Implementing Z in Lego

Savitri Maharaj

M.Sc. Degree

Department of Computer Science
University of Edinburgh.

September 1990

Table of Contents

Abstract

The formal notation known as Z has generated widespread interest within the field of software engineering and has been used for the specification of a varied selection of computer-based systems. While Z offers a precise mathematical notation for the specification of systems and the formulation of theorems regarding the behaviour of these systems, proving these theorems is generally a matter of intuition or of traditional pen and paper methods of theorem proving.

Lego provides a proof-checking environment wherein mathematical structures can be modelled and theorems proved with confidence that the proofs are logically valid within the chosen model. The aim of this project is to complement the Z notation with the proof-checking capabilities of Lego by implementing Z within Lego.

The first step is to translate into Lego a selection of the mathematical concepts most frequently used in Z specifications. Thus the various methods of representing sets, functions, relations and numbers in Lego are examined and particular ones chosen for implementation. These are then implemented and a number of theorems about the resulting models are proven. These provide a library of rules for use in Z proofs.

A Z specification example is then translated into Lego by means of this implementation. Two theorems about the specification are formulated, expressed in Lego and then proved by the use of the library of definitions and rules and Lego's own proof-checking capacity.

Acknowledgements

Sincere thanks must go first of all to my supervisors, Rod Burstall and Claire Jones, for their invaluable guidance and suggestions, and to all the members of the Lego club for their constant willingness to offer help, explanations, and moral support.

Thanks also to the British Council who have made it possible for me to attend this course.

Finally I would like to offer my appreciation to the M.Sc. class of 1989-90 and to all the residents and visitors at 11/5 New Arthur Place, for their friendship during this year together.

Chapter 1

Introduction to the Project

1.1 Introduction

Formal methods of system specification rely on the established soundness of their underlying mathematics for their worth as tools in the development of software. This underlying rigour, however, can only be taken advantage of through the formulation and the proof of theorems about specifications using rigorous mathematical methods of proof. Proving such theorems by hand is difficult, tedious, and prone to error.

Proof checkers, such as Lego, can be used to relieve the tedium and increase the reliability of these proofs. The aim of this project was to implement the formal notation 'Z' within Lego, and then to use the implementation to translate a Z specification of a system and then prove some results about this specification.

This chapter gives some background information about the project, about Z, and about Lego. It also provides an overview of the activities engaged in during the course of the project, and a summary description of the rest of this report.

1.2 Original Project Outline

Title

Implementing Z in Lego

Description

- Translate into Lego one of the specification case studies in [1]. Possible ones would be "Unix file system" or "CAVIAR".
- Formulate some theorems about this specification, both in Z and in Lego, and then prove them in Lego.
- Translate a selection of the more useful concepts in the Z library into Lego.
- Discuss the general principles for translation from Z to Lego, based on the example you have carried out.

1.3 What is Z?

Z is a formal specification language based on set theory and first-order predicate logic. It was originally introduced by Jean-Raymond Abrial at Oxford University and has been developed there over the past few years. It has been used to specify a wide number of computer systems including the UNIX file system, the ICL Data Dictionary system, and a simple assembler.

Two complementary languages make up the Z notation: there is a *schema language* which allows the structured, systematic presentation of large-scale system specifications written in the *mathematical language*.

The mathematical language uses the notations of set theory and first-order predicate logic. The portions of this language which are referred to in this report are described in detail in appendix A.

The schema language addresses the problem of the modularisation of the mathematical description. A schema groups together some declarations of variables and a predicate which relates these variables. Schemas are used to represent identifiable modular components of systems. The *schema calculus* (described in appendix B) provides a large number of different ways to combine these modules to build a specification.

1.4 What is Lego?

An excellent introduction to the Lego system can be found in [13], from which the following description is adapted:

Lego is an interactive proof development system (proof checker) designed and implemented by R. Pollack in Edinburgh. It implements a number of related type systems—the Edinburgh Logical Framework [17], the Calculus of Constructions [10], the Generalised Calculus of Constructions [11] and the Extended Calculus of Constructions [12]. An overview of the basic theory and implementation of Lego may be found in [14].

Lego gives a powerful tool for interactive proof development in a style of natural deduction. It supports *refinement proof* as a basic operation. Features such as argument synthesis and universe polymorphism make proof checking more practical by bringing the level of formalization closer to that of informal mathematics. The higher-order power of its underlying type theories provides an expressive formalism for the expression of mathematical problems. Particularly, the type universes in the type theory make it possible to formalize abstract mathematics, and the strong sum types (Σ-types) can be used to naturally express abstract structures, mathematical theories, and program specifications.

The glossary in appendix C contains explanations of the features of Lego syntax that appear in this report.

1.5 Implementing Z in Lego

The purpose of this project was to complement the Z notation with the proof-checking ability of Lego by implementing Z within Lego. While an implementation of the full Z notation in Lego was considered, even at the outset, to be too ambitious a goal, it was hoped that enough of would be implemented to allow the translation of a specification case study from Z to Lego. It was then planned to formulate some theorems about this specification and prove them within Lego.

The original project specification suggested that the first step should be the translation of a particular Z example, followed by a generalisation to include a wider collection of Z features. It was quickly discovered that even the smallest Z schemas made use of a large selection of Z's mathematical notation, so it was decided to reverse the order of these two steps. The plan that was adopted therefore consisted of the following steps.

- Translate a library of mathematical notation into Lego. This library should contain the most frequently used portions of Z's mathematical language, including set theory, relations and functions. The natural numbers and the integers, which are datatypes which are assumed to exist in Z, must also be represented in some way in Lego.

- Find some representation in Lego of schemas and of a sufficiently large portion of the schema calculus to allow the translation of a Z case study.

- Use the library of notation to translate a Z specification into Lego.
- Formulate some theorems about this specification and prove them in Lego.

1.6 Overview of Activities

Before beginning the main work of this project, it was necessary for the author to spend some time learning about Z since she had no previous familiarity with this notation.

Going along with the original plan, the author then attempted to translate the "CAVIAR" specification [3]. However it was soon decided to defer this translation until a library of Z notation had been translated into Lego.

Since Z is based on set theory, it was natural to begin by expressing this theory in Lego. A number of different representations of sets in Lego were considered, and it was decided to use one of the simpler methods since no need for any of the more sophisticated representations was perceived at the time. All of the set-theoretic operations used by Z were translated into Lego using this model of a set (see Chapter 4).

Next the representation of numbers in Lego was dealt with. The Church technique (see Chapter 7) was used for representing the natural numbers. No method of representing integers in Lego was known, so the author attempted to devise her own technique. About three weeks was spent on this and there was some limited success. Finally it was necessary to abandon the problem for the moment because of the lack of time.

The next thing that was tackled was the representation of relations over sets. The definitions used by Z were implemented directly in Lego, and this was quite straightforward. Most of the relational operators used by Z have been implemented in Lego.

The representation of functions was then dealt with. Again it was possible to implement the Z definitions of functions and functional operators directly in Lego.

The mathematical notation that had been translated at this point was thought to be sufficient to attempt the translation of some Z schemas. First it was necessary to decide upon a representation of schemas in Lego. The most obvious method was to use Lego's Σ-types and this is the representation that was used.

The translation of the general resource-user system (a subsystem of "CAVIAR") was resumed. A problem was immediately encountered—the representation of functions that had been adopted did not lend itself easily to function application. A different method of representing functions was devised and used instead.

With these new functions it was then possible to express in Lego all of the schemas which make up the specification of the general resource-user system. This was done, and two theorems about the specification were formulated. These theorems were expressed in Lego as goals to be proven.

At this late stage, it was discovered that the first of these theorems could not be proven because of an inherent inflexibility in the method of representing sets. A more sophisticated representation of sets was therefore devised and adopted, and the whole of the library of mathematical notation was adjusted so that it was consistent with the new kind of set.

With these new sets, both of the theorems about the general resource-user system were proved in Lego. At this point, the original project specification was considered to have been satisfied.

1.7 The Rest of This Report

The remainder of this report describes in detail the work that has been done in implementing Z in Lego. Each chapter concerns itself with some facet of the implementation.

Chapter 2 discusses the issue of logic. A comparison is made of the logics available in Z and in Lego for use in proofs. Some extra definitions which supplement the predefined 'Logic' file in Lego are presented.

Chapter 3 presents a number of preliminary definitions which turned out to be necessary or convenient. These include the definition of Church-style booleans and of equivalence relations over Lego types.

Chapter 4 discusses the representation of set theory in Lego. Different methods of modelling sets are discussed and then a full implementation of set theory in Lego is presented.

Chapter 5 deals with the representation in Lego of relations between sets. An implementation of relations and of most of the relational operators used by Z is presented.

Chapter 6 is concerned with the modelling in Lego of functions from one set to another. A direct implementation of Z's definition of a function is presented and its shortcomings are discussed. The model which was eventually used is then exhibited.

Chapter 7 talks about representing numbers in Lego. A representation of natural numbers, and an attempt at representing integers are presented.

Chapter 8 discusses the representation of schemas in Lego. A particular method of doing this is illustrated by a small example.

Chapter 9 presents an example of the translation of a Z specification from Z into Lego. The CAVIAR system, and in particular the "general resource-user system" whose specification was developed as an initial stage in the analysis of CAVIAR, are described. A bilingual version of the specification of the general resource-user system is presented. Two theorems about this specification are stated (in Z and in Lego) and the methods by which they have been proven in Lego are outlined.

Chapter 10 states some general observations, conclusions, and speculations about this project.

Chapter 2

Logic

2.1 Introduction

This chapter begins with a comparison of the logical calculi provided by Z and by Lego for use in proving results. It then presents some additions to the predefined logical constructors in Lego which have been necessary in order to facilitate the definitions and proofs presented in the following chapters.

2.2 Z logic and Lego logic

Z is based on first order classical predicate calculus. An informal description of Z's logical basis can be found in [1]. The Lego system provides a higher order intuitionist predicate calculus [9,10,12,13] This means that any logical formula that can be constructed using Z notation can also be expressed in Lego. However there is one major difference in the tools available for proving things in the two systems. The difference arises because whereas Z uses classical logic, the Lego system provides an intuitionistic logic. This means that the axiom of the excluded middle is not intrinsically available in Lego. It is of course possible to add this axiom to the environment in Lego as an assumption. However for the purposes of this project the inclusion of this axiom was not found to be necessary.

2.3 Extensions to Lego logic

Upon entering the Lego environment it is customary to load a file titled 'Logic' which contains a number of useful definitions of traditional logical constructors such as 'and', 'or', etc. This predefined logic was found to contain all of the constructors used in Z in building logical formulae, with two exceptions. Definitions of these two constructors were therefore added to the system.

