys)) &
          Append(b1, b2, binding) .


SchemaType( binding, AddFID ) <-

    SchemaType(binding1,  Del(FileSys)) &
   binding1 = [Bind1(Files, files ), Bind2(Count, count ),
     Bind1(DSet(Files), files1 ), Bind2(DSet(Count), count1 )]  &
      Append(binding1, [ Bind3(IN(NewFile), newfile)] , binding) &
     count < 10 &
     setFID = {x : IsFileId(x) } &
     newfile In  setFID &
  ~ ( newfile In files) &
    files1 = files + {newfile} &
     count1 =  count +   1 .

%% queries  and response
[Demo2] <- SchemaType(b, FileSys).

b = [Bind1(Files,{}),Bind2(Count,0)] ? ;

b = [Bind1(Files,{F1}),Bind2(Count,1)] ? ;

b = [Bind1(Files,{F1,F2}),Bind2(Count,2)] ? ;

b = [Bind1(Files,{F1,F2}),Bind2(Count,2)] ? ;

b = [Bind1(Files,{F1,F2,F3}),Bind2(Count,3)] ? ;

b = [Bind1(Files,{F1,F2,F3}),Bind2(Count,3)] ? ;

b = [Bind1(Files,{F1,F2,F3}),Bind2(Count,3)] ?
Yes

[Demo2] <- SchemaType(b, Del(FileSys) ).

b = [Bind1(Files,{}),Bind2(Count,0),Bind1(DSet(Files),{}),
     Bind2(DSet(Count),0)] ? ;

b = [Bind1(Files,{}),Bind2(Count,0),Bind1(DSet(Files),{F1}),
```

```
     Bind2(DSet(Count),1)] ? ;

b = [Bind1(Files,{}),Bind2(Count,0),Bind1(DSet(Files),{F1,F2}),
     Bind2(DSet(Count),2)] ? ;

b = [Bind1(Files,{}),Bind2(Count,0),Bind1(DSet(Files),{F1,F2}),
     Bind2(DSet(Count),2)] ?
Yes

[Demo2] <- SchemaType(b, AddFID).

b = [Bind1(Files,{}),Bind2(Count,0),Bind1(DSet(Files),{F1}),
     Bind2(DSet(Count),1),Bind3(IN(NewFile),F1)] ? ;

b = [Bind1(Files,{}),Bind2(Count,0),Bind1(DSet(Files),{F2}),
     Bind2(DSet(Count),1),Bind3(IN(NewFile),F2)] ? ;

b = [Bind1(Files,{}),Bind2(Count,0),Bind1(DSet(Files),{F3}),
     Bind2(DSet(Count),1),Bind3(IN(NewFile),F3)] ? ;

b = [Bind1(Files,{F1}),Bind2(Count,1),Bind1(DSet(Files),{F1,F2}),
     Bind2(DSet(Count),2),Bind3(IN(NewFile),F2)] ? ;

b = [Bind1(Files,{F1}),Bind2(Count,1),Bind1(DSet(Files),{F1,F2}),
     Bind2(DSet(Count),2),Bind3(IN(NewFile),F2)] ? ;

b = [Bind1(Files,{F1}),Bind2(Count,1),Bind1(DSet(Files),{F1,F3}),
     Bind2(DSet(Count),2),Bind3(IN(NewFile),F3)] ?
Yes


% Queries which instantiate some variables

[Demo2] <- SchemaType(b, AddFID) & b = [Bind1(Files,{}),Bind2(Count,0),
           Bind1(DSet(Files), x), Bind2(DSet(Count),1),Bind3(IN(NewFile), F1)].

b = [Bind1(Files,{}),Bind2(Count,0),Bind1(DSet(Files),{F1}),
          Bind2(DSet(Count),1),Bind3(IN(NewFile),F1)],
x = {F1} ? ;
No

% Error input
```

```
[Demo2] <- SchemaType(b, AddFID) & b = [Bind1(Files,{}),Bind2(Count,0),
          Bind1(DSet(Files), x), Bind2(DSet(Count),1),Bind3(IN(NewFile),          F5)].

Error: undeclared or illegal symbol in term: "F5".
    SchemaType ( b, AddFID ) & b = [ Bind1 ( Files, { } ),
    Bind2 ( Count, 0 ), Bind1 ( DSet ( Files ), x ),
    Bind2 ( DSet ( Count ), 1 ), Bind3 ( IN ( NewFile ), F5
   **here**
    ) ].

[Demo2] <- SchemaType(b, AddFID) & b = [Bind1(Files,{}),Bind2(Count,0),
          Bind1(DSet(Files),x), Bind2(DSet(Count),1),Bind3(IN(NewFile),y)].

b = [Bind1(Files,{}),Bind2(Count,0),Bind1(DSet(Files),{F1}),
     Bind2(DSet(Count),1),Bind3(IN(NewFile),F1)],
x = {F1},
y = F1 ? ;

b = [Bind1(Files,{}),Bind2(Count,0),Bind1(DSet(Files),{F2}),
     Bind2(DSet(Count),1),Bind3(IN(NewFile),F2)],
x = {F2},
y = F2 ? ;

b = [Bind1(Files,{}),Bind2(Count,0),Bind1(DSet(Files),{F3}),
     Bind2(DSet(Count),1),Bind3(IN(NewFile),F3)],
x = {F3},
y = F3 ? ;
No

%% out of range value (of 12) for count

[Demo2] <- SchemaType(b, AddFID) & b = [Bind1(Files,{}),Bind2(Count,0),
          Bind1(DSet(Files),x), Bind2(DSet(Count), 12),
          Bind3(IN(NewFile),y)].

No
```

```
% Unconstructive form of AddFID
SchemaType( binding, AddFID1 ) <-

    SchemaType(binding1,  Del(FileSys)) &
   binding1 = [Bind1(Files, files ), Bind2(Count, count ),
     Bind1(DSet(Files), files1 ), Bind2(DSet(Count), count1 )]  &
      Append(binding1, [ Bind3(IN(NewFile), newfile)] , binding) &
     count < 10 &
     setFID = {x : IsFileId(x) } &
     newfile In  setFID &
  ~ ( newfile In files) &
     files Subset files1 &
     files = files1 \ {newfile} &
     count1 =  count +   1 .

%query to 'unconstructive' schema

[Demo2] <- SchemaType(b, AddFID1).

b = [Bind1(Files,{}),Bind2(Count,0),Bind1(DSet(Files),{F1}),
     Bind2(DSet(Count),1),Bind3(IN(NewFile),F1)] ? ;

b = [Bind1(Files,{}),Bind2(Count,0),Bind1(DSet(Files),{F2}),
     Bind2(DSet(Count),1),Bind3(IN(NewFile),F2)] ? ;

b = [Bind1(Files,{}),Bind2(Count,0),Bind1(DSet(Files),{F3}),
     Bind2(DSet(Count),1),Bind3(IN(NewFile),F3)] ? ;

b = [Bind1(Files,{F1}),Bind2(Count,1),Bind1(DSet(Files),{F1,F2}),
     Bind2(DSet(Count),2),Bind3(IN(NewFile),F2)] ?
Yes
```

```
%%Schema with values undefined

SchemaType( [ Bind2(X, x ), Bind2(Y, y) ], UnDef)
  <- y In {1, 2, 3} & ( (x = 1) \/ (x = 3) \/
    ({x, 1, 2, 3} = {1, 2, 3, 4}) ).

%% query and response

% [Demo2] <- SchemaType( [ Bind2(X, x ), Bind2(Y, y) ], UnDef).

% x = 1,
% y = 1 ? ;

% x = 3,
% y = 1 ? ;
% Floundered.  Unsolved goals are:
% Goal: {v_1,1,2,3}={1,2,3,4}
% Delayed on: v_1
```

C.1.3 Assembler Code

This section contains code for the assembler.

Assembly Code: EXPORT

```
EXPORT  Assembly.
IMPORT  Lib.
BASE
%%meta variables
  Name, Var, BindVar,
%% given sets
 Sym, A, M, Opsym, Int, Op.


FUNCTION    Bind1 :  Var * Set(OP(A, Sym)) -> BindVar.
FUNCTION    Bind2 :  Var * Set(OP(A, Opsym))  -> BindVar.
FUNCTION    Bind3 :  Var * Set(OP(A, Integer))  -> BindVar.
FUNCTION    Bind4 : Var * Set(OP(M, Integer))  -> BindVar.
FUNCTION    Bind5 : Var * Set(OP(Opsym, Integer))  -> BindVar.
FUNCTION    Bind6 : Var * Set(OP(Integer, A))  -> BindVar. % sequence of A
FUNCTION    Bind7 : Var * Set(OP(Integer, M))  -> BindVar. % sequence of M
FUNCTION    Bind8 : Var * Integer -> BindVar.
FUNCTION    Bind9 : Var  * Set(OP(Sym, Integer)) -> BindVar.
FUNCTION    Bind10 : Var  * Set(OP(Integer, Sym)) -> BindVar.

%%Decorations
%% on Set names, ie priming, input, output
FUNCTION    DSet : Var  -> Var.
FUNCTION    OUT : Var  -> Var.
FUNCTION    IN : Var  -> Var.
%%on Schema names, ie priming, del
FUNCTION    DSch : Name  -> Name.
FUNCTION    Del : Name  -> Name.

PREDICATE SchemaType: List (BindVar ) * Name.

PREDICATE SThetaS: Set(List(BindVar)) * Name.

PREDICATE IsSym : Sym.

PREDICATE IsA : A.
```

PREDICATE IsM : M.

PREDICATE IsOpSym : Opsym.

Assembly Code: LOCAL

LOCAL Assembly.
% Assembly Data %

%% for sets a, m, sym, opsym, int
CONSTANT

V1, V2, Loop, Exit : Sym;
A1, A2, A3, A4, A5, A6, A7, A8, A9: A;
M1, M2, M3, M4, M5, M6, M7, M8, M9 : M;
Load, Subn, Store, Compare, Jumple, Jump, Return : Opsym;
SeqA, SeqM, Lab, Op, Ref, Num, Opcode, Operand, Mnem, Rt, St, Core : Var;

Assembly, Assembly_context1, Assembly_context2, Assembly_context3,
IS, Phase1, Phase2, Implementation : Name.

IsA(A1). IsA(A2). IsA(A3). IsA(A4). IsA(A5). IsA(A6).
IsA(A7). IsA(A8). IsA(A9).

IsM(M1). IsM(M2). IsM(M3). IsM(M4). IsM(M5). IsM(M6). IsM(M7).
IsM(M8). IsM(M9).

IsSym(V1). IsSym(V2). IsSym(Loop). IsSym(Exit).

IsOpSym(Load). IsOpSym(Subn). IsOpSym(Store). IsOpSym(Compare).
IsOpSym(Jumple). IsOpSym(Jump). IsOpSym(Return).

%Schemas

%% Context 1
SchemaType([Bind1(Lab,lab), Bind2(Op,op), Bind1(Ref,ref),

```
      Bind3(Num ,num) ], Assembly_context1) <-
  % given sets
  % associated with declarations
      a = {x : IsA(x) }  &
      sym =  {x : IsSym(x) } &
      opsym  =  {x : IsOpSym(x) } &
      int = {x : 0 < x < 5000 } &
   % vars, label, op, ref, num, declared */
     PF(lab, a, sym) &           % A1
     PF(op, a, opsym) &           % A2
     PF(ref, a, sym) &           % A3
     PF(num , a, int) &           % A4
   % predicate */
     DomContents(op, domop) &
     DomContents(ref, domref) &
     DomContents(num, domnum) &
     domref * domnum = {} &      % A5
     a = (domref + domnum +  domop).
                                  % A6
%% Context 2
  SchemaType( [Bind4( Opcode , opcode ), Bind4(Operand,operand )],
              Assembly_context2 ) <-
   %declarations of opcode, operand
      int = {x : 0 < x < 5000 } &
       m = {x : IsM(x) } &                    % M1

       PF( opcode, m, int ) &
       PF(operand  , m, int ) &                % M3
    %predicate
       DomContents(opcode, domopcode) &
       DomContents(operand, domoperand ) &
       m = (domopcode +  domoperand).          % M4

%% Context 3
 SchemaType( [ Bind5( Mnem, mnem )], Assembly_context3 ) <-

         int = {x : 0 < x < 5000 } &
         opsym = {x : IsOpSym(x) } &
         PF( mnem, opsym, int ).                % M4

[Bind6(IN(SeqA),seqa ), Bind7(OUT(SeqM),seqm),
      Bind1(Lab,lab),  Bind2(Op,op),
      Bind1(Ref,ref), Bind3(Num ,num), Bind4( Opcode , opcode ),
```

```
        Bind4(Operand,operand ),Bind5( Mnem, mnem )],

%% Assembler
 % appended list of input, output and three context bindings
SchemaType([Bind6(IN(SeqA),seqa ), Bind7(OUT(SeqM),seqm),
        Bind1(Lab,lab),  Bind2(Op,op),
        Bind1(Ref,ref), Bind3(Num ,num), Bind4( Opcode , opcode ),
        Bind4(Operand,operand ),Bind5( Mnem, mnem )], Assembly)
         <-
          SchemaType([Bind1(Lab,lab),  Bind2(Op,op), Bind1(Ref,ref),
           Bind3(Num ,num)], Assembly_context1 ) &  % A1-A6
          SchemaType([Bind4( Opcode , opcode ), Bind4(Operand,operand )],
                                          Assembly_context2 ) &
                                                  % M1-M3
           SchemaType([Bind5( Mnem, mnem )], Assembly_context3 ) &
                                                  % M4
        Append(b1, [Bind6(IN(SeqA),seqa ), Bind7(OUT(SeqM),seqm)],
       [Bind4( Opcode , opcode ), Bind4(Operand,operand )],

                a = {x : IsA(x) } &
                m = {x : IsM(x) } &
                sym =  {x : IsSym(x) } &
                int = {x : 0 < x < 5000 } &
                IsSequ(seqa, a, n1)  &
                IsSequ(seqm, m, n2)  &
           SOME [symtab]
             (PF(symtab, sym, int) &
             Composition(seqalab, seqa, lab ) &
             Inverse(symtab, seqalab) &
             DomContents(symtab, domsymtab) &
             Composition( seqaref, seqa, ref ) &
             RanContents(seqaref, rangeseqaref)  &
             rangeseqaref  Subset domsymtab  &          % S3

             Composition(seqarefsymtab, seqaref, symtab ) &
             Composition(seqaop, seqa, op ) &
             RanContents(seqaop, rangeseqaop) &
             DomContents(mnem, dommnem ) &
             rangeseqaop Subset dommnem  &                % S4

             Composition(seqmoperand, seqm, operand ) &
             Composition(seqanum, seqa, num ) &
             seqmoperand = seqarefsymtab + seqanum &      % S5
```

```
            Composition(seqmopcode, seqm, opcode ) &
            Composition(seqaopmnem, seqaop, mnem ) &
            seqaopmnem = seqmopcode    ).                % S6

% Two phase assembler

% Intermediate state
  SchemaType([ Bind9(St, st ), Bind10(Rt, rt ),Bind7(Core, core)] , IS)

    <-     int = {x : 0 < x < 5000 } &
          sym = {x : IsSym(x) } &
          Rel( st, sym, int ) &
          PF( rt, int, sym) &
          m = {x : IsM(x) } &
          IsSequ(core, m, n).