Unique existence

Z uses the constructor '∃!' to assert the existence of a unique object with some property. This constructor was defined in Lego.

```
[UEx [t|Type] [P:t->Prop] =
              Ex [x:t] ( P x) /\ ({y:t} (P y) -> (Q x y))]
```

In reading this definition, the square brackets [...] should be read as standing for the λ-representation of functions. The curly brackets $\{\ldots\}$ represent ordinary dependent Π-types and, in this case, should be read as "for all". `Q` stands for Leibnitz equality whose definition is given in appendix C. The above definition should therefore be read as:

$$UEx = \lambda\, t : \mathit{Type}\ \lambda\, P : (t \rightarrow \mathit{Prop})\ \exists\, x : t \cdot (Px) \wedge (\forall\, y : t \cdot (Py) \rightarrow (x = y))$$

Conditionals

If p, q, and r are propositions then the Z formula $p \rightarrow q, r$ means "if p then q else r". A similar constructor was defined in Lego:

```
[cond [t|Type] [p,q,r:Prop] =  (p -> q) /\ ((not p) -> r)]
```

2.4 Summary

Z bases its logical calculus on first order classical predicate logic. Lego provides a higher order intuitionist predicate logic, which has so far been found to be sufficient for the Z proofs that have been attempted.

The 'Logic' file which is usually loaded on entering the Lego environment contains definitions of most of the constructors used in the Z logical calculus. Two missing constructors—unique existence and conditional implication—have been added.

Chapter 3

Preliminary definitions

3.1 Introduction

This chapter discusses a number of preliminary definitions which were made in Lego in order to facilitate the work in the chapters that follow. The first section describes the implementation of Church-style booleans, which though not essential, made it possible to define the predecessor operation on the natural numbers in a more readable fashion (see Chapter 7). The second section gives the definition of equivalence relations on Lego types. These were necessary for the implementation of set theory that is given in Chapter 4.

3.2 Church-style booleans

The Church trick in defining an object, deriving from Church's representation of the natural numbers in the lambda calculus, is to represent an object by its eliminator function. Church style booleans are an example of this. The booleans are representations of the values 'true' and 'false'; they are expressed as functions which each select one of their two arguments. 'True' is the function which returns its first argument while 'false' returns its second. These booleans are then used in the definition of an 'if' statement which will be used in Chapter 7 in defining the predecessor of a natural number. A 'toggle' operator is also defined which returns the boolean complementary to its argument.

The definitions are as follows:

```
[bool = {t|Prop} t->t->t]

[true = [t|Prop] [f,g:t] f:bool ]
[false = [t|Prop] [f,g:t] g:bool]

[if = [t|Prop] [b:bool] [f,g:t] b f g]

[toggle [s:bool] = s false true:bool]
```

3.3 Equivalences

The usual representation of a relation in Lego (which is different from the relations which will be defined in Chapter 5) is as a function of the type `t->t->Prop` where `t` is some given type. Here the properties of being reflexive, symmetric and transitive are stated for these relations. These are then used to define a type `EQ` of equivalence relations over a given type. Inhabitants of this type will be referred to as 'equivalences'.

These equivalences were used to represent 'equalities' over particular types. This was important in the representation of set theory in Lego; this will be explained in further detail in the next chapter.

```
[Reflexive [t|Type] [R:t->t->Prop] = {x:t} R x x]

[Symmetric [t|Type] [R:t->t->Prop] =
                                {x,y:t} (R x y) -> (R y x)]

[Transitive [t|Type] [R:t->t->Prop] =
                     {x,y,z:t} (R x y) -> (R y z) -> R x z]

[Equivalence [t|Type] [R:t->t->Prop] =
```

```
            (Reflexive R) /\ (Symmetric R) /\ (Transitive R)]

[EQ [t:Type] = <R:t->t->Prop> Equivalence R]
```

Leibnitz equality (Q)[1] is then paired with a proof that it is an equivalence relation to produce an element `Q_EQ` of type `EQ`. The fragment `...Q_is_equiv` below means that the proof of the goal that precedes it has been *saved* in Lego as a theorem named `Q_is_equiv`. Where such theorems are referred to by name later on their names have been given in this manner.

```
Goal {t:Type} Equivalence (Q|t)
...Q_is_equiv

[Q_EQ [t:Type] = ((Q|t), (Q_is_equiv t)) : EQ t]
```

An operator `pair_EQ` which, when given two equivalences over types `s` and `t`, returns an equivalence over the type `s#t` is defined here.

```
Goal {s,t|Type} {EQ1 :EQ s} {EQ2 :EQ t}
      Equivalence [p,q:s#t] (EQ1.1 p.1 q.1) /\ (EQ2.1 p.2 q.2)
...pair_EQ_aux

[pair_EQ [s,t|Type] [EQ1:EQ s] [EQ2:EQ t] =
    ( ([p,q:s#t] (EQ1.1 p.1 q.1) /\ (EQ2.1 p.2 q.2)),
      (pair_EQ_aux EQ1 EQ2)):EQ s#t]
```

3.4 Summary

Some preliminary definitions have been made which have proved to be useful in the work that follows. The objects defined are i. a class of booleans 'true' and 'false' and ii. the type of equivalence relations over a given type.

---

[1] This is defined as `Q [A|Type] [x,y:A] = P:A->Prop(P x)->(P y)`

Chapter 4

Sets

4.1 Introduction

This chapter deals with the translation of set theory into Lego. It is assumed that the reader is familiar with the basics of set theory. The chapter begins with an explanation of the importance of this theory to Z. This is followed by a discussion of the known methods of representing sets in Lego, and the reasons for chosing the representation which has been implemented. This implementation is then exhibited and discussed.

4.2 Z and Set Theory

One might say that Z is set theory! All of the mathematical objects which are used in Z as the building blocks of specifications — functions, relations, etc — have been defined in terms of sets. The axioms of Set theory are used to prove results about these specifications. It would appear therefore that a faithful implementation of Z should begin with a suitable representation of sets and then use this as a basis for defining all the other objects in Z.

4.3 Many Kinds of Sets

The problem of representing sets in Lego is of widespread interest and has been much discussed [8]. The following four methods were considered.

- Sets as types. This is the simplest method. The built-in types of Lego are used as sets.
- Sets as predicates `t -> Prop`, where `t` is some type.
- Representations which combine a predicate of the above type with an equivalence relation defined over `t`.
- Representations in which the predicate and the equivalence relation are combined to form a partial equivalence relation.

This first of these methods is too weak for the purposes of Z. The second method, though also simple, seemed to be powerful enough to express all of the set-theoretic operations used in Z, so it was decided to experiment first with this representation, and to switch to one of the more complicated methods only if a need for this was eventually perceived. At first it seemed that these "naive sets" were sufficient for the purposes of the project. It was possible to express all the definitions regarding sets, relations, and functions in terms of these sets, and even to use these defintions to translate some Z schemas.

However problems arose eventually because of the notion of equality used by these sets in determining membership — an object can be proven to be an element of one of these sets only if it is syntactically identical (or convertible) to some given element of the set. This idea of equality is sometimes too strong. In particular, in the case where one needs to prove that a given set is an element of a given set of sets, it would be preferable to check for membership by using the extensional equality of sets, rather than strict syntactic equality. For example, the set $\{0,1\}$ cannot be proven to be an element of the set $\{\{1,0\}\}$ using syntactic equality to determine membership, whereas with extensional equality the proof is possible.

This problem does not arise with the third definition of a set, since this includes as one of its parameters the kind of equality relation to be used upon the members of the set. Hence this definition was eventually adopted. The details are presented in the following section. As far as this author knows, this particular method of representing a set in Lego has never before been implemented.

4.4 An Implementation of Set Theory

A set is defined over a type `t` and an equivalence `R` on the inhabitants of `t` as a predicate of the type `t->Prop`. The equivalence `R` is used to determine membership of the set.

Basic definitions

The following is the definition of a set over a type `t` and an equivalence `R`.

```
[set [t|Type] [R:EQ t] = (t->Prop)]
```

The equivalence relation over which the set is defined is used to determine membership of the set. This is the definition.

```
[member [t|Type] [R|EQ t] [x:t] [S:set R] =
                              Ex[y:t](S y) /\ (R.1 x y)]
```

For convenience, the type of members of a given set is defined using sigma types. This makes it possible to quantify over the elements of a set.

```
[set_el [t|Type] [R|EQ t] [S:set R] = <x:t> member x S]
```

The following result regarding set membership has proven to be useful in proofs:

```
Goal {t|Type} {R|EQ t} {S:set R} {x,y:t}
                    (R.1 x y)->(member x S)->member y S
```

The empty set, complements, and the universal set

The empty set, complements of sets and the universal set are defined:

```
[void [t|Type] [R:EQ t] = [x:t] absurd :set R]

[compl [t|Type] [R|EQ t] [S:set R] =
                  [x:t] not(member x S):set R]

[universe [t|Type] [R:EQ t] = compl (void R)]
```

Some simple, but convenient, results are proven:

```
Goal {t|Type} {x:t} {R:EQ t} not (member x (void R))

Goal {t|Type} {x:t} {R:EQ t} (member x (universe R))
```

Set inclusion

Set inclusion is defined and is shown to be reflexive and transitive:

```
[subset [t|Type] [R|EQ t] [S,T:set R] =
                 {x:t}((member x S)->(member x T))]

Goal {t|Type} {R|EQ t} {S:set R} subset S S

Goal {t|Type} {R|EQ t} {S,T,U:set R}
             (subset S T)->(subset T U)->subset S U
```

Strict set inclusion (proper subset) is also defined:

```
[ p_subset [t|Type] [R|EQ t] [S,T:set R] =
      (subset S T) /\
      (Ex[x:t] (member x T) /\ (not (member x S)))]
```

Equality of sets

Leibnitz equality is too strong for these sets, so a special set equality is defined. The equality used is really extensionality applied to sets, ie. two sets are equal if they have the same members. The following definition is equivalent (ie. convertible in Lego) to this.