%% Assembler Phase 1
SchemaType([Bind6(IN(SeqA),seqa ), Bind9(St, st), Bind10(Rt, rt),
        Bind7(Core, core),
 % context schemas
        Bind1(Lab,lab),  Bind2(Op,op),
        Bind1(Ref,ref), Bind3(Num ,num), Bind4( Opcode , opcode ),
        Bind4(Operand,operand ),Bind5( Mnem, mnem ) ], Phase1)
         <-
          SchemaType([Bind1(Lab,lab),  Bind2(Op,op), Bind1(Ref,ref),
           Bind3(Num ,num)], Assembly_context1 ) &  %  A1-A6
          SchemaType([Bind4( Opcode , opcode ), Bind4(Operand,operand )],
                                         Assembly_context2 ) &
                                                 %  M1-M3
           SchemaType([Bind5( Mnem, mnem )], Assembly_context3 ) &
                                                 %  M4
               a = {x : IsA(x) } &
               IsSequ(seqa, a, n1)  &

            Composition(seqalab, seqa, lab ) &
            Inverse(st, seqalab) &                    %% P1.1

            Composition( rt, seqa, ref ) &             %% P1.2

            Composition( coreoperand, core, operand ) &
            DomContents(rt, domrt)          &
            Composition(innum, seqa, num)  &
```

```
            DomExclude(innum, domrt, coreoperand)  &    % P1.3

            Composition(seqaop, seqa, op ) &
            Composition(seqaopmnem, seqaop, mnem ) &
            Composition(coreopcode, core, opcode ) &
            seqaopmnem = coreopcode     &                  % P1.4

            RanContents(seqaop, rangeseqaop) &
            DomContents(mnem, dommnem ) &
            rangeseqaop Subset dommnem .                   % P1.5

%% Assembler Phase 2
SchemaType([Bind7(OUT(SeqM),seqm ), Bind9(St, st), Bind10(Rt, rt),
        Bind7(Core, core),
 % context schemas
        Bind1(Lab,lab),  Bind2(Op,op),
        Bind1(Ref,ref), Bind3(Num ,num), Bind4( Opcode , opcode ),
        Bind4(Operand,operand ),Bind5( Mnem, mnem ) ], Phase2)
         <-
          SchemaType([Bind1(Lab,lab),  Bind2(Op,op), Bind1(Ref,ref),
           Bind3(Num ,num)], Assembly_context1 ) &  %  A1-A6
          SchemaType([Bind4( Opcode , opcode ), Bind4(Operand,operand )],
                                          Assembly_context2 ) &
                                                  %  M1-M3
           SchemaType([Bind5( Mnem, mnem )], Assembly_context3 ) &
                                                  %  M4
              m = {x : IsM(x) } &
              int = {x : 0 < x < 5000 } &
              sym = {x : IsSym(x) } &
              IsSequ(seqm, m, n1)  &
              PF(st, sym, int) &                  % P2.1

              RanContents(rt, rangert) &
              DomContents(st, domst) &
              rangert Subset domst   &                 % P2.2

              Composition(coreopcode, core, opcode ) &
              Composition(seqmopcode, seqm, opcode ) &
              seqmopcode =  coreopcode               & % P1.3

              Composition( seqmoperand, seqm, operand ) &
              Composition( coreoperand, core, operand ) &
              DomContents(rt, domrt)          &
```

```
        DomExclude(tmp1, domrt, seqmoperand)  &
      % tmp1 is seqmoperand with domain excluded domrt
      % which equals coreoperand  with domain excl domrt
         DomExclude(tmp1, domrt, coreoperand)  &    % P2.4

         Composition(rtst, rt, st ) &
       % rtst is seqmoperand with domain restricted to rtst
         DomRestrict(rtst, domrt, seqmoperand).       % P2.5


SchemaType([Bind6(IN(SeqA),seqa ), Bind7(OUT(SeqM),seqm), Bind7(Core, core),
      Bind1(Lab,lab),  Bind2(Op,op), Bind1(Ref,ref), Bind3(Num ,num),
      Bind4( Opcode , opcode ), Bind4(Operand,operand ),
         Bind5( Mnem, mnem )], Implementation )
       <-
       SchemaType([Bind6(IN(SeqA),seqa ), Bind9(St, st), Bind10(Rt, rt),
      Bind7(Core, core),
      Bind1(Lab,lab),  Bind2(Op,op),
      Bind1(Ref,ref), Bind3(Num ,num), Bind4( Opcode , opcode ),
      Bind4(Operand,operand ),Bind5( Mnem, mnem ) ], Phase1)
        &
       SchemaType([Bind7(OUT(SeqM),seqm ), Bind9(St, st), Bind10(Rt, rt),
      Bind7(Core, core), Bind1(Lab,lab),  Bind2(Op,op),
      Bind1(Ref,ref), Bind3(Num ,num), Bind4( Opcode , opcode ),
      Bind4(Operand,operand ),Bind5( Mnem, mnem ) ], Phase2).
```

Assembly Queries

This section contains full queries for the assembler.

```
% Query to SchemaType assembly context 1:

 [Assembly] <-
SchemaType([Bind1( Lab, { OrdPair ( A1, V1 ), OrdPair ( A2, V2 ),
          OrdPair ( A3, Loop ), OrdPair ( A9, Exit ) }),
         Bind2(Op, { OrdPair(A3,Load), OrdPair( A4,Subn),
          OrdPair( A5,Store),OrdPair( A6,Compare),
          OrdPair( A7,Jumple),OrdPair( A8,Jump),OrdPair( A9,Return)}),
         Bind1(Ref, {OrdPair(A3,V2),OrdPair(A5,V2),OrdPair(A6,V1),
           OrdPair(A7,Exit), OrdPair(A8,Loop)}),
         Bind3(Num, {OrdPair(A1,100),OrdPair(A2,4095),OrdPair(A4,8)})],
    Assembly_context1).

Yes.
```

% Query to SchemaType assembly context 2:

```
 [Assembly] <-

SchemaType([Bind4( Opcode ,{OrdPair(M3,1),
                OrdPair(M4,3),OrdPair(M5,2),OrdPair(M6,50),OrdPair(M7,61),
                OrdPair(M8,71),OrdPair(M9,77)}),
             Bind4(Operand,{OrdPair(M1,100),OrdPair(M2,4095),
                OrdPair(M3,2),OrdPair(M4,8),
                OrdPair(M5,2),OrdPair(M6,1),OrdPair(M7,9),OrdPair(M8,3)})],
      Assembly_context2 ).

yes.
```

% Query to SchemaType assembly context 3:

```
[Assembly] <-
SchemaType([Bind5( Mnem, {OrdPair(Load,1),OrdPair(Subn,3),OrdPair(Store,2),
                OrdPair(Compare,50), OrdPair(Jumple,61),
                OrdPair(Jump,71),OrdPair(Return,77)})],
      Assembly_context3 ).

yes
```

% Query to SchemaType assembly

```
[Assembly] <-

seqa = { OrdPair ( 1, A1 ), OrdPair ( 2, A2 ), OrdPair ( 3, A3 ),
         OrdPair ( 4, A4 ), OrdPair ( 5, A5 ), OrdPair ( 6, A6 ),
         OrdPair ( 7, A7 ), OrdPair ( 8, A8 ), OrdPair ( 9, A9 ) } &
seqm = { OrdPair ( 1, M1 ), OrdPair ( 2, M2 ), OrdPair ( 3, M3 ),
         OrdPair ( 4, M4 ), OrdPair ( 5, M5 ), OrdPair ( 6, M6 ),
         OrdPair ( 7, M7 ), OrdPair ( 8, M8 ), OrdPair ( 9, M9 ) } &
lab =  { OrdPair ( A1, V1 ), OrdPair ( A2, V2 ), OrdPair ( A3, Loop ),
         OrdPair ( A9, Exit ) } &
op =   { OrdPair ( A3, Load ), OrdPair ( A4, Subn ), OrdPair ( A5, Store ),
         OrdPair ( A6, Compare ), OrdPair ( A7, Jumple ), OrdPair ( A8, Jump ),
         OrdPair ( A9, Return ) } &
ref =  { OrdPair ( A3, V2 ), OrdPair ( A5, V2 ), OrdPair ( A6, V1 ),
         OrdPair ( A7, Exit ), OrdPair ( A8, Loop ) } &

num = {  OrdPair ( A1, 100 ), OrdPair ( A2, 4095 ), OrdPair ( A4, 8 ) } &
opcode = { OrdPair ( M3, 1 ), OrdPair ( M4, 3 ), OrdPair ( M5, 2 ),
```

```
            OrdPair ( M6, 50 ), OrdPair ( M7, 61 ), OrdPair ( M8, 71 ),
            OrdPair ( M9, 77 ) } &
operand = { OrdPair ( M1, 100 ), OrdPair ( M2, 4095 ), OrdPair ( M3, 2 ),
             OrdPair ( M4, 8 ), OrdPair ( M5, 2 ), OrdPair ( M6, 1 ),
             OrdPair ( M7, 9 ), OrdPair ( M8, 3 ) } &
mnem = { OrdPair ( Load, 1 ), OrdPair ( Subn, 3 ), OrdPair ( Store, 2 ),
          OrdPair ( Compare, 50 ), OrdPair ( Jumple, 61 ),
          OrdPair ( Jump, 71 ), OrdPair ( Return, 77 ) } &

 SchemaType([Bind6(IN(SeqA),seqa ), Bind7(OUT(SeqM),seqm), Bind1(Lab,lab),
        Bind2(Op,op),
        Bind1(Ref,ref),
        Bind3(Num ,num), Bind4( Opcode , opcode ),
        Bind4(Operand,operand ),Bind5( Mnem, mnem )], Assembly).

lab = {OrdPair(A1,V1),OrdPair(A2,V2),OrdPair(A3,Loop),OrdPair(A9,Exit)},
mnem = {OrdPair(Compare,50),OrdPair(Jump,71),OrdPair(Jumple,61),
         OrdPair(Load,1),OrdPair(Return,77),OrdPair(Store,2),
         OrdPair(Subn,3)},
num = {OrdPair(A1,100),OrdPair(A2,4095),OrdPair(A4,8)},
op = {OrdPair(A3,Load),OrdPair(A4,Subn),OrdPair(A5,Store),
       OrdPair(A6,Compare),OrdPair(A7,Jumple),OrdPair(A8,Jump),
       OrdPair(A9,Return)},
opcode = {OrdPair(M3,1),OrdPair(M4,3),OrdPair(M5,2),OrdPair(M6,50),
           OrdPair(M7,61),OrdPair(M8,71),OrdPair(M9,77)},
operand = {OrdPair(M1,100),OrdPair(M2,4095),OrdPair(M3,2),OrdPair(M4,8),
            OrdPair(M5,2),OrdPair(M6,1),OrdPair(M7,9),OrdPair(M8,3)},
ref = {OrdPair(A3,V2),OrdPair(A5,V2),OrdPair(A6,V1),OrdPair(A7,Exit),
        OrdPair(A8,Loop)},
seqa = {OrdPair(1,A1),OrdPair(2,A2),OrdPair(3,A3),OrdPair(4,A4),
         OrdPair(5,A5),OrdPair(6,A6),OrdPair(7,A7),OrdPair(8,A8),OrdPair(9,A9)},
seqm = {OrdPair(1,M1),OrdPair(2,M2),OrdPair(3,M3),OrdPair(4,M4),
         OrdPair(5,M5),OrdPair(6,M6),OrdPair(7,M7),OrdPair(8,M8),
         OrdPair(9,M9)} ?

Yes

% Query 1 to Phase1 - operand has intermediate value

 [Assembly] <-
seqa = { OrdPair ( 1, A1 ), OrdPair ( 2, A2 ), OrdPair ( 3, A3 ),
          OrdPair ( 4, A4 ), OrdPair ( 5, A5 ), OrdPair ( 6, A6 ),
          OrdPair ( 7, A7 ), OrdPair ( 8, A8 ), OrdPair ( 9, A9 ) } &
```

```
    core = { OrdPair ( 1, M1 ), OrdPair ( 2, M2 ), OrdPair ( 3, M3 ),
          OrdPair ( 4, M4 ), OrdPair ( 5, M5 ), OrdPair ( 6, M6 ),
          OrdPair ( 7, M7 ), OrdPair ( 8, M8 ), OrdPair ( 9, M9 ) } &
lab =  { OrdPair ( A1, V1 ), OrdPair ( A2, V2 ), OrdPair ( A3, Loop ),
          OrdPair ( A9, Exit ) } &
op =   { OrdPair ( A3, Load ), OrdPair ( A4, Subn ), OrdPair ( A5, Store ),
          OrdPair ( A6, Compare ), OrdPair ( A7, Jumple ), OrdPair ( A8, Jump ),
          OrdPair ( A9, Return ) } &
ref =  { OrdPair ( A3, V2 ), OrdPair ( A5, V2 ), OrdPair ( A6, V1 ),
          OrdPair ( A7, Exit ), OrdPair ( A8, Loop ) } &

num = {  OrdPair ( A1, 100 ), OrdPair ( A2, 4095 ), OrdPair ( A4, 8 ) } &
opcode = { OrdPair ( M3, 1 ), OrdPair ( M4, 3 ), OrdPair ( M5, 2 ),
            OrdPair ( M6, 50 ), OrdPair ( M7, 61 ), OrdPair ( M8, 71 ),
            OrdPair ( M9, 77 ) } &
% intermediate value
operand = { OrdPair ( M1, 100 ), OrdPair ( M2, 4095 ),  OrdPair ( M4, 8 )}  &
mnem = { OrdPair ( Load, 1 ), OrdPair ( Subn, 3 ), OrdPair ( Store, 2 ),
          OrdPair ( Compare, 50 ), OrdPair ( Jumple, 61 ),
          OrdPair ( Jump, 71 ), OrdPair ( Return, 77 ) } &

SchemaType([Bind6(IN(SeqA),seqa ), Bind9(St, st),
                Bind10(Rt, rt), Bind7(Core, core), Bind1(Lab,lab),
                Bind2(Op,op), Bind1(Ref,ref), Bind3(Num ,num),
                Bind4( Opcode , opcode ), Bind4(Operand,operand ),
                Bind5( Mnem, mnem )], Phase1).

core = {OrdPair(1,M1),OrdPair(2,M2),OrdPair(3,M3),OrdPair(4,M4),
        OrdPair(5,M5),OrdPair(6,M6),OrdPair(7,M7),OrdPair(8,M8),OrdPair(9,M9)},
lab = {OrdPair(A1,V1),OrdPair(A2,V2),OrdPair(A3,Loop),
       OrdPair(A9,Exit)},
mnem = {OrdPair(Compare,50),OrdPair(Jump,71),OrdPair(Jumple,61),
        OrdPair(Load,1),OrdPair(Return,77),OrdPair(Store,2),OrdPair(Subn,3)},
num = {OrdPair(A1,100),OrdPair(A2,4095),OrdPair(A4,8)},
op = {OrdPair(A3,Load),OrdPair(A4,Subn),OrdPair(A5,Store),
      OrdPair(A6,Compare),OrdPair(A7,Jumple),OrdPair(A8,Jump),
      OrdPair(A9,Return)},
opcode = {OrdPair(M3,1),OrdPair(M4,3),OrdPair(M5,2),OrdPair(M6,50),
          OrdPair(M7,61),OrdPair(M8,71),OrdPair(M9,77)},
operand = {OrdPair(M1,100),OrdPair(M2,4095),OrdPair(M4,8)},
ref = {OrdPair(A3,V2),OrdPair(A5,V2),OrdPair(A6,V1),OrdPair(A7,Exit),
       OrdPair(A8,Loop)},
```

```
rt = {OrdPair(3,V2),OrdPair(5,V2),OrdPair(6,V1),OrdPair(7,Exit),
      OrdPair(8,Loop)},
seqa = {OrdPair(1,A1),OrdPair(2,A2),OrdPair(3,A3),OrdPair(4,A4),
        OrdPair(5,A5),OrdPair(6,A6),OrdPair(7,A7),OrdPair(8,A8),OrdPair(9,A9)},
st = {OrdPair(Exit,9),OrdPair(Loop,3),OrdPair(V1,1),OrdPair(V2,2)} ?
```

% Query 2 to Phase1 - operand has final value

```
 [Assembly] <-
seqa = { OrdPair ( 1, A1 ), OrdPair ( 2, A2 ), OrdPair ( 3, A3 ),
         OrdPair ( 4, A4 ), OrdPair ( 5, A5 ), OrdPair ( 6, A6 ),
         OrdPair ( 7, A7 ), OrdPair ( 8, A8 ), OrdPair ( 9, A9 ) } &

core = { OrdPair ( 1, M1 ), OrdPair ( 2, M2 ), OrdPair ( 3, M3 ),
         OrdPair ( 4, M4 ), OrdPair ( 5, M5 ), OrdPair ( 6, M6 ),
         OrdPair ( 7, M7 ), OrdPair ( 8, M8 ), OrdPair ( 9, M9 ) } &
lab =  { OrdPair ( A1, V1 ), OrdPair ( A2, V2 ), OrdPair ( A3, Loop ),
         OrdPair ( A9, Exit ) } &
op =   { OrdPair ( A3, Load ), OrdPair ( A4, Subn ), OrdPair ( A5, Store ),
         OrdPair ( A6, Compare ), OrdPair ( A7, Jumple ), OrdPair ( A8, Jump ),
         OrdPair ( A9, Return ) } &
ref =  { OrdPair ( A3, V2 ), OrdPair ( A5, V2 ), OrdPair ( A6, V1 ),
         OrdPair ( A7, Exit ), OrdPair ( A8, Loop ) } &

num = {  OrdPair ( A1, 100 ), OrdPair ( A2, 4095 ), OrdPair ( A4, 8 ) } &
opcode = { OrdPair ( M3, 1 ), OrdPair ( M4, 3 ), OrdPair ( M5, 2 ),
           OrdPair ( M6, 50 ), OrdPair ( M7, 61 ), OrdPair ( M8, 71 ),
           OrdPair ( M9, 77 ) } &
operand = { OrdPair ( M1, 100 ), OrdPair ( M2, 4095 ), OrdPair ( M3, 2 ),
            OrdPair ( M4, 8 ), OrdPair ( M5, 2 ), OrdPair ( M6, 1 ),
            OrdPair ( M7, 9 ), OrdPair ( M8, 3 ) } &
mnem = { OrdPair ( Load, 1 ), OrdPair ( Subn, 3 ), OrdPair ( Store, 2 ),
         OrdPair ( Compare, 50 ), OrdPair ( Jumple, 61 ),
         OrdPair ( Jump, 71 ), OrdPair ( Return, 77 ) } &

    SchemaType([Bind6(IN(SeqA),seqa ), Bind9(St, st),
               Bind10(Rt, rt), Bind7(Core, core), Bind1(Lab,lab),
               Bind2(Op,op), Bind1(Ref,ref), Bind3(Num ,num),
               Bind4( Opcode , opcode ), Bind4(Operand,operand ),
               Bind5( Mnem, mnem )], Phase1).

core = {OrdPair(1,M1),OrdPair(2,M2),OrdPair(3,M3),OrdPair(4,M4),
```

```
        OrdPair(5,M5),OrdPair(6,M6),OrdPair(7,M7),OrdPair(8,M8),OrdPair(9,M9)},
lab = {OrdPair(A1,V1),OrdPair(A2,V2),OrdPair(A3,Loop),
       OrdPair(A9,Exit)},
mnem = {OrdPair(Compare,50),OrdPair(Jump,71),OrdPair(Jumple,61),
        OrdPair(Load,1),OrdPair(Return,77),OrdPair(Store,2),OrdPair(Subn,3)},
num = {OrdPair(A1,100),OrdPair(A2,4095),OrdPair(A4,8)},
op = {OrdPair(A3,Load),OrdPair(A4,Subn),OrdPair(A5,Store),
      OrdPair(A6,Compare),OrdPair(A7,Jumple),OrdPair(A8,Jump),
      OrdPair(A9,Return)},
opcode = {OrdPair(M3,1),OrdPair(M4,3),OrdPair(M5,2),OrdPair(M6,50),
          OrdPair(M7,61),OrdPair(M8,71),OrdPair(M9,77)},
operand = {OrdPair(M1,100),OrdPair(M2,4095),OrdPair(M3,2),
           OrdPair(M4,8),OrdPair(M5,2),OrdPair(M6,1),OrdPair(M7,9),
           OrdPair(M8,3)},
ref = {OrdPair(A3,V2),OrdPair(A5,V2),OrdPair(A6,V1),OrdPair(A7,Exit),
       OrdPair(A8,Loop)},
rt = {OrdPair(3,V2),OrdPair(5,V2),OrdPair(6,V1),OrdPair(7,Exit),
      OrdPair(8,Loop)},
seqa = {OrdPair(1,A1),OrdPair(2,A2),OrdPair(3,A3),OrdPair(4,A4),
        OrdPair(5,A5),OrdPair(6,A6),OrdPair(7,A7),OrdPair(8,A8),OrdPair(9,A9)},
st = {OrdPair(Exit,9),OrdPair(Loop,3),OrdPair(V1,1),OrdPair(V2,2)} ?


% Query to Phase2 values of rt and st are provided by Phase 1
 [Assembly] <-
    core = { OrdPair ( 1, M1 ), OrdPair ( 2, M2 ), OrdPair ( 3, M3 ),
        OrdPair ( 4, M4 ), OrdPair ( 5, M5 ), OrdPair ( 6, M6 ),
        OrdPair ( 7, M7 ), OrdPair ( 8, M8 ), OrdPair ( 9, M9 ) } &

   seqm = {OrdPair(1,M1),OrdPair(2,M2),OrdPair(3,M3),OrdPair(4,M4),
       OrdPair(5,M5),OrdPair(6,M6),OrdPair(7,M7),OrdPair(8,M8),
       OrdPair(9,M9)} &
   st = {OrdPair(Exit,9),OrdPair(Loop,3),OrdPair(V1,1),OrdPair(V2,2)} &

   rt = {OrdPair(3,V2),OrdPair(5,V2),OrdPair(6,V1),OrdPair(7,Exit),
          OrdPair(8,Loop)} &

lab =  { OrdPair ( A1, V1 ), OrdPair ( A2, V2 ), OrdPair ( A3, Loop ),
        OrdPair ( A9, Exit ) } &
op =   { OrdPair ( A3, Load ), OrdPair ( A4, Subn ), OrdPair ( A5, Store ),
        OrdPair ( A6, Compare ), OrdPair ( A7, Jumple ), OrdPair ( A8, Jump ),
        OrdPair ( A9, Return ) } &
ref =  { OrdPair ( A3, V2 ), OrdPair ( A5, V2 ), OrdPair ( A6, V1 ),
```

```
        OrdPair ( A7, Exit ), OrdPair ( A8, Loop ) } &

num = {  OrdPair ( A1, 100 ), OrdPair ( A2, 4095 ), OrdPair ( A4, 8 ) } &
opcode = { OrdPair ( M3, 1 ), OrdPair ( M4, 3 ), OrdPair ( M5, 2 ),
          OrdPair ( M6, 50 ), OrdPair ( M7, 61 ), OrdPair ( M8, 71 ),
          OrdPair ( M9, 77 ) } &
operand = { OrdPair ( M1, 100 ), OrdPair ( M2, 4095 ), OrdPair ( M3, 2 ),
           OrdPair ( M4, 8 ), OrdPair ( M5, 2 ), OrdPair ( M6, 1 ),
           OrdPair ( M7, 9 ), OrdPair ( M8, 3 ) } &
mnem = { OrdPair ( Load, 1 ), OrdPair ( Subn, 3 ), OrdPair ( Store, 2 ),
        OrdPair ( Compare, 50 ), OrdPair ( Jumple, 61 ),
        OrdPair ( Jump, 71 ), OrdPair ( Return, 77 ) } &

SchemaType([Bind7(OUT(SeqM),seqm ), Bind9(St, st), Bind10(Rt, rt),
       Bind7(Core, core),   Bind1(Lab,lab),  Bind2(Op,op),
       Bind1(Ref,ref), Bind3(Num ,num), Bind4( Opcode , opcode ),
       Bind4(Operand,operand ),Bind5( Mnem, mnem ) ], Phase2).

core = {OrdPair(1,M1),OrdPair(2,M2),OrdPair(3,M3),OrdPair(4,M4),
       OrdPair(5,M5),OrdPair(6,M6),OrdPair(7,M7),OrdPair(8,M8),
       OrdPair(9,M9)},
lab = {OrdPair(A1,V1),OrdPair(A2,V2),OrdPair(A3,Loop),OrdPair(A9,Exit)},
mnem = {OrdPair(Compare,50),OrdPair(Jump,71),OrdPair(Jumple,61),
       OrdPair(Load,1),OrdPair(Return,77),OrdPair(Store,2),OrdPair(Subn,3)},
num = {OrdPair(A1,100),OrdPair(A2,4095),OrdPair(A4,8)},
op = {OrdPair(A3,Load),OrdPair(A4,Subn),OrdPair(A5,Store),
     OrdPair(A6,Compare),OrdPair(A7,Jumple),OrdPair(A8,Jump),
     OrdPair(A9,Return)},
opcode = {OrdPair(M3,1),OrdPair(M4,3),OrdPair(M5,2),OrdPair(M6,50),
         OrdPair(M7,61),OrdPair(M8,71),OrdPair(M9,77)},
operand = {OrdPair(M1,100),OrdPair(M2,4095),OrdPair(M3,2),
          OrdPair(M4,8),OrdPair(M5,2),OrdPair(M6,1),OrdPair(M7,9),
          OrdPair(M8,3)},
ref = {OrdPair(A3,V2),OrdPair(A5,V2),OrdPair(A6,V1),OrdPair(A7,Exit),
      OrdPair(A8,Loop)},
rt = {OrdPair(3,V2),OrdPair(5,V2),OrdPair(6,V1),OrdPair(7,Exit),
     OrdPair(8,Loop)},
seqm = {OrdPair(1,M1),OrdPair(2,M2),OrdPair(3,M3),OrdPair(4,M4),
       OrdPair(5,M5),OrdPair(6,M6),OrdPair(7,M7),OrdPair(8,M8),OrdPair(9,M9)},
st = {OrdPair(Exit,9),OrdPair(Loop,3),OrdPair(V1,1),OrdPair(V2,2)} ?


% Query to Implementation with final operand
```

```
 [Assembly] <–
seqa = { OrdPair ( 1, A1 ), OrdPair ( 2, A2 ), OrdPair ( 3, A3 ),
         OrdPair ( 4, A4 ), OrdPair ( 5, A5 ), OrdPair ( 6, A6 ),
         OrdPair ( 7, A7 ), OrdPair ( 8, A8 ), OrdPair ( 9, A9 ) } &
seqm = { OrdPair ( 1, M1 ), OrdPair ( 2, M2 ), OrdPair ( 3, M3 ),
         OrdPair ( 4, M4 ), OrdPair ( 5, M5 ), OrdPair ( 6, M6 ),
         OrdPair ( 7, M7 ), OrdPair ( 8, M8 ), OrdPair ( 9, M9 ) } &
core = {OrdPair(1,M1),OrdPair(2,M2),OrdPair(3,M3),OrdPair(4,M4),
        OrdPair(5,M5),OrdPair(6,M6),OrdPair(7,M7),OrdPair(8,M8),
        OrdPair(9,M9)} &
lab =  { OrdPair ( A1, V1 ), OrdPair ( A2, V2 ), OrdPair ( A3, Loop ),
         OrdPair ( A9, Exit ) } &
op =   { OrdPair ( A3, Load ), OrdPair ( A4, Subn ), OrdPair ( A5, Store ),
         OrdPair ( A6, Compare ), OrdPair ( A7, Jumple ), OrdPair ( A8, Jump ),
         OrdPair ( A9, Return ) } &
ref =  { OrdPair ( A3, V2 ), OrdPair ( A5, V2 ), OrdPair ( A6, V1 ),
         OrdPair ( A7, Exit ), OrdPair ( A8, Loop ) } &

num = {  OrdPair ( A1, 100 ), OrdPair ( A2, 4095 ), OrdPair ( A4, 8 ) } &
opcode = { OrdPair ( M3, 1 ), OrdPair ( M4, 3 ), OrdPair ( M5, 2 ),
           OrdPair ( M6, 50 ), OrdPair ( M7, 61 ), OrdPair ( M8, 71 ),
           OrdPair ( M9, 77 ) } &
operand = { OrdPair ( M1, 100 ), OrdPair ( M2, 4095 ), OrdPair ( M3, 2 ),
            OrdPair ( M4, 8 ), OrdPair ( M5, 2 ), OrdPair ( M6, 1 ),
            OrdPair ( M7, 9 ), OrdPair ( M8, 3 ) } &
mnem = { OrdPair ( Load, 1 ), OrdPair ( Subn, 3 ), OrdPair ( Store, 2 ),
         OrdPair ( Compare, 50 ), OrdPair ( Jumple, 61 ),
         OrdPair ( Jump, 71 ), OrdPair ( Return, 77 ) } &

         SchemaType([Bind6(IN(SeqA),seqa ), Bind7(OUT(SeqM),seqm),
         Bind7(Core, core),
        Bind1(Lab,lab),  Bind2(Op,op), Bind1(Ref,ref), Bind3(Num ,num),
        Bind4( Opcode , opcode ), Bind4(Operand,operand ),
            Bind5( Mnem, mnem )], Implementation) .

core = {OrdPair(1,M1),OrdPair(2,M2),OrdPair(3,M3),OrdPair(4,M4),
        OrdPair(5,M5),OrdPair(6,M6),OrdPair(7,M7),OrdPair(8,M8),
        OrdPair(9,M9)},
lab = {OrdPair(A1,V1),OrdPair(A2,V2),OrdPair(A3,Loop),OrdPair(A9,Exit)},
mnem = {OrdPair(Compare,50),OrdPair(Jump,71),OrdPair(Jumple,61),
        OrdPair(Load,1),OrdPair(Return,77),OrdPair(Store,2),OrdPair(Subn,3)},
num = {OrdPair(A1,100),OrdPair(A2,4095),OrdPair(A4,8)},
```

```
op = {OrdPair(A3,Load),OrdPair(A4,Subn),OrdPair(A5,Store),
      OrdPair(A6,Compare),OrdPair(A7,Jumple),OrdPair(A8,Jump),OrdPair(A9,Return)},
opcode = {OrdPair(M3,1),OrdPair(M4,3),OrdPair(M5,2),OrdPair(M6,50),
          OrdPair(M7,61),OrdPair(M8,71),OrdPair(M9,77)},
operand = {OrdPair(M1,100),OrdPair(M2,4095),OrdPair(M3,2),
           OrdPair(M4,8),OrdPair(M5,2),OrdPair(M6,1),OrdPair(M7,9),OrdPair(M8,3)},
ref = {OrdPair(A3,V2),OrdPair(A5,V2),OrdPair(A6,V1),OrdPair(A7,Exit),
       OrdPair(A8,Loop)},
seqa = {OrdPair(1,A1),OrdPair(2,A2),OrdPair(3,A3),OrdPair(4,A4),
        OrdPair(5,A5),OrdPair(6,A6),OrdPair(7,A7),OrdPair(8,A8),OrdPair(9,A9)},
seqm = {OrdPair(1,M1),OrdPair(2,M2),OrdPair(3,M3),OrdPair(4,M4),
        OrdPair(5,M5),OrdPair(6,M6),OrdPair(7,M7),OrdPair(8,M8),OrdPair(9,M9)} ?

Yes

% Query to Implementation with intermediate operand
[Assembly] <— seqa = { OrdPair ( 1, A1 ), OrdPair ( 2, A2 ),
            OrdPair ( 3, A3 ),
         OrdPair ( 4, A4 ), OrdPair ( 5, A5 ), OrdPair ( 6, A6 ),
         OrdPair ( 7, A7 ), OrdPair ( 8, A8 ), OrdPair ( 9, A9 ) } &
seqm = { OrdPair ( 1, M1 ), OrdPair ( 2, M2 ), OrdPair ( 3, M3 ),
         OrdPair ( 4, M4 ), OrdPair ( 5, M5 ), OrdPair ( 6, M6 ),
         OrdPair ( 7, M7 ), OrdPair ( 8, M8 ), OrdPair ( 9, M9 ) } &
core = {OrdPair(1,M1),OrdPair(2,M2),OrdPair(3,M3),OrdPair(4,M4),
        OrdPair(5,M5),OrdPair(6,M6),OrdPair(7,M7),OrdPair(8,M8),
        OrdPair(9,M9)} &
lab =  { OrdPair ( A1, V1 ), OrdPair ( A2, V2 ), OrdPair ( A3, Loop ),
         OrdPair ( A9, Exit ) } &
op =   { OrdPair ( A3, Load ), OrdPair ( A4, Subn ), OrdPair ( A5, Store ),
         OrdPair ( A6, Compare ), OrdPair ( A7, Jumple ), OrdPair ( A8, Jump ),
         OrdPair ( A9, Return ) } &
ref =  { OrdPair ( A3, V2 ), OrdPair ( A5, V2 ), OrdPair ( A6, V1 ),
         OrdPair ( A7, Exit ), OrdPair ( A8, Loop ) } &

num = {  OrdPair ( A1, 100 ), OrdPair ( A2, 4095 ), OrdPair ( A4, 8 ) } &
opcode = { OrdPair ( M3, 1 ), OrdPair ( M4, 3 ), OrdPair ( M5, 2 ),
           OrdPair ( M6, 50 ), OrdPair ( M7, 61 ), OrdPair ( M8, 71 ),
           OrdPair ( M9, 77 ) } &
operand = { OrdPair ( M1, 100 ), OrdPair ( M2, 4095 ), OrdPair ( M4, 3 ) } &

mnem = { OrdPair ( Load, 1 ), OrdPair ( Subn, 3 ), OrdPair ( Store, 2 ),
         OrdPair ( Compare, 50 ), OrdPair ( Jumple, 61 ),
         OrdPair ( Jump, 71 ), OrdPair ( Return, 77 ) } &
```

```
SchemaType([Bind6(IN(SeqA),seqa ), Bind7(OUT(SeqM),seqm),
       Bind7(Core, core),
      Bind1(Lab,lab),  Bind2(Op,op), Bind1(Ref,ref), Bind3(Num ,num),
      Bind4( Opcode , opcode ), Bind4(Operand,operand ),
          Bind5( Mnem, mnem )], Implementation) .
```

No
% intermediate value is not acceptable

[Xassembly] <–

C.1.4 Unix File Code

This section contains unix file system code.

Unix File: EXPORT

```
EXPORT  UnixFiles.
IMPORT  Lib.
BASE Typevars, SetNames, SchNames, BindVar, FileId, Cid.