In the remainder of this report, whenever two sets are said to be "equal" this means "equal under `QSet`".

```
 [QSet [t|Type] [R|EQ t] [S,T:set R] =
                       (subset S T) /\ (subset T S)]
```

`QSet` is shown to be reflexive, symmetric and transitive.

```
Goal{t|Type} {R|EQ t} {S|set R} QSet S S

Goal{t|Type} {R|EQ t} {S,T|set R} (QSet S T)->(QSet T S)

Goal{t|Type} {R|EQ t} {S,T,U|set R}
                         (QSet S T)->(QSet T U)->QSet S U
```

`QSet` is thence proven to be an equivalence relation and the equivalence `QSet_EQ` is defined:

```
Goal {t|Type} {R|EQ t} Equivalence (QSet|t|R)

[QSet_EQ [t|Type] [R:EQ t] =
             ((QSet|t|R),(QSet_is_equiv|t|R)):EQ (set R)]
```

Membership and complement are shown to respect `QSet`:

```
Goal {t|Type} {R|EQ t} {S,T|set R} {x|t}
               (QSet S T) -> (member x S) -> (member x T)

Goal {t:Type} {R|EQ t} {S,T:set R}
                   (QSet S T) -> QSet (compl S) (compl T)
```

Set intersection

This is defined in the following way:

```
[intersection [t|Type] [R|EQ t] [S,T:set R] =
                  [x:t] (member x S)/\(member x T):set R]
```

The intersection of two sets is shown to be a subset of either set:

```
Goal {t|Type} {R|EQ t} {S,T:set R}
                        (and (subset (intersection S T) S)
                             (subset (intersection S T) T))
```

Intersection is shown to be commutative and associative and to respect `QSet`. Here one of the disadvantages of using the present notion of a set instead of one of the simpler ones becomes obvious—the proofs are long and difficult since they involve much tedious manipulation of the equivalence relation.

```
Goal {t|Type} {R|EQ t} {S,T:set R}
              (QSet (intersection S T) (intersection T S))

Goal {t|Type} {R|EQ t} {S,T,U:set R}
                 (QSet (intersection S (intersection T U))
                      (intersection (intersection S T) U))

Goal {t|Type} {R|EQ t} {S,T,U,V:set R}
          (QSet S U)->(QSet T V) ->
             (QSet (intersection S T) (intersection  U V))
```

The intersection of `void` with any other set is shown to be equal to `void`:

```
Goal {t|Type} {R|EQ t} {S:set R}
                   QSet (void R) (intersection (void R) S)
```

It is shown that a set is equal to its intersection with the universal set:

```
Goal {t|Type} {R|EQ t} {S:set R}
                      QSet S (intersection (universe R) S)
```

Set Union

The union of two sets is defined:

```
[union [t|Type] [R|EQ t] [S,T:set R] =
                [x:t] (member x S) \/ (member x T) :set R]
```

A set is shown to be a subset of its union with any other set:

```
Goal {t|Type} {R|EQ t} {S,T:set R}
                               (and (subset S (union S T))
                                    (subset T (union S T)))
```

Union is shown to be commutative and associative and to respect `QSet`. These proofs are quite long, the associativity proof in particular requiring about 150 steps.

```
Goal {t|Type} {R|EQ t} {S,T:set R}
                            (QSet (union S T) (union T S))

Goal {t|Type} {R|EQ t} {S,T,U:set R}
                            (QSet (union S (union T U))
                                  (union (union S T) U))

Goal {t|Type}{R|EQ t} {S,T,U,V:set R}
                    (QSet S U) -> (QSet T V) ->
                         (QSet (union S T) (union  U V))
```

A set is shown to be equal to its union with the empty set:

```
Goal {t|Type} {R|EQ t} {S:set R} QSet S (union S (void R))
```

It is shown that the union of a set with the universal set is equal to the universal set:

```
Goal {t|Type} {R|EQ t} {S:set R}
                     QSet (universe R) (union S (universe R))
```

Power sets

The power set of a set is defined. For convenience, the type of elements of the powerset of a given set is explicitly defined:

```
[powerset [t|Type] [R|EQ t] [S:set R] =
                   [T:set R ] (subset T S) : set (QSet_EQ R)]

[pset [t|Type] [R|EQ t] [S:set R] =
                           <T:set R> (member T (powerset S))]
```

The empty set is shown to be an element of the powerset of any set:

```
Goal {t|Type} {R|EQ t} {S:set R} member (void R) (powerset S)
```

The subsets of a set are shown to be members of its powerset. Though this may seem obvious, it is a useful result to have because it saves a number of proof steps involving the manipulation of the definition of set membership.

```
Goal {t|Type} {R|EQ t} {S,T|set R}
                         (subset S T) -> member S (powerset T)
```

Distributed intersection and union

Distributed set intersection and union are defined:

```
[F_intersection [t|Type] [R|EQ t] [F:set|(set R) (QSet_EQ R)]
     = [x:t] {P:set R} ((member P F) -> member x P ) : set R]

[F_union [t|Type] [R|EQ t] [F: set|(set R) (QSet_EQ R)]
    = [x:t] Ex[P:set R] (member P F) /\ (member x P) : set R]
```

Set difference

Set difference is defined. Some simple but useful results are proven:

```
[set_diff [t|Type] [R|EQ t] [S,T: set R] =
            [x:t] (member x S ) /\ (not (member x T)) : set R]

Goal {t|Type} {R|EQ t} {S:set R} QSet (void R) (set_diff S S)

Goal {t|Type} {R|EQ t} {S,T:set R} subset ( set_diff S T ) S
```

The next proof follows directly from the one above but has been included because of its special usefulness in creating some of the definitions in the next chapter.

```
Goal {t|Type} {R|EQ t} {S,T:set R}
                            member (set_diff S T) (powerset S)
... Set_diff_in_pset
```

Singletons

The singleton containing a given element is defined.

```
[singleton [t|Type] [R:EQ t] [x:t] = [y:t] (R.1 y x): set R]
```

`Get_el` extracts the element from a given singleton:

```
[get_el [t|Type] [R|EQ t] [x|t]
        [S:set R] [H|QSet S (singleton R x)] = x]
```

The following result about singletons was proven:

```
Goal {S|set E1} {x:set_el S} subset (singleton E1 x.1) S
```

Cross products of Sets

The cross (Cartesian) product of two sets is defined:

```
[cross_product [s,t|Type] [R1|EQ s] [R2|EQ t]
               [X:set R1] [Y:set R2] =
[p:s#t](and(member p.1 X)(member p.2 Y)):set (pair_EQ R1 R2)]
```

Finite Sets

Finite sets are defined inductively using a fixed-point technique. First the operation of adding an element to a set is defined:

```
[add_el [t|Type] [R|EQ t] [S:set R] [x:t] =
                       [y:t] (member y S) \/ (R.1 y x) :set R]
```

The property of being finite, and then the type `fset` of finite sets, are defined:

```
[finite_attr [t|Type] [R|EQ t] [P:(set R) -> Prop] =
  and (P (void R) )
      ({S:set R} {x:t} (P S) ->
                       (not (member x S)) -> (P(add_el S x)))]

[finite [t|Type] [R|EQ t] [S:set R] =
                 {P:(set R) ->Prop} (finite_attr P) -> (P S)]

[fset [t|Type] [R|EQ t] = <S:set R> finite S]
```

Some immediate results are proven:

```
Goal {t:Type} finite (void t)

Goal {t|Type} {S:f_set t} {x:t} (not (member x S)) ->
                                        finite (add_el S.1 x)
```

Decidable sets

Decidable sets are those which have the following property:

```
[dec [t|Type] [E|EQ t] [S:set E] =
                 {x:t} (or (member x S) (not( member x S)))]
```

The type `dset` of decidable sets is defined:

```
[dset [t|Type] [E:EQ t] = <S:set E>(dec S)]
```

Decidable sets have the property that it is possible to tell whether or not any given inhabitant of the type over which the set is based is actually in the set. Now all sets in Z are decidable, since Z uses classical logic and therefore allows the axiom of the excluded middle. This observation led to an attempt to use `dsets` as the basic building block of the system.

An attempt was made to redefine all of the set theory in terms of `dsets`. The first few definitions— `dmember`, `dset_el`, `dvoid`, `dsubset`, `dcompl`, `duniverse`, equality, `dintersection`, and `dunion`—and the accompanying proofs were straightforward. In all these cases the corresponding definition for `sets`, paired if necessary with a proof of the decidability of the object being defined, was sufficient. Problems arose however when trying to define the power set of a decidable set, since this power set cannot in general be shown to be decidable. Hence the full set theory required by Z could not be developed if the notion of set were restricted to decidable sets alone. It was decided therefore to continue working with `sets` and to appeal to `dsets` only when decidability was specifically required.

4.5 Discussion

The method of representing sets presented in this chapter was chosen because of the necessity to specify the kind of equality relation to be used on the members of a given set. Neither of the two simpler versions of sets in Lego allowed this kind of parameterised equality. This method has been found to work satisfactorily as a basis for definitions of relations, functions, and some Z schemas (see following chapters), and for a large number of proofs about these objects. Its main drawback is that certain proofs which are quite straightforward if one of the simpler versions of sets is used, are now made quite long and complicated because of the necessity to manipulate the equality relation supplied with the set by appealing to its properties of reflexivity, symmetry, and transitivity. Of course, certain other proofs which are quite impossible with the simpler kinds of sets are now made possible with this version.

4.6 Summary

Four distinct methods of representing sets in Lego are known. Of these, the first two are not sufficiently powerful to meet the requirements of Z. The third version, in which a set over type `t` is represented by a function of type `t->Prop` which also takes as a parameter an equivalence `EQ` over the inhabitants of type `t`, has therefore been implemented.

All of the set-theoretic operators used by Z have been defined over this representation of sets, and a large number of results have been proven. The definitions include representations of finite sets and of decidable sets.

This method of representing sets has been found to complicate proofs which were quite straightforward with simpler versions of sets. However the use of a representation at least this complex is unavoidable if certain necessary kinds of proof are to be made possible.

Chapter 5

Relations

5.1 Introduction

This chapter deals with the representation in Lego of the idea of a relation between two sets. It begins with a discussion of why relations are important in Z and how they are there defined. This is followed by an argument for choosing a particular method of implementing relations in Lego which is then presented in detail.

5.2 Relations in Z

The mathematical concept of a relation between two sets is of fundamental importance in Z. Systems specified using Z are often modelled by a combination of sets and relations (or restricted kinds of relations, such as partial functions) among these sets, together with a boolean formula giving some necessary constraint. For example, a schema describing a simple symbol table, as used by programming language compilers, might consist of a partial function relating a set of symbols to a set of values.

Z uses sets as the building blocks for its relations. The set of relations between two sets X and Y is defined as the power set of the Cartesian product of X and Y. This feature allows the Z user to use all the operators defined for sets on relations as well.

5.3 Representing Z relations in Lego

It seems reasonable to expect that the fact that all Z definitions are built up upon the mathematically well-founded set theory must play an important role in any argument for the logical consistency of Z. A faithful implementation of Z in Lego should likewise base its notion of relations upon its set theory. The definitions exhibited in the following section were devised with this in mind.