%%CONSTRUCTOR BD/2.

FUNCTION    Bind1 : SetNames * FileId -> BindVar.
FUNCTION    Bind2 : SetNames * Integer -> BindVar.
FUNCTION    Bind3 : SetNames * Cid -> BindVar.
FUNCTION    Bind4 : SetNames * Set(OP(Cid, List(BindVar)))
                  -> BindVar.


% Decorations
%  on set names, ie priming, imput, output
FUNCTION    DSet : SetNames  -> SetNames.
FUNCTION    OUT : SetNames  -> SetNames.
FUNCTION    IN : SetNames  -> SetNames.
% on Schema names, ie priming, del
FUNCTION    DSch : SchNames  -> SchNames.
FUNCTION    Del : SchNames  -> SchNames.

PREDICATE SchemaType: List (BindVar ) * SchNames.

PREDICATE SThetaS: Set(List(BindVar)) * SchNames.

PREDICATE IsFileId : FileId.

PREDICATE IsCid : Cid.
```

Unix File: LOCAL

```
LOCAL UnixFiles.
CONSTANT     F1, F2, F3, F5, F7 : FileId;
             Cid1, Cid2 , Cid3 : Cid;
             Fid,  Posn, Cstore, ChId  : SetNames;
             CloseCs,  OpenCs, Chan, Cs  : SchNames.

%Data%

IsFileId(F1). IsFileId(F2). IsFileId(F3). IsFileId(F5). IsFileId(F7).
IsCid(Cid1). IsCid(Cid2). IsCid(Cid3).

%Schemas%


SchemaType(binding, Chan) <-
      binding = [Bind1(Fid, f), Bind2(Posn,posn) ] &
     fid = {x :  IsFileId(x) } &
     posfile = {x : 0 =< x < 100} &
% Types of variables
                 posn >= 0 &
                 posn In posfile &
                 f In fid .


SchemaType(binding, DSch( Chan)) <-
    binding = [Bind1(DSet(Fid), f), Bind2(DSet(Posn),posn)] &
% Given sets
        fid = {x :  IsFileId(x) } &
        posfile = {x : 0 =< x < 100} &

% Types of variables
                 posn >= 0 &
                 posn In posfile &
                 f In fid .

SThetaS(val, schname) <-
     val = {binding : SchemaType(binding, schname)}.

SchemaType(binding, Del(Chan)) <-
SchemaType(b1,  Chan) &
```

```
SchemaType(b2, DSch( Chan)) &
          Append(b1, b2, binding) &
          b1 = [Bind1(Fid,f),_] &
          b2 = [Bind1(DSet(Fid),f),_] .

SchemaType(binding, Cs) <-
        binding = [Bind4(Cstore, cs)] &

        cid = {x : IsCid(x) } &
         PF(cs, cid, schtyp) &
         SThetaS(schtyp, Chan).

SchemaType(binding, DSch( Cs) ) <-
        binding = [Bind4(DSet(Cstore), cs1)] &
          cid = {x : IsCid(x) } &

         PF(cs1, cid, schtyp) &
         SThetaS(schtyp, Chan).
% Query:

SchemaType(binding, Del(Cs)) <-
         SchemaType(b1, Cs) &
         SchemaType(b2, DSch( Cs)) &
          Append(b1, b2, binding) .

% Bind1, Bind2 are from delaration of CHAN
% Bind4 vars are from decaration of Cs and Cs'

SchemaType([Bind1(Fid, f), Bind2(Posn,posn), Bind3(OUT( ChId), outc),
  Bind4(Cstore, cs), Bind4(DSet(Cstore), cs1) ], OpenCs )  <-

     SchemaType([Bind1(Fid, f), Bind2(Posn,posn)], Chan) &
     SchemaType([Bind4(Cstore, cs),Bind4(DSet(Cstore), cs1)], Del(Cs)) &
          cid = {x : IsCid(x) } &
          posfile = {x : 0 =< x < 100} &
          outc In cid &
          DomContents(cs, domcs) &
          ~(outc In domcs) &
           posn In posfile &
           posn = 0 &
 % Old channel store is updated by addition of a new
% channel whose file position is zero, but FileId is unconstrained
 FunOveride(cs1, cs, {OrdPair(outc, [Bind1(Fid, f), Bind2(Posn,posn)])} ).
```

```
SchemaType([Bind3(IN( ChId), inc),
  Bind4(Cstore, cs), Bind4(DSet(Cstore), cs1) ],CloseCs ) <-
       SchemaType([Bind4(Cstore, cs),Bind4(DSet(Cstore), cs1)], Del(Cs)) &
         cid = {x :  IsCid(x) } &
         inc  In cid &
         DomContents(cs, domcs) &
         inc In domcs  &
         DomExclude(cs1, {inc}, cs).
```

Unix File Queries

This section contains queries for the unix file system.

```
% A query  to Cs where Fid is F1 and Posn is 2.
% There are no other values which satisfy the predicate.
[UnixFiles] <-
SchemaType ( binding, Cs ) &
     binding = [ Bind4 ( Cstore, { OrdPair ( Cid1, [ Bind1 ( Fid, F1 ),
                                   Bind2 ( Posn, 2 ) ])})].

binding = [Bind4(Cstore,{OrdPair(Cid1,[Bind1(Fid,F1),Bind2(Posn,2)])})] ? ;
No

                          % A query  to OpenCS.
% A new channel is added. This can be any of the remaining (unopened) ones and
% the file identifier can be any. The position in the file is 0.

[UnixFiles] <- cs = {OrdPair(Cid1,[Bind1(Fid,F1),Bind2(Posn,2)])} &
                 SchemaType([b1,b2,b3,Bind4(Cstore, cs),
                  Bind4(DSet(Cstore), cs1) ], OpenCs ).

b1 = Bind1(Fid,F1),
b2 = Bind2(Posn,0),
b3 = Bind3(OUT(ChId),Cid2),
cs = {OrdPair(Cid1,[Bind1(Fid,F1),Bind2(Posn,2)])},
cs1 = {OrdPair(Cid1,[Bind1(Fid,F1),Bind2(Posn,2)]),OrdPair(Cid2,
      [Bind1(Fid,F1),Bind2(Posn,0)])} ? ;

b1 = Bind1(Fid,F1),
b2 = Bind2(Posn,0),
b3 = Bind3(OUT(ChId),Cid3),
cs = {OrdPair(Cid1,[Bind1(Fid,F1),Bind2(Posn,2)])},
```

```
cs1 = {OrdPair(Cid1,[Bind1(Fid,F1),Bind2(Posn,2)]),OrdPair(Cid3,
        [Bind1(Fid,F1),Bind2(Posn,0)])} ? ;

b1 = Bind1(Fid,F2),
b2 = Bind2(Posn,0),
b3 = Bind3(OUT(ChId),Cid2),
cs = {OrdPair(Cid1,[Bind1(Fid,F1),Bind2(Posn,2)])},
cs1 = {OrdPair(Cid1,[Bind1(Fid,F1),Bind2(Posn,2)]),OrdPair(Cid2,
        [Bind1(Fid,F2),Bind2(Posn,0)])} ? ;

b1 = Bind1(Fid,F2),
b2 = Bind2(Posn,0),
b3 = Bind3(OUT(ChId),Cid3),
cs = {OrdPair(Cid1,[Bind1(Fid,F1),Bind2(Posn,2)])},
cs1 = {OrdPair(Cid1,[Bind1(Fid,F1),Bind2(Posn,2)]),OrdPair(Cid3,
        [Bind1(Fid,F2),Bind2(Posn,0)])} ? ;

b1 = Bind1(Fid,F3),
b2 = Bind2(Posn,0),
b3 = Bind3(OUT(ChId),Cid2),
cs = {OrdPair(Cid1,[Bind1(Fid,F1),Bind2(Posn,2)])},
cs1 = {OrdPair(Cid1,[Bind1(Fid,F1),Bind2(Posn,2)]),OrdPair(Cid2,
        [Bind1(Fid,F3),Bind2(Posn,0)])} ?
Yes

% A query to CloseCs
% There are no other values of channel to close.
% Attempting to close Cid3 will not succeed.

[UnixFiles] <- cs = {OrdPair(Cid1,[Bind1(Fid,F1),Bind2(Posn,2)]),
                     OrdPair(Cid2,[Bind1(Fid,F1),Bind2(Posn,5)])} &
                SchemaType ([Bind3(IN( ChId ),inc),Bind4(Cstore, cs ),
                 Bind4(DSet ( Cstore ), cs1)], CloseCs).

cs = {OrdPair(Cid1,[Bind1(Fid,F1),Bind2(Posn,2)]),OrdPair(Cid2,
        [Bind1(Fid,F1),Bind2(Posn,5)])},
cs1 = {OrdPair(Cid2,[Bind1(Fid,F1),Bind2(Posn,5)])},
inc = Cid1 ? ;

cs = {OrdPair(Cid1,[Bind1(Fid,F1),Bind2(Posn,2)]),OrdPair(Cid2,
        [Bind1(Fid,F1),Bind2(Posn,5)])},
cs1 = {OrdPair(Cid1,[Bind1(Fid,F1),Bind2(Posn,2)])},
inc = Cid2 ? ;
```

No

```
[UnixFiles] <- cs = {OrdPair(Cid1,[Bind1(Fid,F1),Bind2(Posn,2)]),

            OrdPair(Cid2,[Bind1(Fid,F1),Bind2(Posn,5)])} &
        SchemaType( [Bind3(IN( ChId ),Cid3),Bind4(Cstore, cs),
            Bind4(DSet( Cstore ), cs1)], CloseCs).
```

No

Appendix D

Proofs: Abstract Approximation

D.1 Induction Process

The base types for the induction are (i) integers, (ii) sets of integers, (iii) given sets and their instantiated elements and (iv) variables, so the first task is to show how their interpretation in the LP underestimates the interpretation in Z. Induction is over each Z construct and includes

1. Numeric expressions
2. Set expressions (union and distributed union)
3. Predicate expressions: infix
4. Set comprehension and variable declarations
5. Predicates: quantified expressions (which depend on declarations)
6. Schemas and Schema Expressions

D.2 Base Types

(i) Integers
The assumption is that there are largest positive and negative integers available in

the system, *MaxInt*, *MinInt*, which cannot be exceeded. Any attempt to do so may cause the computation to terminate. Thus for $m \in \mathbb{Z}$:

$$\mathcal{E}_{LP}[\![\mathrm{m}]\!]\rho_{LP} = m = \mathcal{E}_{Z}[\![m]\!]\rho_{Z}, \quad -\ MinInt \leq m \leq MaxInt$$
$$\mathcal{E}_{LP}[\![\mathrm{m}]\!]\rho_{LP} = \bot, m < -MinInt \text{ or } m > MaxInt$$

($\bot$ may be implemented by the output of an error message, or alternatively to the character ∞. The latter is suggested by the IEEE floating point standard.) Thus since $\gamma(\bot) = \bot$:

$$\gamma(\mathcal{E}_{LP}[\![m]\!]\rho_{LP}) \sqsubseteq \mathcal{E}_{Z}[\![m]\!]\rho_{Z}, m \in \mathbb{Z}$$

(ii) Sets of integers s

Suppose s is a subset of $\{i : \mathbb{N} \mid -MinInt \leq i \leq MaxInt\}$ and assuming that the memory bounds are not exceeded, then the abstract interpretation is *exact*.

Where *MinInt*, *MaxInt* are exceeded, (for example $s = \mathbb{Z}$) then s is interpreted as $\bot \circ \bot$ in the LP, and therefore underestimates its interpretation in the Z domain.

$$\gamma(\mathcal{E}_{LP}[\![\{i : -MaxInt \leq i \leq MaxInt\}_{\cup\bot}]\!]\rho_{LP}) = \gamma(\bot \circ \bot) = \varnothing_{\cup\bot} \sqsubseteq \mathcal{E}_{Z}[\![\mathbb{Z}]\!]\rho_{Z},$$

(iii) Given sets and their instantiated elements

Suppose G, g is a given set and typical element. The are interpreted in the LP by base type G, associated constant g and predicate IsG. In each case the abstract interpretation is exact for:

$$\gamma(\mathcal{E}_{LP}[\![g]\!]\rho_{LP}) = g$$
$$\gamma(\mathcal{E}_{LP}[\![G]\!]\rho_{LP}) = \gamma(\{x : IsG(x)\}) = G$$

(iv) Variables

The value of a variable can be obtained as a 'lookup' in the environment, where *Env* interprets the LP environment as in Andrews [6]. Assuming that the variable has a defined value, the trivial interpretation in the LP is:

$$(x_i = a_i) \longleftarrow Env_{LP}$$

which can be denoted:

$$\mathcal{E}_{LP}[\![x_i]\!]\rho_{LP} = a_i \Leftrightarrow$$
$$\qquad (x_i = a_i)\ \&\ true \quad (a_i \neq \bot).$$

If the variable is undefined because of finite failure, the answer returned is false:

$$\mathcal{E}_{LP}[\![x_i]\!]\rho_{LP} = \bot \Leftrightarrow (a_i = \bot) \ \& \ \mathit{false}.$$

If the variable is associated with a schema binding, there are *no* values which satisfy the instantiated variables and the schema predicate, so no answer substitutions. For further discussion see Section D.7.

If the variable is undefined because of non-termination or floundering, the answer returned is $\bot$:

$$\mathcal{E}_{LP}[\![x_i]\!]\rho_{LP} = \bot \Leftrightarrow (a_i = \bot) \ \& \ \bot.$$

In either case this is an exact approximation of the interpretation in D_Z, since $\rho_Z = \gamma \circ \rho_{LP}$.

D.3 Numerical and Set Expressions

Provided *MaxInt*, *MinInt* are not exceeded (as in Section D.2) the evaluation is via the Peano rules of arithmetic, as in the concrete domain, otherwise the expression evaluates to $\bot$. Thus if *fx* is a numerical expression, evaluating to m:

$$\mathcal{E}_{LP}[\![fx]\!]\rho_{LP} = m = \mathcal{E}_Z[\![fx]\!]\rho_Z, - \mathit{MinInt} \leq m \leq \mathit{MaxInt}$$
$$\mathcal{E}_{LP}[\![fx]\!]\rho_{LP} = \bot; \ \mathcal{E}_Z[\![fx]\!]\rho_Z = m, m > \mathit{MaxInt} \text{ or } m < - \mathit{MinInt}$$

Thus the abstract evaluation underestimates the concrete and:

$$\gamma(\mathcal{E}_{LP}[\![fx]\!]\rho_{LP}) \sqsubseteq \mathcal{E}_Z[\![fx]\!]\rho_Z,$$

We next apply the rules to set expressions, beginning with set union[1]. Set operators such as intersection and power set are a special case of set comprehensions and will be treated in Section D.5.

D.3.1 Set Union

Consider the syntactic expression '$x_1 \cup x_2$' which is interpreted via an equivalent 'term' in the LP, denoted '$x_1 \cup_{LP} x_2$'. (Recall that in Gödel, 'union' is provided by

[1] The reason for following this route rather than commencing from $\bigcup$ and specialising is that $\cup$ is easier to demonstrate.

a function '+'.)

Suppose sets are complete and with complete elements:

$$x_1 \mapsto a_1, x_2 \mapsto a_2 \in \rho_{LP}.$$

The expression $x_1 \cup_{LP} x_2$ is evaluated using the LP ground substitution $\{x_1/a_1, x_2/a_2\}$ so that $(x_1 \cup_{LP} x_2)\{x_1/a_1, x_2/a_2\}$ evaluates to $(a_1 \cup_{LP} a_2)$. We assume that $\cup_{LP}$ is set-theoretic and implements $\cup$ for finite sets in the same manner as $\cup$ for ZF. (See Appendices B, C.)

Condition 1 becomes:

$$f_{LP}(\mathcal{E}_{LP}[\![(x_1, x_2)]\!]\rho_{LP}) = a_1 \cup_{LP} a_2 = \mathcal{E}_{LP}[\![x_1 \cup x_2]\!]\rho_{LP}.$$

which will hold for set operations for terminating computations. That is the interpretation of ZF operations on sets is built recursively.

Condition 2

If x_1, x_2 are complete sets, $\gamma(x_1, x_2)$ in D_Z evaluates in the expected way to $(\gamma(a_1), \gamma(a_2))$ and

$$f_Z(\mathcal{E}_Z[\![(x_1, x_2)]\!]\rho_Z) = \gamma(a_1) \cup_Z \gamma(a_2) = \mathcal{E}_Z[\![x_1 \cup x_2]\!]\rho_Z.$$

Since $\cup_{LP}$ is set-theoretic then $\gamma(a_1 \cup_{LP} a_2) = \gamma(a_1) \cup_Z \gamma(a_2)$ and **Condition 3** becomes:

$$\begin{aligned}
&\gamma\ (f_{LP}\ (\mathcal{E}_{LP}[\![(x_1, x_2)]\!]\rho_{LP})) \\
&\qquad = \gamma(a_1 \cup_{LP} a_2) = \gamma(a_1) \cup_Z \gamma(a_2) = f_Z(\gamma(a_1, (a_2))) = \\
&\qquad f_Z(\gamma(\mathcal{E}_{LP}[\![(x_1, x_2)]\!]\rho_{LP})).
\end{aligned}$$

In other words the computation is exact for terminating computations. There are two ways of extending the result to non terminating computations.

1. Provide an extension of union to incomplete sets and use **AR2** as the proof rule. If

   $$x_1 = a_{\cup\perp}, x_2 = b,$$

   we define the extension for union:

   $$(a_{\cup\perp} \cup_Z b) = (a \cup_Z b_{\cup\perp}) = (a \cup_Z b)_{\cup\perp}$$

which is pointwise monotonic in the Z domain for non-standard sets. **Conditions 1–3** thus hold when 'f' is '$\cup$', thus: since **AR2** holds for x_1, x_2, then **AR2** holds for $\epsilon = x_1 \cup x_2$, where $\cup$ is pointwise monotonic for both standard and non-standard sets.

2. Interpret figure 5.4 directly and we see that in D_{LP}, if x_1 is not a standard finite set it can only have the value $x_1 = \perp \circ \perp$ and the left hand side of **AR1** for $\epsilon = x_1 \cup x_2$ is

$$\gamma\ ((\perp \circ \perp) \cup_{LP} b) = \gamma\ (\perp \circ \perp) = \varnothing_{\cup\perp},$$

since all set terms in the LP involving $\perp \circ \perp$ evaluate to $\perp \circ \perp$. Then since

$$x_1 \mapsto \varnothing_{\cup\perp}, x_2 \mapsto \gamma(b)$$

are both members of environment $\rho_Z (= \gamma \circ \rho_{LP})$, the right hand side of the ordering relationship becomes:

$$\varnothing_{\cup\perp} \cup_Z \gamma(b)$$

which will, in any case always exceed $\varnothing_{\cup\perp}$, whatever its value, provided that it is still type correct. Since we have established conditions (1 – 3) for complete sets and **AR1** *directly* for incomplete or infinite sets, then **AR1** holds when 'f' is '$\cup$'.

D.3.2 Distributed Union

We denote distributed union in the LP by $\bigcup_{LP}$ and it is defined so that it implements $\bigcup$ for finite sets in the same manner as $\bigcup$ for ZF. The argument that the LP interpretation underestimates the Z interpretation follows in a similar fashion to the argument for $\cup$.

Consider, first, the evaluation of $\bigcup x$ where $x = \{a_1 \ldots a_n\}$ is a complete, finite set in the LP environment. Then x is $\gamma(\{a_1 \ldots a_n\}) = \{\gamma(a_1) \ldots \gamma(a_n)\}$ in the Z environment.

$$f_{LP}(\mathcal{E}_{LP}[\![(x)]\!]\rho_{LP}) = \bigcup\nolimits_{LP}(\{a_1 \ldots a_n\}) = \mathcal{E}_{LP}[\![\bigcup x]\!]\rho_{LP} = \mathcal{E}_{LP}[\![fx]\!]\rho_{LP}.$$

Thus **Condition 1** will hold for set operations for terminating computations.
Condition 2

If x is complete and involves only complete sets, the interpretation in D_Z evaluates in the expected way, $\gamma(x) = \{\gamma(a_1)\ldots\gamma(a_n)\}$ and we have

$$f_Z(\mathcal{E}_Z[\![(x)]\!]\rho_Z) = \bigcup_Z\{\gamma(a_1)\ldots\gamma(a_n)\} = \bigcup_Z \gamma(\{a_1\ldots a_n\}) = \mathcal{E}_Z[\![\bigcup x]\!]\rho_Z.$$

Since $\bigcup_{LP}$ is set-theoretic then: $\gamma(\bigcup_{LP}(\{a_1\ldots a_n\}) = \bigcup_Z(\{\gamma(a_1)\ldots\gamma(a_n)\})$ and **Condition 3** becomes:

$$\begin{aligned}&\gamma\ (f_{LP}\ (\mathcal{E}_{LP}[\![(x)]\!]\rho_{LP}))\\ &\quad= \gamma(\bigcup_{LP}\{a_1\ldots a_n\}) = \bigcup_Z\{\gamma(a_1)\ldots\gamma(a_n)\} = f_Z(\gamma(\mathcal{E}_{LP}[\![(x)]\!]\rho_{LP})).\end{aligned}$$

Since it is equivalent to the set-theoretic definition then $\bigcup_{LP}$ approximates exactly in its interpretation of $\bigcup x$ in the case where x is complete. Thus **AR1** is true for complete sets.

In [17], the operation $\bigcup_Z$ is extended to incomplete sets or sets containing incomplete sets so

$$\begin{aligned}&\bigcup_Z\{u_{\cup\perp}, v, w\} = \bigcup_Z\{u, v, w\}_{\cup\perp},\\ &\bigcup_Z(t_{\cup\perp}) = (\bigcup_Z t)_{\cup\perp}\end{aligned}$$

and is thus monotonic.

The case for *incomplete sets* follows for $\bigcup_{LP} x = \perp\circ\perp$ for x non-standard for the LP and we use the direct method rather than **AR1**. Evaluating left and right hand sides $\bigcup_{LP} x$, where x is incomplete, contains incomplete elements or is infinite. Thus if $(x \mapsto \perp\circ\perp) \in \rho_{LP}$, then

$$\begin{aligned}\text{LHS } &= \gamma(\mathcal{E}_{LP}[\![\bigcup x]\!]\rho_{LP})\\ &= \gamma(\bigcup_{LP} x) = \gamma(\perp\circ\perp) = \varnothing_{\cup\perp}\\ \text{RHS } &= \mathcal{E}_Z[\![\bigcup x]\!](\gamma\circ\rho_{LP})\\ &= \bigcup_Z \varnothing_{\cup\perp}\end{aligned}$$

and the LP interpretation underestimates whatever the value of the right hand side of the order relationship **AR1**.

Thus $\bigcup$ is interpreted exactly for complete sets and underestimates where sets involved are infinite or non-standard.

D.4 Predicate Expressions

The evaluator $\mathcal{P}_{LP}$ interprets syntactic predicates p in the LP domain in the manner expected, where a predicate evaluates to $\perp$ when a program flounders or fails to terminate during its evaluation. Thus if

$$
\begin{aligned}
&Bool_Z = \{tt, ff, \perp\}\\
&Bool_{LP} = \{true, false, \perp\}
\end{aligned}
$$

then

$$\gamma(true) = tt, \gamma(false) = ff, \gamma(\perp) = \perp.$$

We also have:

$$
\begin{aligned}
&\mathcal{P}_{LP}[\![P_1 \wedge P_2]\!]\rho_{LP} = (\mathcal{P}_{LP}[\![P_1]\!]\rho_{LP} \ \& \ \mathcal{P}_{LP}[\![P_2]\!]\rho_{LP} = true) \Leftrightarrow\\
&\qquad (\ (\mathcal{P}_{LP}[\![P_1]\!]\rho_{LP} = true) \ \& \ (\mathcal{P}_{LP}[\![P_2]\!]\rho_{LP} = true)\ )
\end{aligned}
$$

The approximation requirement, **AR1** for predicates becomes:

$$\gamma(\mathcal{P}_{LP}[\![\epsilon]\!]\rho_{LP}) \sqsubseteq \mathcal{P}_Z[\![\epsilon]\!](\gamma \circ \rho_{LP}).$$

Examples of predicates to be interpreted are infix predicates $=$, $\subseteq$ and $\in$. Quantification predicates $\forall\, D \mid p \bullet q$ and $\exists\, D \mid p \bullet q$, where D is a declaration will be treated later, after we have covered declarations.

In an LP, infix predicates $p \in \Sigma_2$ of the form $p(x_1, x_2),\ x_1, x_2 \in \Sigma_1$ are interpreted in such a way that they potentially provide *enhancements* to the existing environment as well as evaluating to boolean values. There are three constraint properties associated with predicate evaluation. Suppose $\mathcal{I}$ is an infix predicate, standing for equality, subset or membership. Then if either (or both) x_1 or x_2 is undefined or only partially defined they can become ground through resolution. We call this property: **Constraint Property 1**:

$$\mathcal{P}_{LP}[\![x_1 \mathcal{I} x_2]\!]\rho_{LP} = \mathcal{P}_{LP}[\![x_1 \mathcal{I} x_2]\!]\rho'_{LP} = true$$

where $\rho'_{LP} = \rho_{LP} \oplus \{x_1 \mapsto a_1, x_2 \mapsto a_2\}$.
The environments of predicates conjoined to the infix predicates are also enhanced:

Constraint Property 2:

$$\mathcal{P}_{LP}[\![P \wedge (x_1 \mathcal{I} x_2)]\!]\rho_{LP} = \mathcal{P}_{LP}[\![(x_1 \mathcal{I} x_2) \wedge P]\!]\rho_{LP} = \mathcal{P}_{LP}[\![P]\!]\rho'_{LP}$$

where $\rho'_{LP} = \rho_{LP} \oplus \{x_1 \mapsto a_1, x_2 \mapsto a_2\}$. The same constraint properties can be extended to Z: **Constraint Property 1**:

$$\mathcal{P}_Z[\![x_1 \mathcal{I} x_2]\!]\rho_Z = \mathcal{P}_Z[\![x_1 \mathcal{I} x_2]\!]\rho'_Z = true$$

where $\rho'_Z = \rho_Z \oplus \{x_1 \mapsto \gamma(a_1), x_2 \mapsto \gamma(a_2)\}$.
The environments of predicates conjoined to the infix predicates are also enhanced:
Constraint Property 2:

$$\mathcal{P}_Z[\![P \wedge (x_1 \mathcal{I} x_2)]\!]\rho_Z = \mathcal{P}_Z[\![(x_1 \mathcal{I} x_2) \wedge P]\!]\rho_Z = \mathcal{P}_Z[\![P]\!]\rho'_Z$$

where $\rho'_Z = \rho_Z \oplus \{x_1 \mapsto \gamma(a_1), x_2 \mapsto \gamma(a_2)\}$.

An extension of these properties is the case where x_1 can take many values. We call this:
Constraint Property 3

$$\mathcal{P}_{LP}[\![x_1 \mathcal{I} x_2]\!]\rho_{LP} = \mathcal{P}_{LP}[\![x_1 \mathcal{I} x_2]\!]\rho'_{LP} = true$$
$$\mathcal{P}_{LP}[\![P \wedge (x_1 \mathcal{I} x_2)]\!]\rho_{LP} = \mathcal{P}_{LP}[\![(x_1 \mathcal{I} x_2) \wedge P]\!]\rho_{LP} = \mathcal{P}_{LP}[\![P]\!]\rho'_{LP}$$

where

$$\begin{aligned}&\rho'_{LP} = \rho_{LP} \oplus \{x_1 \mapsto a_1\} \vee \rho'_{LP} = \rho_{LP} \oplus \{x_1 \mapsto a_2\} \vee \ldots \\ &\qquad \vee \rho'_{LP} = \rho_{LP} \oplus \{x_1 \mapsto a_n\}\end{aligned}$$

where $\rho'_{LP} \in Env_{LP}$. **Constraint Property 3** can similarly be applied to the Z interpretation. The first infix predicate we shall examine is equality, which considers the equality or otherwise of expressions x_1, x_2.

D.4.1 Infix Predicate: Equality

The interpretation of $x_1 = x_2$ in the LP not only evaluates to true or false but also potentially provides values for the environment. We need to examine three cases, where *both* (x_1, x_2) are defined, where *only one* is defined and where *neither* are defined.
(i) Both (x_1, x_2) defined:

We assume that both (x_1, x_2) are defined, or that there is sufficient information in ρ_{LP} for their evaluation. The truth or falsity of $x_1 = x_2$ in both the LP and in Z is determined by the value of both expressions:

$$\begin{aligned}
&\mathcal{P}_{LP}[\![x_1 = x_2]\!]\rho_{LP} \Leftrightarrow (\mathcal{E}_{LP}[\![x_1]\!]\rho_{LP} = \mathcal{E}_{LP}[\![x_2]\!]\rho_{LP})\\
&\mathcal{P}_{Z}[\![x_1 = x_2]\!]\rho_{Z} \Leftrightarrow (\mathcal{E}_{Z}[\![x_1]\!]\rho_{Z} = \mathcal{E}_{Z}[\![x_2]\!]\rho_{Z})
\end{aligned}$$

When both variables are defined, it is appropriate to consider **AR1**.
Condition 1 will hold for set operations for terminating computations, for if 'f' is equality, then

$$f_{LP}(\mathcal{E}_{LP}[\![x_1]\!]\rho_{LP}, \mathcal{E}_{LP}[\![x_2]\!]\rho_{LP})$$

is the boolean value of

$$(\mathcal{E}_{LP}[\![x_1]\!]\rho_{LP} = \mathcal{E}_{LP}[\![x_2]\!]\rho_{LP})$$

and so

$$\begin{aligned}
&f_{LP}(\mathcal{P}_{LP}[\![(x_1, x_2)]\!]\rho_{LP}) = f_{LP}(\mathcal{E}_{LP}[\![x_1]\!]\rho_{LP}, \mathcal{E}_{LP}[\![x_2]\!]\rho_{LP})\\
&\qquad = \mathcal{P}_{LP}[\![x_1 = x_2]\!]\rho_{LP}.
\end{aligned}$$

Condition 2 follows in a similar manner because if 'f' is equality, then

$$f_{Z}(\mathcal{E}_{Z}[\![x_1]\!]\rho_{Z}, \mathcal{E}_{Z}[\![x_2]\!]\rho_{Z}) \Leftrightarrow (\mathcal{E}_{Z}[\![x_1]\!]\rho_{Z} = \mathcal{E}_{Z}[\![x_2]\!]\rho_{Z})$$

and so

$$\begin{aligned}
&f_{Z}(\mathcal{P}_{Z}[\![(x_1, x_2)]\!]\rho_{Z}) = f_{Z}(\mathcal{E}_{Z}[\![x_1]\!]\rho_{Z}, \mathcal{E}_{Z}[\![x_2]\!]\rho_{Z})\\
&\qquad = \mathcal{P}_{Z}[\![x_1 = x_2]\!]\rho_{Z}.
\end{aligned}$$

Condition 3:
We require for f syntactic infix '=':

$$\gamma(f_{LP}(\mathcal{P}_{LP}[\![(x_1, x_2)]\!]\rho_{LP})) = f_{Z}(\gamma(\mathcal{P}_{LP}[\![(x_1, x_2)]\!]\rho_{LP})).$$

Suppose that $(a_1 = a_2)$ evaluates to *true* and since from finite theory of sets, equality is set-theoretic for finite sets, then $(\gamma(a_1) = \gamma(a_2))$ evaluates to tt and we have

$$\begin{aligned}
&\gamma(a_1 = a_2) = \gamma(true)\\
&\qquad = tt = \gamma(a_1) = \gamma(a_2) = f_{Z}(\gamma(a_1), \gamma(a_2)).
\end{aligned}$$

The same holds if $(a_1 = a_2)$ evaluates to *false*. Thus when x_1, x_2 are both defined and termination is successful, the predicate evaluation in the LP is an exact approximation to evaluation in the Z domain.

(ii) One of (x_1, x_2) defined:

Supposing that x_1 is undefined, $x_1 \mapsto \bot \in \rho_{LP}$ and $x_2 \mapsto a$ then the equality predicate results in the evaluation of x_1 via unification (and we obtain a similar result for x_1 defined, x_2 undefined).

If either of x_1, x_2 is undefined, it is more appropriate to consider **AR1** directly. Thus assuming that the execution terminates, and **Constraint Property 1** applies, then the approximation is again exact, for the left and right hand sides of **AR1** are equal:

$$\begin{aligned}\textbf{LHS of AR1} = &\ \gamma((\mathcal{P}_{LP}[\![(x_1 = x_2)]\!]\rho_{LP})) \\ &= \gamma(true) = tt \\ \textbf{RHS of AR1} = &\ (\mathcal{P}_Z[\![(x_1 = x_2)]\!]\rho_Z)) = tt\end{aligned}$$

It is possible that the computation fails to terminate and the LHS to evaluate to $\bot$. In that case the execution underestimates the Z interpretation.

(iii) Both (x_1, x_2) undefined:

If both x_1, x_2 are undefined or incomplete, then it is still possible for the environment to be enhanced since both of x_1, x_2 may become ground through unification. This is another example of **Constraint Property 1** : where x_1, x_2 both unify to the same ground term a and $\rho'_{LP} = \rho_{LP} \oplus \{x_1 \mapsto a, x_2 \mapsto a\}$.

AR1 can be evaluated for the case where x_1, x_2 are undefined, and there are three possibilities:

Program terminates This occurs when both x_1, x_2 become ground through unification, as explained above. **Conditions 1–3** hold in the same manner as when x_2 is defined and the approximation is exact.

Execution does not terminate, but variables become ground in Z This occurs when (for example) constraint properties on sets are such that x_1, x_2 in Z, but non-ground in the LP. The approximation underestimates for

$$\begin{aligned}\textbf{LHS of AR1} = &\ \gamma((\mathcal{P}_{LP}[\![(x_1 = x_2)]\!]\rho_{LP})) \\ &= \gamma(\bot) = \bot \\ \textbf{RHS of AR1} = &\ (\mathcal{P}_Z[\![(x_1 = x_2)]\!]\rho_Z)) = tt\end{aligned}$$

Execution does not terminate and neither variable becomes ground in Z

In that case the approximation is exact for

$$\begin{aligned}\textbf{LHS of AR1} &= \gamma(\bot) = \bot\\ \textbf{RHS of AR1} &= \ (\mathcal{P}_Z[\![(x_1 = x_2)]\!]\rho_Z)) = \bot\end{aligned}$$

We can summarise, thus. Three cases have been examined, depending on whether or not x_1, x_2 are defined prior to execution of equality function and in each case **AR1** is true where 'f' is the syntactic predicate $=$ for variable (x_1, x_2): **AR1** holds for $(x_1 = x_2)$.

D.4.2 Infix Predicate: Subset

The proof of correctness for predicate $\subseteq x$ where $x = (x_1, x_2)$ follows a similar structure to that for $=$. However the subset predicate potentially provides *more than one possible value* for the environment. In this case we must have x_2 defined. However, before the computation commences it may be the case that $x_1 \mapsto \bot \in \rho_{LP}$. After the computation, x_1 takes values from set x_2. The proof follows that for equality, with the difference that x_1 can take many values. The different values contribute to different answer substitutions. It can be shown that the predicate evaluation in the LP underestimates the evaluation in the Z domain. Again there are three cases, where we shall assume that the environment does not include incomplete sets and that sets are finite

(i) Both (x_1, x_2) defined:

Condition 1-2 will hold for set operations for terminating computations: the interpretation of ZF predicates on sets is built recursively in both the LP and Z.

$$\begin{aligned}&f_{LP}(\mathcal{P}_{LP}[\![(x_1, x_2)]\!]\rho_{LP}) = f_{LP}(\mathcal{E}_{LP}[\![x_1]\!]\rho_{LP}, \mathcal{E}_{LP}[\![x_2]\!]\rho_{LP})\\ &\quad = (\mathcal{E}_{LP}[\![x_1]\!]\rho_{LP} \subseteq_{LP} \mathcal{E}_{LP}[\![x_2]\!]\rho_{LP}) = \mathcal{P}_{LP}[\![x_1 \subseteq x_2]\!]\rho_{LP}.\\ &f_Z(\mathcal{P}_Z[\![(x_1, x_2)]\!]\rho_Z) = f_Z(\mathcal{E}_Z[\![x_1]\!]\rho_Z, \mathcal{E}_Z[\![x_2]\!]\rho_Z)\\ &\quad = (\mathcal{E}_Z[\![x_1]\!]\rho_Z \subseteq_Z \mathcal{E}_Z[\![x_2]\!]\rho_Z) = \mathcal{P}_Z[\![x_1 \subseteq x_2]\!]\rho_Z.\end{aligned}$$

Condition 3:

We require for f syntactic infix '$\subseteq$':

$$\gamma(f_{LP}(\mathcal{E}_{LP}[\![(x_1, x_2)]\!]\rho_{LP})) \sqsubseteq f_Z(\gamma(\mathcal{E}_{LP}[\![(x_1, x_2)]\!]\rho_{LP})).$$

Suppose that $(a_1 \subseteq_{LP} a_2)$ evaluates to *true* and since 'subset' for finite sets in the LP is set-theoretic, then $(\gamma(a_1) \subseteq_Z \gamma(a_2))$ evaluates to *tt* and we have

$$\begin{aligned}\gamma(a_1 \subseteq_{LP} a_2) &= \gamma(\mathit{true}) \\ &= \mathit{tt} = \gamma(a_1) \subseteq_Z \gamma(a_2) = f_Z(\gamma(a_1), \gamma(a_2)).\end{aligned}$$

The same holds if $(a_1 \subseteq_{LP} a_2)$ evaluates to *false*. Thus when termination is successful, the predicate evaluation in the LP is an exact approximation to evaluation in the Z domain.

(ii) Variable x_1 undefined

If variable x_1 is undefined, then we have an example of **Constraint Property 3** for suppose $\{a_1 \ldots a_n\}$ are subsets of x_2 in the LP then

$$\begin{aligned}&\mathcal{P}_{LP}[\![x_1 \subseteq x_2]\!]\rho_{LP} = \mathcal{P}_{LP}[\![x_1 \subseteq x_2]\!]\rho'_{LP} = \mathit{true} \\ &\mathcal{P}_{LP}[\![P \wedge (x_1 \subseteq x_2)]\!]\rho_{LP} = \mathcal{P}_{LP}[\![(x_1 \subseteq x_2) \wedge P]\!]\rho_{LP} = \mathcal{P}_{LP}[\![P]\!]\rho'_{LP}\end{aligned}$$

where

$$\begin{aligned}\rho'_{LP} = \rho_{LP} \oplus \{x_1 \mapsto a_1\} &\vee \rho'_{LP} = \rho_{LP} \oplus \{x_1 \mapsto a_2\} \vee \ldots \\ &\vee \rho'_{LP} = \rho_{LP} \oplus \{x_1 \mapsto a_n\}.\end{aligned}$$

In a similar fashion suppose $\{\gamma(a_1) \ldots \gamma(a_n)\}$ are subsets of x_2 in Z then

$$\begin{aligned}&\mathcal{P}_Z[\![x_1 \subseteq x_2]\!]\rho_Z = \mathcal{P}_Z[\![x_1 \subseteq x_2]\!]\rho'_Z = \mathit{true} \\ &\mathcal{P}_Z[\![P \wedge (x_1 \subseteq x_2)]\!]\rho_Z = \mathcal{P}_Z[\![(x_1 \subseteq x_2) \wedge P]\!]\rho_Z = \mathcal{P}_Z[\![P]\!]\rho'_Z\end{aligned}$$

where

$$\begin{aligned}\rho'_Z = \rho_Z \oplus \{x_1 \mapsto \gamma(a_1)\} &\vee \rho'_Z = \rho_Z \oplus \{x_1 \mapsto \gamma(a_2)\} \vee \ldots \\ &\vee \rho'_Z = \rho_Z \oplus \{x_1 \mapsto \gamma(a_n)\}.\end{aligned}$$

Where there is only one way the environment can be enhanced, then we can consider **AR1**. However where there is more than one way of enhancing the environment, the comparison between the Z and LP domains will be deferred to Section D.5 for in that case the values contribute to a set expression.

Thus assuming that the execution terminates, and x_1, x_2 take unique values then the left and right hand sides of **AR1** are as follows

$$\begin{aligned}\textbf{LHS of AR1} = \ & \gamma((\mathcal{P}_{LP}[\![(x_1 \subseteq x_2)]\!]\rho'_{LP})) \\ = \gamma(true) = tt & \\ \textbf{RHS of AR1} = \ & (\mathcal{P}_Z[\![(x_1 \subseteq x_2)]\!]\rho'_Z)) = tt\end{aligned}$$

(iii) Both Variables x_1, x_2 undefined

If both x_1, x_2 are undefined, then the computation in an LP such as Gödel will fail to terminate:

$$\begin{aligned}\textbf{LHS of AR1} = \ & \gamma((\mathcal{P}_{LP}[\![(x_1 \subseteq x_2)]\!]\rho_{LP})) \\ = \gamma(\perp) = \perp &\end{aligned}$$

which will underestimate the RHS. Thus the interpretation in the LP underestimates the interpretation in Z for all three cases above.

If either of x_1, x_2 are incomplete sets, then $x_1 \subseteq x_2$ evaluates to '$\perp$' in the LP. which underestimates the interpretation in Z. Thus **AR1** is true where 'f' is the syntactic predicate $\subseteq$ for variable (x_1, x_2), and the LP interpretation of $\subseteq$ underestimates the Z interpretation.

D.4.3 Infix Predicate: Membership

The last infix predicate is membership, $x_1 \in x_2$ and the proof follows that for $\subseteq$. If $x_1 \in x_2$ then x_1 has potentially many values for x_2 defined and not empty. The three cases can be summarised:

(i) Both x_1, x_2 are defined

$$\mathcal{P}_{LP}[\![x_1 \in x_2]\!]\rho_{LP} = true$$

where $x_1 \in x_2$. Otherwise the value if *false*. The predicate is interpreted as tt, ff respectively when evaluated in D_Z for $x_1 \in x_2, x_1 \notin x_2$.

If the computation terminates then the approximation is exact, as for = and $\subseteq$. The result follows similarly for Z, where as before we extend the interpretation of predicates in D_Z to cover constraint satisfaction. Thus for x_2 defined and not empty:

$$\mathcal{P}_Z[\![x_1 \in x_2]\!]\rho_Z = \mathcal{P}_Z[\![x_1 \in x_2]\!]\rho'_Z \ \& \ true$$

where $\rho'_{LP} = \rho_{LP} \oplus \{x_1 \mapsto a\}, a \in x_2$. If x_2 is defined and empty, then the predicate evaluates to *ff* in Z:

$$\mathcal{P}_Z[\![x_1 \in \varnothing]\!]\rho_Z = \mathcal{P}_Z[\![x_1 \in x_2]\!]\rho_Z \;\&\; ff$$

Thus the approximation for $\in$ is exact for both x_1, x_2 defined

(ii) for x_1 undefined and x_2 defined.

Constraint Properties 2-3 apply also for 'membership': Suppose $x_2 = \{a_1 \ldots a_n\}$, then

$$\mathcal{P}_{LP}[\![x_1 \in x_2]\!]\rho_{LP} = \mathcal{P}_{LP}[\![x_1 \in x_2]\!]\rho'_{LP}$$
$$\mathcal{P}_{LP}[\![P \wedge (x_1 \in x_2)]\!]\rho_{LP} = \mathcal{P}_{LP}[\![(x_1 \in x_2) \wedge P]\!]\rho_{LP} = \mathcal{P}_{LP}[\![P]\!]\rho'_{LP}$$

where

$$\rho'_{LP} = \rho_{LP} \oplus \{x_1 \mapsto a_1\} \vee \rho'_{LP} = \rho_{LP} \oplus \{x_1 \mapsto a_2\}, \ldots \vee \rho'_{LP} = \rho_{LP} \oplus \{x_1 \mapsto a_n\}.$$

Thus assuming that the execution terminates, the situation is the same as for subset in that the choice of the binding for x_1 is non-deterministic for sets with more than one member. Where the choice is deterministic then

$$\textbf{LHS of AR1} = \gamma((\mathcal{P}_{LP}[\![(x_1 \in x_2)]\!]\rho_{LP}))$$
$$= \gamma(true) = tt$$
$$\textbf{RHS of AR1} = (\mathcal{P}_Z[\![(x_1 \in x_2)]\!]\rho_Z)) = tt$$

(iii) Both x_1, x_2 undefined

For x_2 undefined the LP interpretation results in $\perp$ and underestimates the Z interpretation, however it evaluates.

We have examined all possibilities for values of x_1, x_2 and in all cases **AR1** is true where '*f*' is the syntactic predicate $\in$ for variable (x_1, x_2); the LP interpretation of $\in$ underestimates the Z interpretation as required.

D.5 Set Comprehension and Variable Declarations

Set Comprehension is defined in terms of declarations $D_1;\ \ldots;\ D_n$, a constraining predicate p and an expression t involving the declared variables:

$$x_1 : \tau_1;\ x_2 : \tau_2;\ \ldots x_n : \tau_n \mid p \bullet t$$

We first present the interpretation of declarations, then within the context of a set declaration.

D.5.1 Variable Declarations

Variable declarations occur within bound expressions with structure: $\_\_\ D \mid p \bullet t\ \_\_$ where D is a declaration,p is a predicate and t a term. D is of the form:

$$\{x_1 : \tau_1;\ x_2 : \tau_2;\ \ldots x_n : \tau_n\}$$

These include *set comprehensions, quantified expressions, lambda expressions* and *schemas.* The declaration results in a single tuple of values $(x_1, \ldots x_n)$ being generated (or tested in the case of schemas). Each value is constrained by p and used to evaluate t.

An evaluation function $\mathcal{D}_{LP}$ gives the interpretation in D_{LP} of syntactic declarations $x : \tau$, where x is a variable and τ is set-valued with value provided by ρ_{LP}. The declarations considered in this section do not include schema references, for these are treated separately.

1. τ is a *set:*

   $$\mathcal{D}_{LP}[\![x : \tau]\!]\rho_{LP} = \mathcal{P}_{LP}[\![x \in \tau]\!]\rho_{LP}$$

   '$x : \tau$' has the effect of either testing a value or updating the environment as in the case of the membership predicate.

2. τ is a *Power Set*, $\tau = \mathbb{P}\,\tau'$ say:

   $$\mathcal{D}_{LP}[\![x : \mathbb{P}\,\tau']\!]\rho_{LP} = \mathcal{P}_{LP}[\![x \subseteq \tau']\!]\rho_{LP}$$

   '$\tau : \mathbb{P}\,\tau'$ ' uses a 'subset' test rather than a 'membership of power set' test for reasons of efficiency. It has the same effect on the environment as the subset predicate.

3. τ is a *Cartesian Product*, $\tau_1 \times \tau_2$:

   $$\begin{aligned}\mathcal{D}_{LP}[\![x : \tau_1 \times \tau_2]\!]\rho_{LP} = {} & \mathcal{P}_{LP}[\![x = (x_1 \mapsto x_2)]\!]\rho_{LP} \\ & \& \ \mathcal{P}_{LP}[\![x_1 \in \tau_1]\!]\rho_{LP}) \ \& \ \mathcal{P}_{LP}[\![x_2 \in \tau_2]\!]\rho_{LP})\end{aligned}$$

   'T2' captures a representation of ordered pair (as an example of a tuple) in the LP. In our Gödel library this is 'OrdPair'. The following shows the implementation of $\nrightarrow$, which illustrates the interpretation of cartesian product.