5.4 An Implementation of Relations

The definitions presented here are intended to be as close as possible to the corresponding definitions in Z. The Z definitions are presented in appendix A.

5.4.1 Basic definitions

First an environment is set up in which to build the definitions:

```
[t1,t2,t3|Type] [E1|EQ s] [E2|EQ t] [E3|EQ u]
```

At appropriate points the types `t1`, `t2`, and `t3` and equivalences `E1`, `E2` and `E3` are discharged so that the definitions and rules are parameterised over these types.

Relations as sets and as types

The set of relations between two given sets is defined:

```
[relations [X:set E1] [Y:set E2] =
                       powerset ( cross_product X Y )]
```

For convenience of use, sigma types are used to define the corresponding type:

```
[rel [X:set E1] [Y:set E2] = set_el (relations X Y)]
```

The predicate `is_relation` will be useful in future definitions:

```
[is_relation [X:set R1] [Y:set R2] [R :set (pair_EQ R1 R2)] =
                                     member R (relations X Y)];
```

Relatedness

To help preserve syntactic similarity to Z, relatedness is defined explicitly:

```
[is_related [X|set E1] [Y|set E2] [R: rel X Y] [x:t1] [y:t2]
                                       = member (x,y) R.1]
```

The empty relation

The empty relation between two sets is represented by pairing the empty set of the appropriate type with a proof that this set is a relation between the two sets:

```
Goal {X:set E1} {Y:set E2}
                     is_relation X Y (void (pair_EQ E1 E2))
...void_is_relation

[void_rel [X:set E1] [Y:set E2] =
  ((void (pair_EQ E1 E2)), (void_is_relation X Y)):rel X Y]
```

5.4.2 Relational operators

Domains and ranges

The domain and the range of a relation are defined:

```
[dom [X|set E1] [Y|set E2] [R:rel X Y] =
              [x:t1] Ex [y:t2] is_related R x y : set E1]

[rng [X|set E1] [Y|set E2] [R:rel X Y] =
              [y:t2] Ex [x:t1] is_related R x y : set E2]
```

The domain and range of a relation are shown to be subsets of the sets over which the relation is defined:

```
Goal {X:set E1} {Y:set E2} {R:rel X Y}
                            subset (dom R) X
... Save dom_is_subset

Goal {X:set E1} {Y:set E2} {R:rel X Y}
                            subset (rng R) X
... Save rng_is_subset
```

The domain and range of the empty relation are shown to be equal to the empty set of the appropriate type:

```
Goal {X:set E1} {Y:set E2}
     {x:t1} not (member x (dom (void_rel X Y)))

Goal {X:set E1} {Y:set E2}
     {y:t2} not (member y (rng (void_rel X Y)))
```

Subrelations

Inclusion of relations is defined:

```
[subrel [X|set E1] [Y|set E2] [R1,R2:rel X Y]
                               = subset R1.1 R2.1]
```

Composition of relations

Forward composition of relations is defined. First an auxilliary definition is required:

```
[fcomp_aux [X|set E1] [Y|set E2] [Z|set E3]
           [R1:rel X Y] [R2:rel Y Z]
= [p:t1#t3] Ex [y:t2] (is_related R1 p.1 y) /\
                  (is_related R2 y p.2): set (pair_EQ E1 E3)]
```

The set so defined is then paired with a proof that it is a relation to complete the definition:

```
Goal {X|set E1} {Y|set E2} {Z|set E3} {R1:rel X Y} {R2:rel Y Z}
                                is_relation X Z (fcomp_aux R1 R2)
   ... Save fcomp_aux_is_rel

   [fcomp [X|set E1][Y|set E2][Z|set E3][R1:rel X Y] [R2:rel Y Z]
        = ((fcomp_aux R1 R2), (fcomp_aux_is_rel R1 R2)): rel X Z]
```

Relational composition is defined:

```
[comp [X|set E1] [Y|set E2] [Z|set E3]
                 [R1:rel Y Z] [R2:rel X Y] =  fcomp R2 R1]
```

Inverse relations

The inverse of a relation is defined. Again this is done by defining an auxilliary operation which produces a set, and then pairing this with a proof that the set is a relation.

```
[inv_aux [X|set E1] [Y|set E2] [R:rel X Y] =
          [p:t2#t1] is_related R p.2 p.1  :set (pair_EQ E2 E1)]

Goal {X|set E1} {Y|set E2}  {R:rel X Y}
                                is_relation Y X ( inv_aux R )
...inv_aux_is_rel

[inv [X|set E1] [Y|set E2] [R:rel X Y]  =
                 ((inv_aux R), (inv_aux_is_rel R)): rel Y X]
```

The domain of a relation is shown to be equal to the range of the inverse relation:

```
Goal {X|set E1} {Y|set E2} {R:rel X Y}
                                      QSet (dom R) (rng (inv R))
```

The identity relation

The identity relation on a given set is defined:

```
[id_aux [X:set E1] =
[p:t1#t1] (member p.1 X) /\ (E1.1 p.1 p.2): set (pair_EQ E1 E1)]

Goal {X:set E1} is_relation X X (id_aux X);
... Save id_aux_is_rel

[id [X:set E] = ((id_aux X), (id_aux_is_rel X)):rel X X]
```

Singleton relations

The mini-relation consisting of a single pair of related elements of some given sets is defined:

```
[s_r_aux [X|set E1] [Y|set E2] [x:set_el X] [y:set_el Y] =
                          singleton (pair_EQ E1 E2) (x.1,y.1)]

Goal{X|set E1}{Y|set E2}{x:set_el X} {y:set_el Y}
                                 is_relation X Y (s_r_aux x y)

[singleton_rel [X|set E1] [Y|set E2] [x:set_el X] [y:set_el Y]
                   = ((s_r_aux x y),(s_r_aux2 x y)) : rel X Y]
```

Relational images

The image of a set under a relation is defined:

```
[image [X|set E1] [Y|set E2] [R:rel X Y] [Z:pset X] =
[y:t2] Ex [x:t1] (member x Z.1) /\ (is_related R x y) :set E2 ]
```

The image of a particular element under a relation is also defined:

```
[apply [X|set E1] [Y|set E2] [R: rel X Y] [x:t1] =
                             [y:t2] is_related R x y : set E2 ]
```

Domain restriction and subtraction

Domain restriction to a given set, and subtraction of a given set from the domain, are defined:

```
[dr_aux [X|set E1] [Y|set E2] [R:rel X Y] [Z:pset X] =
[p:t1#t2] (member p.1 Z.1) /\ (member p R.1) :set (pair_EQ E1 E2)]

Goal {X|set E1} {Y|set E2} {R:rel X Y} {Z:pset X}
                                  is_relation X Y (dr_aux R Z)
...dr_aux_is_rel

[domain_res [X|set E1] [Y|set E2] [R:rel X Y] [Z:pset X] =
                  ((dr_aux R Z), (dr_aux_is_rel R Z)):rel X Y]

[domain_sub [X|set E1] [Y|set E2] [R:rel X Y] [Z:pset X] =
  domain_res R
 (((set_diff X Z.1),(Set_diff_in_pset X Z.1)):pset X) :rel X Y ]
```

The domain-restricted and domain-subtracted relations are shown to be subrelations of the original relation:

```
Goal {X|set E1} {Y|set E2} {R:rel X Y} {Z:pset X}
                               subrel (domain_res R Z) R

Goal {X|set E1} {Y|set E2} {R:rel X Y} {Z:pset X}
                               subrel (domain_sub R Z) R
```

Range restriction and subtraction

The definitions of range restriction and range subtraction are analogous to those of the corresponding operations regarding domains:

```
[rr_aux [X|set E1][Y|set E2][R:rel X Y][Z:pset Y] =
[p:t1#t2] (member p.2 Z.1) /\ (member p R.1) :set (pair_EQ E1 E2)]

Goal {X|set E1}{Y|set E2}{R:rel X Y}{Z:pset Y}
                              is_relation X Y (rr_aux R Z)
...rr_aux_is_rel

[range_res [X|set E1][Y|set E2] [R:rel X Y] [Z:pset Y] =
                   ((rr_aux R Z), (rr_aux_is_rel R Z)):rel X Y]

[range_sub [X|set E1][Y|set E2] [R:rel X Y] [Z:pset Y] =
  range_res R
(((set_diff Y Z.1),(Set_diff_in_pset Y Z.1)):pset Y) :rel X Y ]
```

These are also shown to be sub relations of the original relations:

```
Goal {X|set E1}{Y|set E2}{R:rel X Y}{Z:pset Y}
                                subrel (range_res R Z) R

Goal {X|set E1}{Y|set E2}{R:rel X Y}{Z:pset Y}
                                subrel (range_sub R Z) R
```

5.4.3 Set-theoretic operations

In Z, relations are sets, so that all set operators can be applied directly to them. The use of sigma types to define the type `rel X Y` makes it necessary to modify these operators slightly so that they can be used for relations. In general, the modified operator combines the original set operator with a proof that the result of applying this operator to the set components of relations is itself another relation.

Unions of relations

The relation obtained by taking the union of two relations is defined:

```
[rel_union_aux [X|set E1] [Y|set E2] [R1,R2:rel X Y] =
                                             union R1.1 R2.1]
```

```
Goal {X|set E1} {Y|set E2} {R1,R2:rel X Y}
                          is_relation X Y (rel_union_aux R1 R2)
...rel_union_is_rel

[rel_union [X|set E1] [Y|set E2] [R1,R2:rel X Y] =
     ((rel_union_aux R1 R2),(rel_union_is_rel R1 R2)): rel X Y]
```

It is shown that the domain, range, and inverse operators distribute over union:

```
Goal {X|set E1} {Y|set E2} {R1,R2:rel X Y}
  QSet (dom (rel_union R1 R2)) (union (dom R1) (dom R2))

Goal {X|set E1} {Y|set E2} {R1,R2:rel X Y}
  QSet (rng (rel_union R1 R2)) (union (rng R1) (rng R2))

Goal {X|set E1} {Y|set E2} {R1,R2|rel X Y}
  QRel (inv (rel_union R1 R2)) (rel_union (inv R1) (inv R2))
```

Intersections of relations

The intersection of two relations is defined:

```
[rel_intersect_aux [X|set E1] [Y|set E2] [R1,R2:rel X Y] =
                                        intersection R1.1 R2.1]

Goal {X|set E1} {Y|set E2} {R1,R2:rel X Y} is_relation X Y
                                      (rel_intersect_aux R1 R2)
...rel_intersect_is_rel

[rel_intersect [X|set E1] [Y|set E2] [R1,R2:rel X Y] =
((rel_intersect_aux R1 R2),
 (rel_intersect_is_rel R1 R2)): rel X Y]
```

The difference of two relations

The difference of two relations is defined:

```
[rel_diff_aux [X|set E1][Y|set E2][R1,R2:rel X Y]  =
```

```
                        set_diff R1.1 R2.1 :set (pair_EQ E1 E2)]

Goal {X|set E1} {Y|set E2} {R1,R2:rel X Y} is_relation X Y
                                          (rel_diff_aux R1 R2)
...rel_diff_is_rel

[rel_diff [X|set E1] [Y|set E2] [R1,R2:rel X Y] =
       ((rel_diff_aux R1 R2),(rel_diff_is_rel R1 R2)): rel X Y]
```

The domain and range of the difference of two relations are shown to be subsets of the first of the given relations:

```
Goal {X|set E1} {Y|set E2}{R1,R2:rel X Y}
                  subset (dom (rel_diff R1 R2)) (dom R1)

Goal {X|set E1} {Y|set E2}{R1,R2:rel X Y}
                  subset (rng (rel_diff R1 R2)) (rng R1)
```

Overriding

The relation obtained by allowing a given relation to override another can now be defined:

```
[over_ride [X|set E1][Y|set E2][R1,R2:rel X Y] =
   rel_union
     ((domain_sub
         R1
        (((dom R2),
          (Subset_in_pset(dom_is_subset X Y R2))):pset X)))
     R2  : rel X Y]
```

5.4.4 Equality of relations

An equality `QRel` on relations is defined simply as the equality of the corresponding sets and is shown to be reflexive, transitive, and symmetric.

```
[QRel = [X|set E1] [Y|set E2] [R1,R2:rel X Y] QSet R1.1 R2.1]

Goal {X|set E1} {Y|set E2} {R|rel X Y} QRel R R

Goal {X|set E1} {Y|set E2} {R1,R2|rel X Y}
                                  (QRel R1 R2) -> QRel R2 R1

Goal {X|set E1} {Y|set E2} {R1,R2,R3|rel X Y}
                (QRel R1 R2) ->  (QRel R2 R3) -> QRel R1 R3
```

QRel is proven to be an equivalence relation and the equivalence QRel_EQ is constructed:

```
Goal {X|set E1}{Y|set E2} Equivalence (QRel|s|t|E1|E2|X|Y)

[QRel_EQ [X:set E1] [Y:set E2] =
((QRel|s|t|E1|E2|X|Y),(QRel_equiv|s|t|E1|E2|X|Y)):EQ (rel X Y)]
```

QRel is shown to be respected by is_related, and by domains, ranges, inverses and differences:

```
Goal {X|set E1}{Y|set E2} {R1,R2|rel X Y} {x|s} {y|t}
 (QRel R1 R2) -> (is_related R1 x y) -> (is_related R2 x y)

Goal {X|set E1} {Y|set E2} {R1,R2|rel X Y}
                     (QRel R1 R2) -> QSet (dom R1) (dom R2)

Goal {X|set E1} {Y|set E2} {R1,R2|rel X Y}
                     (QRel R1 R2) -> QSet (rng R1) (rng R2)

Goal {X|set E1} {Y|set E2} {R1,R2|rel X Y}
                     (QRel R1 R2) -> QRel (inv R1) (inv R2)

Goal {X|set E1} {Y|set E2} {R1,R2,R3|rel X Y}
     (QRel R1 R2) -> QRel (rel_diff R1 R3) (rel_diff R2 R3)
```

5.5 Discussion

This implementation of relations in Lego is derived directly from the definitions used in Z. It worked satisfactorily in the examples which were attempted, but it is possible to anticipate some problems which might arise. The difficulty would appear to lie in getting at the results of the application of these relations to particular arguments. Though the set obtained as a result of this application is defined (`apply`), it is not clear how to get at an actual element of this set for use in a proof or in a definition.

5.6 Summary

A relation from one set `X` to another set `Y` has been defined as a subset of the cross product of `X` and `Y`. This is in strict adherence to the definition of a relation that is used in Z. Some of the set-theoretic operators of the previous chapter have had to be modified slightly so that they can be applied to these relations. Most of the relational operators used by Z have been defined. A number of useful results about the behaviour of these operators have been proven.

These relations have been found to work satisfactorily in all the schema examples that have been attempted. Possible problems with relational application, however, are anticipated.

Chapter 6

Functions

6.1 Introduction

This chapter concerns the representation in Lego of functions from one set to another. It starts with a discussion of the definition of functions that is actually used by Z. Two different implementations of functions in Lego are then exhibited and their various merits and failings are discussed.

6.2 Functions in Z

Like all else in Z, the idea of a function is based in set theory. Z builds the definition of a function upon the idea of a relation - a function is understood as a relation with the property that for each element in its domain there is a unique element in its range related to it. All operators on sets and relations can therefore also be used for functions.

6.3 A Direct Implementation of Z Functions

Given the observations in the section above, the obvious way to go about representing functions in Lego would seem to be to implement the Z definitions directly, building upon the previous definitions of sets and relations. This method was therefore the one first attempted.

All the definitions in this section are direct expressions in Lego of the corresponding Z definitions which are given in appendix A. The environment is assumed to be that set up in the previous chapter.

6.3.1 Partial functions

A partial function is defined as a relation with the special property that each element in its domain has a unique element in the range to which it is related. First the set `pfunctions` of partial functions over two given sets is defined:

```
[pfunctions  [X:set E1] [Y:set E2] = [f:rel X Y ] {x:s}
 ((member x (dom f)) ->
          UEx [y:t] (is_related f x y)): set (pair_EQ E1 E2)]
```

Sigma types are used to obtain a type from this set:

```
[pfunction [X:set E1] [Y:set E2] =
                <r:rel X Y> member r (pfunctions X Y)]
```

6.3.2 Total functions

Total functions are then defined as those partial functions which have the property that their domain is equal to the first of the sets over which the partial function is defined. As for partial functions, these are defined first as a set and then a type is derived from this set:

```
[tfunctions [X:set E1] [Y:set E2] =
                    [f:pfunction X Y ] QSet (dom f.1) X]

[tfunction [X:set E1] [Y:set E1] =
           <p:pfunction X Y> member p (tfunctions X Y)]
```

6.3.3 The problem—function application

At first this implementation of functions appeared to be ideal. Every kind of function used in Z could be expressed, and the definitions used were reassuringly close to the original Z definitions.

But problems arose when an attempt was made to use these functions in an actual Z schema. The schema being defined called for a total function to be applied to a given argument, and for the value obtained as a result to be known explicitly. This necessitated a definition of function application of the following type:

```
{X|set E1} {Y|set E2} {F:pfunction X Y} {x:s}
                                (member x (dom F.1)) -> t
```

Now apply has been defined for general relations - it returns the set of all values to which its argument is related under the given relation. This apply can also be used for functions. It is even possible to prove that in the case of a function the set that is returned is a singleton:

```
Goal {X|set E1} {Y|set E2} {F:pfunction X Y} {x:s}
   (member x (dom F.1)) ->
        Ex [y:t] (member y Y) /\
                 (QSet (apply F.1 x) (singleton y))
```

It would seem that all that remains is to use get_el to extract the element of the singleton, and the definition would be complete. But this is not so easy. The difficulty lies with the definition of the existential quantifier in Lego - there is no way to obtain an actual example of the entity whose existence is asserted.

Now this was a serious set-back since function application is a ubiquitous feature of Z schemas. How then should it be defined? One way would be by axiomatizing an `apply` with the desired properties, but proofs would then require the complicated and laborious manipulation of the axioms. Another tack would be to try to redefine the existential quantifier in Lego in some way which enabled witnesses to be obtained. This approach was not explored but bears examining.

Finally it was decided to use another representation of functions - one in which function application is easy. This is the subject of the next section.

6.4 Using Lego's Built-in Functions

Functions from one type to another are part of Lego. Application of these functions is easy - the function and its argument are simply juxtaposed. This section describes how these functions can be modified to model functions over sets.

6.4.1 Total functions

Because the schema example which was targeted for translation into Lego did not make use of partial functions, at first only total functions were represented. A possible way of generalising this definition to represent partial functions is discussed in the next section.

Basic definitions

A function f from set X with equality $=_X$ to set Y with equality $=_Y$ is said to be *well-defined* if whenever two elements of X are equal under $=_X$ then their images under f are equal under $=_Y$.

A total function is modelled by a sigma type whose components are a Lego function over two given types, a set over each of the given types, and a proof that the function (a) maps elements of the first set onto elements of the second and (b) is well-defined. This definition is based upon that of a *deliverable* which was introduced in [16].

Some basic types and equivalences are defined:

```
[s,t,u|Type] [E1|EQ1] [E2|EQ2] [E3|EQ3]
```

First the formula to be proved is stated:

```
[prop1 [X:set E1] [Y:set E2] [f:s->t] =
                {x:s} (member x X) -> (member (f x) Y)]

[prop2 [X:set E1] [Y:set E2] [f:s->t] =
{x,y:s} (member x X) -> (member y X) ->
                      (E1.1 x y) -> (E2.1 (f x) (f y))]
```

This, then, is the definition of a function:

```
[fun [X:set E1] [Y:set E2] = <f:s->t> (is_fun X Y f)]
```

Domains and ranges

The domain and range of functions are defined. The domain is easy since the functions are total.

```
[f_dom [X|set E1] [Y|set E2]  [F:fun X Y] = X]

[f_rng [X|set E1] [Y|set E2] [F:fun X Y] =
       [y:t] Ex [x:s] (member x X) /\ (E2.1 (F.1 x) y) :set E2]
```

Identity function

The definition of the identity function is straightforward:

```
[f_id_aux [X:set E1] = [x:s] x]

Goal {X:set E} is_fun X X (f_id_aux X)
...f_id_aux2

[f_id [X:set E] = ((f_id_aux X),(f_id_aux2 X)): fun X X]
```

Composition of functions

Composition of relations can also be defined:

```
[comp_aux [X|set E1] [Y|set E2] [Z|set E3]
          [F1:fun X Y] [F2:fun Y Z] = [x:s] F2.1 (F1.1 x )]

Goal{X|set E1}{Y|set E2}{Z|set E3} {F1:fun X Y} {F2:fun Y Z}
                                  is_fun X Z (comp_aux F1 F2)
...comp_aux2

[f_comp [X|set E1] [Y|set E2] [Z|set E3]
        [F1:fun X Y] [F2:fun Y Z] =
              ((comp_aux F1 F2),(comp_aux2 F1 F2)):fun X Z ]
```