```
PF(pf, s1, s2) <- ALL [z,x,y] (z In pf &
                               (z = OrdPair(x,y))
                    ->  (x In s1) & (y In s2)  &
                    ALL [u] (OrdPair(x, u) In pf -> u = y)).
```

Thus a single declaration (such as $x : \tau$) has the effect of enhancing the environment as for the membership predicate:

$$\mathcal{D}_{LP}[\![x : \tau]\!]\rho_{LP} = \mathcal{P}_{LP}[\![x \in \tau]\!]\rho_{LP} = \mathcal{P}_{LP}[\![x \in \tau]\!]\rho'_{LP}$$

where if $\tau = \{a_1 \dots a_n\}$ then

$$\rho'_{LP} = \rho_{LP} \oplus \{x \mapsto a_1\} \vee \rho'_{LP} = \rho_{LP} \oplus \{x \mapsto a_2\}, \dots \vee \rho'_{LP} = \rho_{LP} \oplus \{x \mapsto a_n\}.$$

In general, if $x : \tau$ is a declaration, then

$$\mathcal{D}_{LP}[\![x : \tau]\!]\rho_{LP} = \mathcal{P}_{LP}[\![x \in \tau]\!]\rho'_{LP}$$

where it is possible for ρ'_{LP} to take many values determined by the nature of the type τ.

A sequence of declarations is evaluated in the LP as a conjunction

$$\mathcal{D}_{LP}[\![D_1;\ \dots;\ D_n]\!]\rho_{LP} = \mathcal{D}_{LP}[\![D_1]\!]\rho_{LP}\ \&\ \dots\ \&\ \mathcal{D}_{LP}[\![D_n]\!]\rho_{LP}.$$

Declarations can be represented in a simpler manner in Z, where again values are chosen from some set-valued τ. However in this case τ is a type constructor thus

$$\mathcal{D}_{Z}[\![x : \tau]\!]\rho_{Z} = \mathcal{P}_{Z}[\![x \in \tau]\!]\rho'_{Z}$$

where τ is a set, or a power set, $\mathbb{P}\,\tau'$, or a cartesian product $\tau' \times \tau''$ and ρ'_Z takes its values from τ.

D.5.2 Interpretation of Set Comprehension

A set comprehension is

$$\{x_1 : \tau_1;\ x_2 : \tau_2;\ \dots x_n : \tau_n \mid p \bullet t\}$$

where each $x_i : \tau_i$ provides a value which contributes to the tuple $(x_1, \ldots x_n)$ which is used to evaluate t. Thus if $s = \{d \mid p \bullet t\}$ is a syntactical set comprehension it is interpreted in D_{LP} as $\mathcal{E}_{LP}[\![s]\!]\rho_{LP}$ and in D_Z as $\mathcal{E}_Z[\![s]\!]\rho_Z$. Since declarations in the LP are treated as predicates, then the set comprehension of s is interpreted in the LP

$$\mathcal{E}_{LP}[\![s]\!]\rho'_{LP} = \{\mathcal{D}_{LP}[\![d]\!]\rho'_{LP} \;\;\&\;\; \mathcal{P}_{LP}[\![p]\!]\rho'_{LP} \bullet \mathcal{E}_{LP}[\![t]\!]\rho'_{LP}\}$$

The environment ρ'_{LP} inside the comprehension is the variable which acts as a set generator, for recall that $\mathcal{D}_{LP}[\![x : \tau]\!]\rho_{LP} = \mathcal{P}_{LP}[\![x \in \tau]\!]\rho'_{LP}$. A similar interpretation is true for D_Z.

We assert that for terminating computations, **AR1** is true, since the interpretation of set comprehension is exact. We initiate an induction process over the set generators (as in [17]).

A set with *no* set generators, is defined in the LP domain:

$$\mathcal{E}_{LP}[\![\{\mid p \bullet t\}]\!]\rho'_{LP} = \{\mathcal{E}_{LP}[\![t]\!]\rho'_{LP}\} \text{ where } \mathcal{P}_{LP}[\![p]\!]\rho'_{LP} = \mathit{true}$$
$$\mathcal{E}_{LP}[\![\{\mid p \bullet t\}]\!]\rho'_{LP} = \{\} \text{ where } \mathcal{P}_{LP}[\![p]\!]\rho'_{LP} = \mathit{false}$$

In the Z domain

$$\mathcal{E}_{Z}[\![\{\mid p \bullet t\}]\!]\rho'_{Z} = \{\mathcal{E}_{Z}[\![t]\!]\rho'_{Z}\}, \text{ where } \mathcal{P}_{Z}[\![p]\!]\rho'_{Z} = \mathit{tt}$$
$$\mathcal{E}_{Z}[\![\{\mid p \bullet t\}]\!]\rho'_{Z} = \{\} \text{ where } \mathcal{P}_{Z}[\![p]\!]\rho'_{Z} = \mathit{ff}$$

Induction is based on the equivalence:

$$\{x_1 : \tau_1;\ x_2 : \tau_2;\ \ldots x_n : \tau_n \mid p \bullet t\} = \bigcup\{x_1 : \tau_1 \bullet \{x_2 : \tau_2;\ \ldots x_n : \tau_n \mid p \bullet t\}\}$$

We first consider terminating computations where the interpretation is proposed as exact. The induction process depends on showing that if we assume that **AR1** holds for

$$\{x_1 : \tau_1;\ x_2 : \tau_2;\ \ldots x_n : \tau_n \mid p \bullet t\}$$

then it holds for

$$\{x_1 : \tau_1;\ x_2 : \tau_2;\ \ldots x_n : \tau_n;\ x_{n+1} : \tau_{n+1} \mid p \bullet t\}$$

For the induction process we first consider the base case

Base Case: no set generators

We consider **AR1** for the base case where there are no set generators. **Conditions 1–2** are true for standard sets since the interpretation is built recursively in both the LP and Z domains:

$$
\begin{array}{l}
\mathcal{E}_{LP}[\![\{\mid p \bullet t\}]\!]\rho'_{LP} = \{\mathcal{E}_{LP}[\![t]\!]\rho'_{LP}\} \text{ where } P_{LP}[\![p]\!]\rho'_{LP} = \mathit{true} \\
\mathcal{E}_{LP}[\![\{\mid p \bullet t\}]\!]\rho'_{LP} = \{\} \text{ where } \mathcal{P}_{LP}[\![p]\!]\rho'_{LP} = \mathit{false} \\
\mathcal{E}_{Z}[\![\{\mid p \bullet t\}]\!]\rho'_{Z} = \{\mathcal{E}_{Z}[\![t]\!]\rho'_{Z}\}, \text{ where } \mathcal{P}_{Z}[\![p]\!]\rho'_{Z} = \mathit{tt} \\
\mathcal{E}_{Z}[\![\{\mid p \bullet t\}]\!]\rho'_{Z} = \{\} \text{ where } \mathcal{P}_{Z}[\![p]\!]\rho'_{Z} = \mathit{ff}
\end{array}
$$

Assuming that the calculation of p, t for environment ρ'_{LP} terminates, the approximation of '$\{\mid p \bullet t\} = f(p,t)$' in the LP domain is exact, since **Condition 3** becomes:

$$\gamma(f_{LP}(\mathcal{E}_{LP}[\![(p,t)]\!]\rho'_{LP})) = f_Z(\gamma(\mathcal{E}_{LP}[\![(p,t)]\!]\rho'_{LP})).$$

If the calculation of p, t fails to terminate, then the LHS of the approximation evaluates to $\gamma(\perp \circ \perp) = \varnothing_{\cup\perp}$ and thus underestimates the RHS, however it is evaluated.

$$
\begin{array}{ll}
\text{LHS} & = \gamma(\mathcal{E}_{LP}[\![f(p,t)]\!]\rho_{LP}) = \gamma(\perp \circ \perp) = \varnothing_{\cup\perp} \\
\text{RHS} & = \mathcal{E}_Z[\![f(p,t)]\!](\gamma \circ \rho_{LP})
\end{array}
$$

Set Comprehension – Induction on Declaration Sequence

Induction is based on the equivalence:

$$\{x_1 : \tau_1;\ x_2 : \tau_2;\ \ldots x_n : \tau_n \mid p \bullet t\} = \bigcup\{x_1 : \tau_1 \bullet \{x_2 : \tau_2;\ \ldots x_n : \tau_n \mid p \bullet t\}\}$$

for values of $\tau_1, \ldots \tau_n$ in the environment. Write the interpretation of $\bigcup$ in Z domain and LP domains as $\bigcup_Z$ and $\bigcup_{LP}$ as in Section D.3.

The equivalence means that the set comprehension with one generator, '$\{x : \tau \mid p \bullet t\}$', can be evaluated in the LP environment:

$$\mathcal{E}_{LP}[\![\{x : \tau \mid p \bullet t\}]\!]\rho_{LP} = \bigcup_{LP} \mathcal{E}_{LP}[\![\{\mid p \bullet t\}]\!]\rho'_{LP}$$

where τ is $\{a_1 \ldots a_n\}$ in ρ_{LP} and $\rho'_{LP} = \rho_{LP} \oplus \{x \mapsto a_i\}$. The interpretation is similar for D_Z.

For n generators, $x_1 : \tau_1;\ x_2 : \tau_2;\ \ldots x_n : \tau_n$, if $\tau_1 \mapsto s \in \rho_{LP}$, then $\tau_1 \mapsto \gamma(s) \in \rho_Z$ and the set comprehensions in both the Z and LP domains can be represented as the distributed union of a family of sets indexed by i where $a_i \in s$, $b_i \in \gamma(s)$ respectively:

$$\begin{aligned}
&\mathcal{E}_{LP}[\![\{x_1 : \tau_1;\ \ldots x_n : \tau_n \mid p \bullet t\}]\!]\rho_{LP} = \\
&\qquad \bigcup\nolimits_{LP}\{a_i \in s \bullet \mathcal{E}_{LP}[\![\{x_2 : \tau_2;\ \ldots x_n : \tau_n \mid p \bullet t\}]\!](\rho_{LP} \oplus \{x_1 \mapsto a_i\})\} \\
&\mathcal{E}_{Z}[\![\{x_1 : \tau_1;\ x_2 : \tau_2;\ \ldots x_n : \tau_n \mid p \bullet t\}]\!]\rho_{Z} = \\
&\qquad \bigcup\nolimits_{Z}\{b_i \in \gamma(s) \bullet \mathcal{E}_{Z}[\![\{x_2 : \tau_2;\ \ldots x_n : \tau_n \mid p \bullet t\}]\!](\rho_{Z} \oplus \{x_1 \mapsto b_i\})\}
\end{aligned}$$

For finite sets, the approximation of **Condition 3** is exact. Thus since **AR1** holds for the empty sequence and assuming it holds for the sequence

$$\{x_2 : \tau_2;\ \ldots x_n : \tau_n \mid p \bullet t\}$$

in environments

$$\rho_Z \oplus \{x_1 \mapsto b_i\},\ \rho_{LP} \oplus \{x_1 \mapsto a_i\}$$

it then holds for

$$\{x_1 : \tau_1;\ x_2 : \tau_2;\ \ldots x_n : \tau_n \mid p \bullet t\}$$

in environments ρ_Z, ρ_{LP}. Thus **AR1** holds for $\{x_1 : \tau_1;\ x_2 : \tau_2;\ \ldots x_n : \tau_n \mid p \bullet t\}$ since it holds for each of its components τ_i, p, t.

For infinite sets, or if any set is non-standard in the LP, the induction process depends on whether we are addressing set terms or sets of answer substitutions:

- For set terms the LP interpretation of s evaluates to $\perp \circ \perp$ and underestimates the Z interpretation in the same manner as the 'no set generator case'.
- For the 'answer set' the result depends on distributed union, where incomplete sets are involved. This is because we can equivalently express a set comprehension as a distributed union. We see that this underestimates for incomplete sets.

Thus set comprehension in the LP is an exact interpretation for finite or complete sets. For infinite or incomplete sets the LP interpretation is an underestimation. This is true for either set terms or sets of answer substitutions.

D.5.3 Set Operations Power Set, Set Intersection

Other set operations $\epsilon = f(x_1, x_2, \ldots, x_n)$ can be expressed via set comprehensions. Examples are set intersection and power set:

Set Intersection $s = x_1 \cap x_2$ in the LP is part of the library of set operations. However $\cap$ can be expressed as