Functional images

The image of a set under a function is defined.

```
[f_image [X|set E1] [Y|set E2] [F:fun X Y] [S:pset X] =
         [y:t] Ex [x:set_el S.1] (E2.1 (F.1 x.1) y) :set E2]
```

Domain restriction and subtraction

The domain restricted function is fairly easy to define since the original Lego function already has all the necessary properties:

```
Goal {X|set E1}{Y|set E2}{F:fun X Y} {Z:pset X}
                                             is_fun Z.1 Y F.1
... Save f_dr_aux
```

```
[f_domain_res [X|set E1] [Y|set E2] [F:fun X Y] [Z:pset X]
                   =  (F.1, (f_dr_aux F Z)): fun Z.1 Y]
```

The domain subtracted function can also be defined:

```
[f_domain_sub [X|set E1] [Y|set E2] [F:fun X Y] [Z:pset X] =
   f_domain_res F
         (((set_diff X Z.1),(Set_diff_in_pset X Z.1)):pset X)
                                    :fun (set_diff X Z.1) Y]
```

Range restriction and subtraction

The range restricted function is somewhat more complicated to define since the domain set must be modified when the range is restricted:

```
[f_rr_aux [X|set E1] [Y|set E2] [F:fun X Y] [Z:pset Y] =
        [x:s] (member x X) /\ (member (F.1 x) Z.1) :set E1]

Goal {X|set E1} {Y|set E2} {F:fun X Y} {Z:pset Y}
                              is_fun (f_rr_aux F Z) Z.1 F.1
... Save f_rr_aux2

[f_range_res [X|set E1][Y|set E2][F:fun X Y] [Z:pset Y] =
           (F.1,(f_rr_aux2 F Z)) : fun (f_rr_aux F Z) Z.1 ]
```

Equality of functions

Extensional equality is used as the definition of equality for these functions:

```
[Qfun = [X|set E1] [Y|set E2] [f1,f2:fun X Y]
        {x:s} (member x X) -> (E2.1 (f1.1 x) (f2.1 x))]
```

Updating functions

It is not obvious how to model the updating of a function. Here the relational operators of the previous section are not available to generate new functions from old. One way of doing it is to axiomatise a general operator `f_update`.

For use in this definition a special kind of set is needed. The type `dec_pset` represents all decidable sets which belong to the power set of a given set. A set of this type will be supplied as an argument to the update operator—it will represent the portion of the domain of the updated function over which the updating will have effect.

```
[dec_pset [S:set t] = <T:pset S> (dec T.1)]
```

The update operator must also be supplied with information about what the new results of the updated function should be. This is given in the form of an `updator`, as defined below. The updator must respect the equality that obtains over the co-domain of the function; this property is formalised below as `is_good_updater`.

```
[is_good_updater [f:s->s] =
               {x,y:s} (E1.1 x y)->(E1.1 (f x) (f y))]

[updater  = <f:s->s> (is_good_updater E1 f)]
```

The update operator is then declared — it takes a `fun` over two sets `X` and `Y`, a decidable subset of X for whose elements the update should take effect, and an updater over the type of `Y`. It returns a `fun` over `X` and `Y`:

```
[f_update : {X|set E1} {Y|set E2}
        (fun X Y) -> (dec_pset X) -> (updater t E2) -> fun X Y]
```

Note that while `updater` as defined above takes no parameters, here it is used as if it does. This is because the assumed types `s`, `t`, and `u` and their respective equalities are discharged from the environment at various times so that all the objects that have been defined have in fact been parameterised over these types and equalities. For the sake of conciseness, the details about when these discharges take place have been omitted.

Some axioms about the behaviour of `f_update` are now added. The first describes how the new function behaves for arguments that are within the updated set:

```
[update_ax1 : {X|set E1} {Y|set E2} {f:fun X Y}
 {S:dec_pset X} {g:updater t E2} {z:set_el X}
 (member z.1 S.1.1 ) ->
            (E2.1 (g.1 (f.1 z.1)) ((f_update f S g).1 z.1))]
```

The second axiom states that the function is unchanged for arguments not within the updated set:

```
[update_ax2 : {X|set E1} {Y|set E2} {f:fun X Y}
 {S:dec_pset X} {g:updater t E2} {z:set_el X}
(not (member z.1 S.1.1)) ->
            (E2.1 (f.1 z.1) ((f_update f S g).1 z.1))]
```

6.4.2 Partial functions

The implementation of total functions shown in the previous subsection was eventually used in the schema examples that were translated. It is possible to use a similar trick to represent partial functions in Lego.

The method is to use sigma types to package a Lego function with three sets, two of these being the sets over which the function is defined, while the third would represent the actual domain of the function.

Total functions could then be built up upon these partial functions. A partial function f from set X to set Y with domain Z can be said to be total if X equals Z. Total functions could therefore be modelled as those partial functions which have this property.

6.5 Discussion

Two methods of representing Z functions in Lego were presented in this chapter. The first of these, while remaining very close to the original Z definitions, proved unsuitable for use in translating schemas into Lego because of difficulties with function application. The second method was therefore used instead. While this made it possible to translate some schemas it also means that marked differences were introduced between the definitions of functions in Z and in Lego. It may well be that this results in significant logical inconsistencies between Z itself and the Lego implementation. The full implications of this, however, have not yet been considered.

6.6 Summary

An attempt was made to represent Z functions from set to set directly in Lego by using the Z definitions. This was quite straightforward, except that problems arose in defining function application.

It was then decided to represent functions in a manner more conducive to function application. For the time being, only total functions have been represented though the basic idea can be extended for partial functions.

Several of the relational operators of the previous chapter have been redefined to operate on these functions.

Chapter 7

Numbers

7.1 Introduction

Whereas Z assumes the existence of the sets $\mathbf{N}$ and $\mathbf{Z}$ of natural numbers and integers respectively, some representation of these sets must be supplied to Lego. This chapter presents an implementation of natural numbers and an attempt at representing integers in Lego.

7.2 Numbers in Z

Z uses natural numbers and integers as basic data types. The usual arithmetic operations (addition, subtraction, and multiplication) with all of their usual properties (commutativity and associativity of addition and multiplication; distributivity of multiplication over addition) are assumed.

7.3 Natural Numbers in Lego

It is necessary to find some way of modelling the natural numbers in Lego. Much work has already been done on this problem [8]. In general there are two ways in which they can be represented — by axioms, or by using Church natural numbers.

7.3.1 Axiomatizing the natural numbers

It is possible to model the natural numbers in Lego by simply giving a list of axioms. Thus a type `nat`, an inhabitant `zero` of this type, and a function `succ` from `nat` to `nat` are declared:

```
[nat:Type] [zero:nat] [succ:nat->nat]
```

The next step is to add the Peano axioms for natural numbers, and an induction principle.

The problem with this method is that even fairly trivial proofs require difficult and laborious manipulation of the axioms. The Church method of representing natural numbers which is presented in the next subsection is generally preferred.

7.3.2 Church natural numbers

The Church trick consists of representing an object by its eliminator operation [8]. In the case of the natural numbers, this means that a number is represented as an iterator. The number 3, for example, is represented as an operator which, given some type `t`, a function `s` of type `t->t`, and a starting value `z` of type `t`, returns the result obtained by applying `f` to `z` 3 times, ie. it returns `f(f(f 3))`. Typing constraints in Lego dictate that the type `t` being operated upon must be of the special class `Prop` and this sometimes restricts the usefulness of the Church natural numbers. However they possess one great advantage over the axiomatized version, which is that they normalise. This means that the Church representation of the number 3 is equivalent in Lego to the number 1+2. Hence it is easy to prove that they are equal, even under the strict Leibnitz equality. This is not so in the axiomatised implementation — the two numbers would have to be proven to be equal (under some special definition of equality) by manipulation of the axioms.

Definition environment

The definitions in this section require the booleans which were described in Chapter 3.

Basic definitions

The type `nat` is defined. An inhabitant `ze` of this type is defined—it is an iterator which performs no iterations of the function which it is given, but simply returns the starting value. It is intended to represent the natural number 0. A function `su` of type `nat->nat` is also defined. This function returns a `nat` which performs one more iteration than the given `nat`—intuitively it represents the successor function on natural numbers.

```
[nat = {t|Prop} (t->t) -> (t->t)]
[ze = [t|Prop] [s: t->t] [z: t] z : nat]
[su [n: nat] = [t|Prop] [s: t->t] [z: t] s ((n s) z): nat]
```

Some natural numbers are defined:

```
[one = su ze] [two = su one ] [three = su two ]
```

A predecessor operation

The predecessor of a natural number is defined. The definition should be thought of as a recursive function which computes the predecessor. It is also an example of the use of `nat`s as iterators.

```
[step = [p:and nat bool]
          (if (snd p)
              (pair (su (fst p)) true)
              (pair ze true))]

[pre_aux [x:nat] = x step (pair ze false)]

[pre [x:nat] = fst (pre_aux x)]
```

Arithmetic with nats

Some arithmetic operations are defined on these `nats`. `nat_sub` is zero-truncated subtraction — it gives a value of 0 if the number being subtracted is greater than the number it is subtracted from.

```
[nat_plus [m,n:nat] = m su n ]

[nat_sub [n,m:nat] = m pre n]

[nat_times [n,m:nat] = n (nat_plus m) ze]

[nat_exp [n,m:nat] = n (nat_times m) one]
```

Inductive natural numbers

Many mathematical proofs involving natural numbers make use of the Principle of Mathematical Induction. Before a similar principle can be expressed in Lego, it is necessary to restrict the `nats` which are being discussed to the inductive `nats` generated by `ze` and the function `su`. This restriction is done using the same fixed-point method used to define the finite sets in Chapter 3. First the property of being an inductive `nats` is defined. Sigma types are then used to construct the type `Nat` of `nats` with this property, so that it becomes possible to quantify over them.

```
[Inductive[S:nat->Prop] =
   ((S ze)/\ {n:nat}((S n) -> (S (su n))))]

[is_Nat = [x:nat] {P:nat->Prop} (Inductive P) ->P x]

[Nat = <n:nat> is_Nat n]
```

`ze` is shown to be inductive, and a zero `Ze` of type `Nat` is then defined:

```
Goal is_Nat ze
```

```
... Save ze_is_Nat

[Ze = (ze, ze_is_Nat):Nat]
```