```
s = {x : (x In x1) & (x In x2 )}.
```

where we are assuming that x_1, x_2 are appropriately typed. In Z this last condition is expressed explicitly so

$$s = x_1 \cap x_2 = \{x : X \mid (x \in x_1 \land x \in x_2) \bullet x\}$$

This is treated as a set comprehension where $p = x \in x_1 \land x \in x_2$ Thus for terminating computations, the interpretation in the LP approximates exactly, and for non-terminating computations, the LP interpretation underestimates.

Power Set in the LP, $s = \mathbb{P}\, x$ can be be expressed in Gödel as

```
s = {z : z Subset x }.
```

Its generic 'LP form' is as a set comprehension with predicate *true*:

$$s = \mathcal{E}_{LP}[\![\{z \subseteq x \mid true \bullet z\}]\!]\rho_{LP}.$$

Since 'power set' is a type in Z, there is no specific definition for it (see Chapter 3). The power set axiom of ZF (from Appendix A) provides a definition in Z: The power set set $s = \mathbb{P}\, x$ is such that

$$\forall z \bullet (z \in s \Leftrightarrow z \subseteq x)$$

and this set and the interpretation in the LP can be shown to be equal. Thus the interpretation of power set is exact for finite sets. For infinite sets, the LP interpretation is $\bot \circ \bot$ which always underestimates.

The interpretation is exact if the computations terminate. For infinite or incomplete sets, the interpretation in the LP evaluates to $\bot \circ \bot$ and so underestimates the interpretation in Z.

D.5.4 Quantifiers

Universal Quantification

The syntactic predicate '$\forall x : s \mid p \bullet q$' is interpreted in the LP :

```
ALL [x] (x In s <-> p => q )
```

and in Z $\forall x : s \mid p \Rightarrow q$ and is evaluated for finite sets s on an element by element basis for values of s. Its interpretation can be denoted in the LP as $\mathcal{P}_{LP}[\![fx]\!]\rho_{LP}$, where x is the tuple s, p, q and 'f' the syntactic '$\forall$'. For terminating computations, **Condition 1–2** hold in the LP and in Z. If '$\forall x : s \mid p \Rightarrow q$' is *true* then **Condition 3** becomes

$$\begin{aligned}
\textbf{LHS} =\ & \gamma(f_{LP}(\mathcal{P}_{LP}[\![(s,p,q)]\!]\rho_{LP})) \\
=\ & \gamma(true) = tt \\
\textbf{RHS} =\ & f_Z(\gamma(\mathcal{P}_{LP}[\![(s,p,q)]\!]\rho_{LP})) = tt.
\end{aligned}$$

and is thus exact for each p, q in an environment containing s. The result follows similarly if '$\forall x : s \mid p \bullet q$' is *false*.

For infinite sets the truth value in the LP will be $\perp$ i.e. it will fail to terminate and the LHS of Condition 3 will evaluate to $\perp$. For infinite sets, the Z interpretation of the quantification will result in the value tt or ff, and $\gamma(\perp) = \perp$ and $\perp \sqsubseteq ff, \perp \sqsubseteq tt$. For cases where s is incomplete, or not fully defined, then the LP interpretation results in $\perp$, which either underestimates the interpretation in Z, if it is ff, tt, or is exact, if the interpretation in Z is $\perp$. Thus in all cases, the interpretation in the LP of universal quantification adheres to **AR1**.

Existential Quantification

A similar interpretation is true for $\exists$ where '$\exists x : s \mid p \bullet q$' is interpreted in the LP

```
SOME [x] (x In s <-> p & q )
```

and in Z: $\exists x : s \mid p \wedge q$ The LP interpretation evaluates to $true, false, \perp$, which always underestimates its interpretation in Z, as for $\forall$.

D.6 Function Application and Lambda Expressions

Function application of t_1 to t_2 assumes that t_1 is appropriately typed, as a set of pairs. It is interpreted in the LP by

$$\mathcal{E}_{LP}[\![t_1 t_2]\!]\rho_{LP} = a \Leftrightarrow t_2 \mapsto a \in t_1$$

It is mapped in a similar way in Z. For terminating computations, where set t_1 is finite, the interpretation is exact. Where t_1 is infinite or incomplete, the LP underestimates the Z interpretation for

$$\mathcal{E}_{LP}[\![t_1 t_2]\!]\rho_{LP} = \bot.$$

Lambda expressions require evaluation individually:
$\lambda\, x_1 : \tau_1;\ \ldots x_n : \tau_n \mid p \bullet t$ where t is a term can be expressed (in Z) as a set of maplets $x \mapsto a$ where the x is a tuple $(x_1, \ldots, x_n)$ and a is the term t evaluated at $(x_1, \ldots, x_n)$:

$$\{x_1 : \tau_1;\ \ldots x_n : \tau_n \mid p \bullet (x_1, \ldots, x_n) \mapsto t)\}$$

It is interpreted as the equivalent set expression in the LP:

$$\begin{array}{l}\mathcal{E}_{LP}[\![\lambda\, x_1 : \tau_1;\ \ldots x_n : \tau_n \mid p \bullet t]\!]\rho_{LP} = \\ \mathcal{E}_{LP}[\![\{x_1 : \tau_1;\ \ldots x_n : \tau_n \mid p \bullet (x_1, \ldots, x_n) \mapsto t)\}]\!]\rho_{LP} = \\ \{\mathcal{D}_{LP}[\![x_1 : \tau_1;\ \ldots x_n : \tau_n]\!]\rho_{LP}\ \&\ \mathcal{P}[\![p]\!] \mid T2(Tn(x_1, \ldots, x_n) \mapsto t)\}\end{array}$$

An example can be seen in Appendix C. The approximation is exact for terminating computations and underestimates for the rest.

D.7 Interpretation of Schemas and Schema Expressions

Suppose that the syntactic objects $schema, axdef \in \Sigma_3$ are interpreted in the LP and in Z by $\mathcal{S}_{LP}$, $\mathcal{S}_Z$ respectively. A schema can be represented (in its horizontal form) by the following syntactic object:

$$Sch \mathrel{\widehat{=}} \big[\, D_1;\ \ldots;\ D_n \ \big|\ CP \,\big]$$

where $D_i = X_i : \tau_i$, and $CP \ ::=\ CP_1 \wedge \ldots \wedge CP_m$.

Sch evaluates to a set expression, of bindings of variable name(s) to values. The bindings are constrained by the variable declarations and by the schema predicate. Suppose *GCP* is defined as *CP* where all the free occurrences of $X_1 \ldots X_n$ are replaced by $x_1 \ldots x_n$

$$GCP(x_1, \ldots, x_n) = CP(X_1/x_1, \ldots, X_n/x_n)$$

and any bound variables replaced by arbitrary local variables. *A set of schema* *bindings* of *Sch* can be represented in Z (as suggested in [17]) by a set expression:

$$\{x_1 : \tau_1;\ \ldots;\ x_n : \tau_n \mid GCP \bullet \{X_1 \mapsto x_1, \ldots, X_n \mapsto x_n\}\}$$

There is a similar representation in the LP where $[Bind_1(X_1, x_1), \ldots, Bind_n(X_n, x_n)]$ replaces $\{X_1 \mapsto x_1, \ldots, X_n \mapsto x_n\}$. If we assume that the set of bindings is constrained by an initial imposed environment ρ^o then the interpretation of the schema $Sch \mathrel{\widehat=} [\, D \mid CP \,]$ is the interpretation of a set expression

$$\begin{aligned}
&\mathcal{S}_{LP}[\![[\, X_1 : \tau_1;\ \ldots;\ X_n : \tau_n \mid CP \,]]\!]\rho^o_{LP} \\
&= \mathcal{E}_{LP}[\![\{x_1 : \tau_1;\ \ldots;\ x_n : \tau_n \mid GCP \\
&\qquad \bullet [Bind_1(X_1, x_1), \ldots, Bind_n(X_n, x_n)]\}]\!]\rho^o_{LP}
\end{aligned}$$

The interpretation of schemas and schema expressions is in terms of a *characteristic predicate*, providing a single binding for a schema expression.

D.7.1 Characteristic Predicate for a Schema Expression

A schema binding is obtained by providing the schema with some initial environment, ρ^0_{LP}. In its initial state a schema is interpreted as:

$$\mathcal{S}_{LP}[\![[\, X_1 : \tau_1;\ \ldots;\ X_n : \tau_n \mid CP \,]]\!]\rho^o_{LP}$$

and this evaluates in the LP to bindings of variable names to values. During the execution the environment has been enhanced to ρ This binding is a member of the set defined previously:

$$\begin{aligned}
&\mathcal{S}_{LP}[\![[\, X_1 : \tau_1;\ \ldots;\ X_n : \tau_n \mid CP \,]]\!]\rho^o_{LP} \\
&= \mathcal{E}_{LP}[\![\{x_1 : \tau_1;\ \ldots;\ x_n : \tau_n \mid GCP \\
&\qquad \bullet [Bind_1(X_1, x_1), \ldots, Bind_n(X_n, x_n)]\}]\!]\rho^o_{LP}
\end{aligned}$$

where each x_i satisfies

$$\mathcal{D}_{LP}[\![D_1;\ \ldots;\ D_n]\!]\rho_{LP} \;\&\; \mathcal{P}_{LP}[\![GCP]\!]\rho_{LP}.$$

where each enhanced environment $\rho_{LP} \in Env_{LP}$. The characteristic schema predicate of Sch is as follows:

$$\begin{array}{l} SchemaType(binding, Sch) \Leftrightarrow \\ \quad (binding = [Bind_1(X_1, x_1), \ldots, Bind_n(X_n, x_n)]) \ \& \\ \qquad \mathcal{D}_{LP}[\![D_1; \ \ldots; \ D_n]\!]\rho_{LP} \ \& \ \mathcal{P}_{LP}[\![GCP]\!]\rho_{LP}. \end{array}$$

The values $x_1 \mapsto a_1 \ldots x_n \mapsto a_n$ which satisfy $SchemaType$ have been either generated or were part of the initial environment. Note that although the schema definition in the LP uses 'if' ($\Leftarrow$), by the CWA, this has the same effect as 'if an only if' ($\Leftrightarrow$).

The Z interpretation can similarly be represented by a set of bindings where

$$binding = \{X_1 \mapsto \gamma(x_1), \ldots, X_n \mapsto \gamma(x_n)\}$$

The values $\gamma(x_i) \in \operatorname{ran} \rho_Z$ satisfy

$$(\mathcal{D}_Z[\![x_1 : \tau_1]\!]\rho_Z) \ \& \ \ldots (\mathcal{D}_Z[\![x_n : \tau_n]\!]\rho_Z) \ \& \ \mathcal{P}_Z[\![GCP]\!]\rho_Z.$$

AR1 can now be considered for schemas and is worth restating. If ϵ is a syntactic Z expression for a set of schema bindings then condition **AR1** must hold for a correct animation of Z in D_{LP}:

Approximation Rule 1 (AR1)

$$\gamma(\mathcal{S}_{LP}[\![\epsilon]\!]\rho_{LP}) \sqsubseteq \mathcal{S}_Z[\![\epsilon]\!](\gamma \circ \rho_{LP}).$$

where

$$\epsilon = \{X_1 : \tau_1 \ldots X_n \tau_n \mid CP \bullet \{X_1 \mapsto x_1, \ldots X_n \mapsto x_n\}\}$$

The structural induction rule states that if it can be shown that **AR1** holds for syntactic variable $\epsilon = x$, then it also holds for syntactical expression $\epsilon = fx$, where in this case, f is a syntactic operator which forms a schema from tuple $\epsilon = (D, CP)$, where D is a declaration and CP is a predicate. We denote by f_Z, f_{LP} the interpretation in the Z domain and LP domain respectively of the syntactic expression fx. Thus the left hand side of **AR1** is

$$\begin{array}{l} \gamma(\mathcal{S}_{LP}[\![[\, X_1 : \tau_1; \ \ldots; \ X_n : \tau_n \mid CP \,]]\!]\rho^o_{LP}) \\ = \gamma(\mathcal{E}_{LP}[\![\{x_1 : \tau_1; \ \ldots; \ x_n : \tau_n \mid GCP \bullet [Bind_1(X_1, x_1), \ldots, Bind_n(X_n, x_n)]\}]\!]\rho^o_{LP}) \end{array}$$

The right hand side of **AR1** is:

$$\mathcal{S}_Z[\![[\, X_1 : \tau_1;\ \ldots;\ X_n : \tau_n \mid CP \,]]\!]\rho_Z^o$$
$$= \mathcal{E}_Z[\![\{x_1 : \tau_1;\ \ldots;\ x_n : \tau_n \mid GCP \bullet \{X_1 \mapsto x_1, \ldots, X_n \mapsto x_n\}\}]\!]\rho_Z^o$$

These are *set comprehension*, which have been treated in Section D.5. These interpret exactly where components are finite and complete. Where the answer sets is incomplete the LP interpretation underestimates.

D.7.2 Schema Conjunction and Disjunction

We now interpret syntactical objects such as $Sch \mathrel{\widehat{=}} Sch^1 \wedge Sch^2$ and $Sch \mathrel{\widehat{=}} Sch^1 \vee Sch^2$. Provided that Sch^1, Sch^2 have compatible declarations their conjunction and disjunction can be defined. These are modelled by conjunction and disjunction of the LP predicates of Sch^1, Sch^2 with lists of bindings appended. This does cause duplication but has not (so far) been found a practical problem. Suppose Sch^1 has predicate CP^1 and declaration sequence D^1 where

$$D^1 = X_1^1 : \tau_1^1;\ \ldots;\ X_n^1 : \tau_n^1$$

modelled by Gödel list b_1:

$$[\ Bind_1^1(X_1^1, x_1^1), \ldots, Bind_n^1(X_n^1, x_n^1)\ ]$$

and Sch^2 has a predicate CP^2 and compatible declaration sequence D^2 where

$$D^2 = X_1^2 : \tau_1^2;\ \ldots;\ X_n^2 : \tau_n^2$$

modelled by Gödel list b_2. Given that the characteristic predicates of Sch^1, Sch^2 are respectively

$$SchemaType(binding, Sch^1), SchemaType(binding, Sch^2)$$

then the characteristic predicate of Sch is

$$SchemaType(binding, Sch) \Leftrightarrow (binding = b_1 \frown b_2)\ \&$$
$$\mathcal{D}_{LP}[\![D^1;\ D^2]\!]\rho_{LP}^o\ \&\ \mathcal{P}_{LP}[\![GCP^1 \wedge GCP^2]\!]\rho_{LP}^o.$$

We show that this interprets exactly the Z interpretation where the program terminates, and provides an incomplete set of answer substitutions when the program fails to terminate.

We show this by showing that the above definition gives the same value(s) as the value(s) obtained by 'expanding out' the version of $Sch \mathrel{\widehat{=}} Sch^1 \wedge Sch^2$. which is interpreted in Z as

$$\begin{aligned}&\mathcal{S}_Z[\![Sch^1 \wedge Sch^2]\!]\rho_Z\\ &\qquad = \mathcal{S}_Z[\![[\, D^1;\ D^2 \mid CP^1 \wedge CP^2 \,]]\!]\rho_Z^0\end{aligned}$$

The above represents the declaration sequences before they are merged, so some repetitions would be expected. The above represents the declaration sequences before they are merged, so some repetitions would be expected. Sch is then expressed as a set comprehension in the usual way:

$$\begin{aligned}&\mathcal{E}_Z[\![\{x_1^1 : \tau_1^1;\ \ldots;\ x_n^1 : \tau_n^1;\quad x_1^2 : \tau_1^2;\ \ldots;\ x_n^2 : \tau_n^2 \mid GCP^1 \wedge GCP^2\\ &\qquad \bullet \{X_1^1 \mapsto x_1^1, \ldots, X_n^1 \mapsto x_n^1;\quad X_1^2 \mapsto x_1^2, \ldots, X_n^2 \mapsto x_n^2\}\}]\!]\rho_Z^o\end{aligned}$$

Then $Sch^1 \wedge Sch^2$ in the LP is:

$$\mathcal{S}_{LP}[\![Sch^1 \wedge Sch^2]\!]\rho_{LP}$$

where given that the characteristic predicates of Sch^1, Sch^2 are respectively

$$SchemaType(binding, Sch^1), SchemaType(binding, Sch^2).$$

then the characteristic predicate of Sch evaluates to

$$\begin{aligned}&SchemaType(binding, Sch) \Leftrightarrow\\ &(binding = [\ Bind_1^1(X_1^1, x_1^1), \ldots, Bind_n^1(X_n^1, x_n^1),\\ &Bind_1^2(X_1^2, x_1^2), \ldots, Bind_n^2(X_n^2, x_n^2)]\ )\\ &\qquad \&\ \mathcal{D}_{LP}[\![D^1;\ D^2]\!]\rho_{LP}^o\ \&\ \mathcal{P}_{LP}[\![GCP^1 \wedge GCP^2]\!]\rho_{LP}^o\end{aligned}$$

which is the same as if the expression had been expanded first. The criteria for exactness or underestimation for each of these interpretations has already been discussed. In general, where each component of an expression is exact, the whole expression is exact,but where one component underestimates, the whole underestimates.

Schema disjunction is defined in a similar manner. If $Sch \mathrel{\widehat{=}} Sch^1 \vee Sch^2$ then the bindings are appended and the LP interpretations of the schema predicates are disjoined. In Chapter 4, the convention for naming variables is further refined, so that priming, input, output becomes apparent. The formalism is not explored here.

However the naming convention enables schema composition and piping to be accomplished. An outline is presented in [119].

D.7.3 Schema Reference in a Declaration

A declaration can contain a schema reference. If $x_i \in VAR, t, t_i \in expr, S_i \in NAME$, where *Sch* is a schema reference then recall from Section 5.3.1

$$basic\_decl ::= x_1, \ldots, x_n : t \mid Sch$$

The interpretation of this in Z is that its declarations are merged with the declarations of the schema which reference it and its predicate is conjoined. Thus if

$$Sch_1 \mathrel{\widehat{=}} [\, X_1 : \tau_1; \; \ldots X_n : \tau_n; \; S \mid \; predicate \;\; of \;\; Sch_1 \,]$$

Then this is equivalent to $Sch_1 \mathrel{\widehat{=}} Sch \wedge Sch_2$ where

$$Sch_2 \mathrel{\widehat{=}} [\, X_1 : \tau_1; \; \ldots X_n : \tau_n \mid \; predicate \;\; of \;\; Sch_1 \,]$$

Thus if a schema *Sch* appears as a reference in the declarations of schema Sch_1, then this is treated as for schema conjunction above: *Sch* is removed from the declarations and conjoined to the predicate of *Sch* and its remaining declarations.

D.7.4 Binding Formation θ

The binding formation θSch can be used to form a binding. Its value depends on the environment. However we interpret it here in the same context as in the Unix file system case study. In that case study $\{Sch \bullet \theta Sch\}$ was constructed first and θSch was interpreted as a member of that set. The set $\{Sch \bullet \theta Sch\}$ in the LP is the 'same' set as $\mathcal{S}_{LP}[\![Sch]\!]\rho_{LP}$ however in this case it is a set *term* and not an answer set. It is the set comprehension S defined by

$$S = \{SchemaType(binding, Sch) \bullet binding\}$$

so that the binding formation $\theta Sch \in S$ where the code can be found in Appendix C.

This means that if the computation terminates its interpretation is exact, and if one of the members of s fails to terminate then the output of the whole computation is $\perp$.

D.7.5 Axiomatic and Generic Definitions

Axiomatic and Generic Definitions require individual definitions; they are interpreted in such a way that they are exact where the computations terminate.

Axiomatic Definitions are modelled in the same way as a schema, and suitable *names* must be generated for them `Axiom1, Axiom2`.... They must then be conjoined to the schema which refer to them, as in the assembly case study in Chapter 4. Their interpretation is the same as for schemas,

Generic definitions are treated in the same way as the parametrised definitions of partial function etc, ie by using parameters `a,b..` They are instantiated when the set is instantiated, and are defined by a predicate in the LP, as for Sequence in Appendix C.