The function su is shown to preserve Natness, and hence a successor Su over Nat is defined:

```
Goal {n:nat} (is_Nat n) -> is_Nat (su n)
... Save su_is_Nat

[Su [n:Nat] = ((su n.1), su_is_Nat n.1):Nat]
```

Some Nats can be defined:

```
[One = ((su Ze.1), su_is_Nat Ze.1):Nat]

[Two = ((su One.1), su_is_Nat One.1):Nat]

[Three = ((su Two.1), su_is_Nat Two.1):Nat]
```

Induction over natural numbers

It is now possible to prove an induction principle over those nats which have the property is_Nat. This induction principle is used in most of the proofs presented in the subsections which follow.

```
Goal {P:nat->Prop}
 (Q ze) -> ({n:nat} ((is_Nat n) -> ((P n) -> P (su n)))
       ->  {n:nat} ((is_Nat n) -> P n))
```

Arithmetic with Nats

The arithmetic operators defined for nats are modified to operate on Nats. The modification consists of pairing the original operator with a proof that the operator preserves is_Natness. First addition is defined:

```
Goal {x,y:nat} (is_Nat x) -> (is_Nat y)
                          -> (is_Nat (nat_plus x y))
... Save plus_is_Nat

[Nat_plus [m,n:Nat] =
     ((nat_plus m.1 n.1), (plus_is_Nat m n)) :Nat]
```

The closure proof for multiplication follows from that for addition.

```
Goal {x,y:nat} (is_Nat x) -> (is_Nat y)
                          -> (is_Nat (nat_times x y))
... Save times_is_Nat

[Nat_times [m,n:Nat] =
   ((nat_times m.1 n.1), (times_is_Nat m n)) :Nat]
```

Before subtraction can be dealt with it is necessary to show that the predecessor operation `pre` preserves the property of `is_Nat`ness. This proof is fairly complicated and involves the use of the following lemma:

```
Goal {x:nat} (is_Nat x) -> Q (pre (su x)) x
```

Note that interchanging `pre` and `su` in the above goal results in a statement which is false — the counter-example is the case where `x` is `ze`.

With this lemma proven it is now possible to define a predecessor operator on `Nats`:

```
Goal {x:nat} (is_Nat x) -> (is_Nat (pre x))
... Save pre_is_Nat

[Pre [n:Nat] = ((pre n.1),(pre_is_Nat n.1)):Nat]
```

Subtraction of `Nats` can now be defined:

```
Goal {x,y:nat} (is_Nat x) -> (is_Nat y)
                  -> (is_Nat (nat_sub x y))
... Save sub_is_Nat

[Nat_sub [m,n:Nat] =
   ((nat_sub m.1 n.1),(sub_is_Nat m.1 n.1)):Nat]
```

Equality of Nats

Two Nats are defined to be equal if their nat components are equal under Leibnitz equality. It is easy to prove that this equality QNat is reflexive, symmetric, and transitive:

```
[QNat [m,n:Nat] = Q m.1 n.1]

Goal {n:Nat} QNat n n

Goal {m,n:Nat} (QNat m n) -> QNat n m

Goal {m,n,o:Nat} (QNat m n) -> (QNat n o) -> QNat m o
```

Some arithmetic properties

It is possible to prove, with varying levels of difficulty, a number of results about the arithmetic operations which have been defined. A disadvantage of the packaging of Nats as sigma types becomes apparent at this stage: in order to prove goals that are quantified over Nats it is necessary to prove as a lemma the corresponding goal for nats with the property is_Nat and then to refine by this lemma.

It is proven that Nat_plus is commutative and associative and that Nat_times distributes over Nat_plus. The last of these proofs is fairly involved:

```
Goal {m,n:Nat} QNat (Nat_plus x y) (Nat_plus y x)

Goal {m,n,o:Nat} QNat (Nat_plus m (Nat_plus n o))
                      (Nat_plus (Nat_plus m n) o)

Goal {m,n,o:Nat}
   QNat (Nat_times x (Nat_plus y z))
        (Nat_plus (Nat_times x y) (Nat_times x z))
```

Discussion

Two general comments should be made at this stage.

- There still remain a number of arithmetic operations and results which can be defined or proven using this implementation of natural numbers. Time did not permit a full exploration of the possibilities.

- The work that has been exhibited differs from the customary presentation of Church natural numbers in that the inductive natural numbers have been packaged into a sigma type (`Nat`). This was done with Z in mind — the packaging was intended to allow quantification over the type `Nat` just as in Z schemas it is possible to quantify over $\mathbf{N}$. The advantage of visual clarity gained by this technique is somewhat offset by an increase in the amount of work that needs to be done in proving things about `Nats` — the proofs in the previous subsection are testimony to this.

7.4 Implementing Integers in Lego

An attempt was made to build a representation of the set of integers in Lego, using as a base the implementation of the natural numbers. It is not known whether this has ever before been attempted.

The integers were represented as pairs of the form (sign, magnitude). The magnitude was represented by the corresponding natural number. Various different representations of the sign were considered, and in the end it was decided that the most useful method would be to use the `bool`s of Chapter 2. Thus the boolean `true` was used as the positive sign, and `false` as the negative sign. The details of this implementation are now presented:

Basic definitions

The type `int` is defined as described above and some integers are defined. Note that there are two different integer representations of zero—positive zero, and negative zero.

```
[int = bool # nat]

[int_ze = (true, ze):int]
[m_int_ze = (false, ze):int]

[int_one = (true, one):int]
[m_int_one = (false, one):int]
```

Operators which return the sign and the magnitude of an `int` are defined:

```
[sign [i:int] = i.1]

[mag [i:int] = i.2]
```

Checking for zero

This useful operator returns the boolean `true` if its `nat` argument is not zero. Otherwise it returns `false`

```
[not_ze [n:nat] = n ([b:bool] true) false]
```

Getting rid of negative zero

This operator is intended to allow the subsequent defintion of functions on integers which never return the value `m_int_ze`. For all integers except `m_int_ze` it simply returns its argument. If given `m_int_ze` however, it returns `int_ze`, positive zero.

```
[norm [i:int] = (not_ze (mag i)) i int_ze]
```

Negating an integer

For those `int`s whose magnitude is not `ze`, this operator returns an `int` with the same magnitude but of opposite sign. For `int`s with zero magnitude it returns positive zero:

```
[negate_int [i:int] =
  norm (((toggle (sign i)), (mag i)):int)]
```

Successors

A successor operation on integers is defined. It makes use of the `norm` operator defined previously to ensure that negative zero is never generated.

The successor is computed by effectively doing a case analysis of the sign and the magnitude of the integer `i` being operated on.

If `i` is positive then `int_su i` is simply constructed by pairing the positive sign with the natural number successor (`su`) of the magnitude of `i`.

If i is negative then it is a little more complicated. The special case where i is m_int_ze must be accounted for. In this case, int_su returns int_one; in all other cases the successor is created by pairing the negative sign with the natural number predecessor (pre) of the magnitude of i.

The use of norm ensures that the successor of m_int_one is int_ze (positive zero) rather than m_int_ze (negative zero).

The definitions are as follows:

```
[pos_su [i:int] = (true, (su i.2)):int]

[neg_su_aux [b:bool] [i:int] =
 if b ((false, pre i.2):int) (int_one:int) :int]

[neg_su [i:int] =
            neg_su_aux (not_ze (mag i)) i :int]

[int_su_aux [i:int] =
             (sign i) (pos_su i) (neg_su i):int]

[int_su [i:int] = norm (int_su_aux i)]
```

Predecessors

The definition of the predecessor of an int is analogous to that of the successor. Here the crucial point is to ensure that the predecessor of int_ze is m_int_one rather than int_ze.

```
[neg_pre [i:int] = (false, su i.2):int]

[pos_pre_aux [b:bool] [i:int] =
 if b ((true, pre i.2):int) (m_int_one:int) :int]

[pos_pre [i:int] =
             pos_pre_aux (not_ze (mag i)) i :int]

[int_pre_aux [i:int] =
```

```
              (sign i) (pos_pre i) (neg_pre i):int]

   [int_pre [i:int] = norm (int_pre_aux i)]
```

Addition

The addition of integers is defined. Again this is a simple case analysis based on the sign of one of the integers being added. The fact that the definition is built upon those of `int_su` and `int_pre` should ensure that negative zero is never produced as the result of any addition of `int`s. This however remains to be proven in Lego.

```
[plusp [i,j:int] = (mag j) int_su i]

[plusn [i,j:int] = (mag j) int_pre j]

[int_plus [i,j:int] =
  (sign j) (plusp i j) (plusn i j):int]
```

Subtraction

Subtraction is simply defined as the addition of the negated integer:

```
[int_sub [i,j:int] = int_plus i (negate_int j)]
```

Multiplication

Multiplication of integers is defined.

The magnitude of the result of the multiplication of two integers `i` and `j` is computed as a positive integer in the auxilliary operation `times_2`.

The signs of `i` and `j` are used as boolean selectors to correct the sign of the computed result in the auxilliary operation `times_1` and in the main definition.

```
[times_2 [i,j:int] =
               (mag i) (int_plus j) int_ze :int]

[times_1 [i,j:int] =
  (sign j)
    (times_2 i j)
    (negate_int (times_2 i (negate_int j))):int]

[int_times [i,j:int] =
   (sign i)
     (times_1 i j)
     (negate_int (times_1 (negate_int i) j)):int]
```

7.4.1 Inductive integers

It would be useful to have some sort of induction principle on these integers, similar to that defined for the natural numbers. An attempt was made therefore to formulate such a principle.

The inductive property

First it is necessary to restrict the integers being discussed to those which possess some sort of inductive property, just as was done for the `nats`. The most obvious restriction is the following. Here the intention is to restrict the `ints` to `int_ze` and those `ints` obtained from `int_ze` by the use of the operations `int_pre` and `int_su`.

```
[IntInductive[S:int->Prop] = (S int_ze)/\
        {n:int}(S n) -> ((S (int_pre n))/\(S (int_su n)))]

[is_Int = [x:int] {P:int->Prop} (IntInductive P) ->P x]

[Int = <i:int> is_Int i]
```

Some simple consequences

It follows immediately from the above definition that `int_ze` has the property `is_Int` and that this property is preserved by the operations `int_pre` and `int_su`. This makes it possible to define zero and successor and predecessor operations for the type `Int`.

```
Goal is_Int int_ze
... Save int_ze_is_Int

[Int_ze = (int_ze, int_ze_is_Int):Int]

Goal {i:int} (is_Int i) -> is_Int (int_su i)
... Save int_su_is_Int

[Int_su [i:Int] =
 ((int_su i.1), (int_su_is_Int i.1 i.2)):Int]

Goal {i:int} (is_Int i) -> is_Int (int_pre i)
... Save int_pre_is_Int

[Int_pre [i:Int] =
 ((int_pre i.1), (int_pre_is_Int i.1 i.2)):Int]
```

An induction principle for Ints

It is even possible to prove the following, which is a kind of induction principle over these `Int`s.

```
Goal
 {P:int->Prop}(P int_ze)  ->
             ({n:int}((is_Int n) -> (P n) ->
                         (and (P (int_pre n))
                              (P(int_su n))))) ->
   {n:int}((is_Int n) -> P n)
```

Discussion

At this point it would seem that the implementation of integers in Lego is almost complete.

Problems arise however when one attempts to prove in Lego that the set of `int`s with property `is_Int` is closed under the arithmetic operations `int_plus`, `int_sub`, and `int_times` which have been defined. No proof of any of these results has yet been found.

It is not clear whether the reason for this failure is simply a lack of skill in using Lego and of time for tackling the problem, or whether some axioms need to be added to the system in order to make the proof possible, or whether the proof goals themselves are in fact false.

7.5 Summary

Natural numbers may be represented in Lego either by a collection of axioms or by using Church natural numbers. The latter is much more convenient for use in proofs and has therefore been used in this implementation.

An attempt has been made to represent integers in Lego by an extension of the method used to represent natural numbers. This has met with some success, but proofs of some desirable results still remain to be found.

Chapter 8

Schemas

8.1 Introduction

The most useful feature of the Z notation is the ability to decompose specifications into small chunks by using *schemas*. The schema notation allows the separate description of different facets of a system. These facets may be logically distinct subsystems of a complete system, or they may be general abstract systems which may be enriched or constrained in some way in order to model some particular system. The Z schema calculus contains a rich collection of operators for combining schemas in various different ways to build a specification.

Unfortunately, it has not been possible to represent more than a small fraction of this schema calculus in Lego. This chapter begins with a discussion of one method of representing schemas in Lego. The reasons why this method, though useful as far as it goes, does not lend itself to expression of the full schema calculus are discussed. Some speculations are made about other ways of representing schemas in Lego which might make it possible to model more of the schema operators.

8.2 A model for schemas in Lego

8.2.1 Schemas as sigma types

A Z schema generally consists of a collection of variables and a predicate giving some constraint on these variables. The variables are listed in some visual sequence and one often observes that the type of a variable is dependent upon the variables that preceded it. Certainly the predicate portion of a schema is dependent on the variables that were previously defined.

This observation leads naturally to the expression of schemas in Lego as dependent sums (or sigma types). Sigma types too allow the introduction of a number of variables, each of type dependent on those previously introduced. A schema can therefore be represented by a nested sigma type containing all the schema variables (in an order determined by the typing dependencies among the variables) and culminating in a proposition which expresses the schema constraint. A small example which illustrates this strategy is presented in the next section.

8.3 An example

This example is adapted from [5]. It concerns a very simple system which is called a birthday book. The birthday book is a system which keeps a record of people's birthdays. The *state space* of the system is described in Z by the following schema.

$BirthdayBook$

$known : \mathbb{P} NAME$

$birthday : NAME \rightarrow DATE$

$known = \operatorname{dom} birthday$

In order to translate this schema into Lego it is first necessary to declare the underlying types `name_type` and `date_type`, their respective equalities, and then the sets `Name` and `Date`:

```
[name_type, date_type:Type]

[name_EQ = Q_EQ name_type]
[date_EQ = Q_EQ date_type]

[Name: set name_EQ]
[Date: set date_EQ]
```

The schema makes use of the definition of functions `funs` which was presented in 6.4. This is what it looks like:

```
[BirthdayBook = <known: pset Name             >
                <birthday: fun Name Date      >
               (QSet known.1 (f_dom birthday))]
```

It is now possible to declare variables of the type `BirthdayBook` in other schemas, or to prove results which quantify over the type `BirthdayBook`. For example the schema which represents the initial state of the birthday book can be expressed in Lego. Here is what it looks like in Z:

InitBirthdayBook

$BirthdayBook$

$known = \varnothing$

The Lego version looks like this:

```
[InitBirthdayBook =
     <birthday_book : BirthdayBook>
     [known = birthday_book.1.1   ]
     (QSet known (void name_EQ  ))]
```

The explicit definition of the variable `known` in this schema illustrates one of the differences between the Z schema notation and the use of Lego sigma types — whereas in the Z notation the inclusion of one schema within the body of another results in the automatic inclusion of all variable names declared in the included schema, in Lego this is not the case. The components of the included sigma type must be explicitly named in the new schema (eg. the component `known` in the example above) if the presence of these variable names is desired.

8.4 The Shortcomings of this Approach

The expression of Z schemas in Lego as sigma types is natural and, except for the need to name explicitly the components of included schemas, is fairly non-cumbersome to work with. The major shortcoming, however, is that it is not clear how to extend this approach to model the full Z schema calculus. Sigma types can support the definition of simple schemas (by this is meant schemas which are not built up from any other schemas by any of the operations of the schema calculus); the use of schemas as types; the projection of the components of schemas; the inclusion of one schema within the definition of another; the parameterisation of schemas; the enriching of schemas by adding new variables or conjoining a new predicate. Many of the other operations on schemas, however cannot be expressed directly as any existing Lego operation on sigma types, nor can they be defined in the general case. These include schema decoration; negation; conjoining or disjoining two schemas; the operations $S \Rightarrow T$ and $S \Leftrightarrow T$ where S and T are two schemas; hiding of variables; extraction of pre and post conditions; overriding one schema by another; schema composition; piping of schemas. For full descriptions of all these schema operations see appendix B.

It would seem then that this approach is a dead end. How then should schemas be expressed in Lego? One approach might be to explicitly define a type `Schema` consisting of a dependent pair, the first component of which might be some list of variables (using some representation of lists in Lego), while the second would be a predicate of type dependent on the first component. It might then be possible to define some operations on this type `Schema`s which model the operations of the Z schema calculus.

However the representation of schemas as sigma types is sufficiently expressive to allow the translation of non-trivial Z schemas into Lego, and was therefore used in the translation example presented in the following chapter.

8.5 Summary

Schemas are one of the most useful concepts in the Z notation. It is quite straightforward to model schemas in Lego as sigma types so this is the representation which has been chosen as the working model in this implementation.

A small example of this use of sigma types for modelling schemas has been given.

Sigma types do not capture the full richness of the Z schema calculus. Many Z operators on schemas cannot therefore be expressed in this implementation as it stands. However this model is expressive enough to allow the large translation example presented in the next chapter.

Chapter 9

A Specification Example

9.1 Introduction

This chapter presents an application of the portions of the Z notation which have been expressed in Lego. A specification case study in Z has been translated into Lego, and some theorems about this specification have been formulated and proved. The case study chosen for translation is the one referred to as CAVIAR in [3].

9.2 The analysis of the CAVIAR system

CAVIAR is an acronym for a Computer Aided Visitor Information And Retrieval System. This is a system concerned with the "recording and retrieval of data about arrangements for visitors and meetings at a large industrial site." The system in its entirety is briefly described in the following account taken from [3].

"We now present a brief account of the existing system, emphasising the important concepts in boldface. **Visitors** come to the site to attend **meetings** and/or consult Company employees. A visitor may require a **hotel reservation** and/or **transport reservation**. Each meeeting is also required to take place in a designated **conference room**, at a certain **time**. A meeting may require the use of a **dining room** for lunch on a particular date. Booking a dining room requires **lunch information** including the number of places needed. Each conference room booking requires **session information** about resources required for use in the meeting, e.g., viewgraphs, projectors. The main operations required at the user interface can briefly be described as facilities for booking, changing and cancelling the use of resources."

The first step in the analysis of the system is the identification of the basic sets of which the system is composed. These are the objects highlighted in boldface in the above account. The analyst then makes the observation that several of the identified sets can be viewed as *resources* while other sets can be viewed as *users* of these resources. This leads to the decomposition of the system into a number of resource-user subsystems. Some of these subsystems are listed below:

Resource	**User**
Conference Rooms	Meetings
Dining Rooms	Meetings
Meetings	Visitors
Hotel Reservations	Visitors
Transport Reservations	Visitors

This mathematical abstraction motivates the development in Z of a general theory of general theory of resource-user systems. A number of schemas are defined which specify a general resource-user system with operations allowing the booking of a resource, the cancellation of a booking, and the deletion of a resource or of a user. The specification of each of the subsystems of CAVIAR is created by refining the general model to suit the constraints of the particular subsystem. Finally, the complete CAVIAR specification is built up from the subsystem specifications.

9.3 A general resource-user system

The portions of Z which have been expressed in Lego are not sufficient to allow the translation of the complete CAVIAR specification, since this involves the use of many of the operations of the Z schema calculus which have not been translated. However it is possible to translate the specification of the general resource-user system on which the Caviar specification is based. This translation is presented in this section.

A number of theorems concerning the Z specification are also given (without proofs) in [3]. Some of these theorems have been translated into Lego, and two of them have been proven in Lego. This work is also presented in this section.

9.3.1 Background definitions

Z version

The general resource-user system is considered to be parameterised over three sets, T, R, and U. Informally, T is meant to represent a set of *time slots*, R is a set of *resources* and U is a set of *users*.

Lego version

In Lego it is necessary to declare the types `Time`, `Resource`, and `User`, and then the sets `T`, `R`, and `U`. Straight syntactic equality is used in defining these sets.

In addition it has been found necessary to postulate that `T` is not empty in order to prove one of the theorems that follow. This assumption appears to be implicit in the original Z specification.

```
[Time, Resource, User |Type]
[T:set (Q_EQ Time)] [R:set (Q_EQ Resource)] [U:set (Q_EQ User)]
[sometime:Time]  [T_not_empty: member sometime T]
```

9.3.2 The general resource-user system

Z version

The following is the Z schema which describes the state of the system. The state is captured by a total function ru which maps the time slots t in T to *relations* between R and U. The schema component *in-use* gives the set of resources in use at any instant, *users* gives the set of users occupying resources at any instant, and ur gives the inverse relation to ru at any given time. These last three components are all derivable from the first, so it is necessary to state as a constraint that the components are consistent with each other.

R-U

$ru : T \rightarrow (R \leftrightarrow U)$

$in\text{-}use : T \rightarrow \mathbb{P} R$

$users : T \rightarrow \mathbb{P} U$

$ur : T \rightarrow (U \leftrightarrow R)$

$\forall\, t : T \bullet$

$\quad in\text{-}use(t) = \mathrm{dom}\; ru(t) \wedge$

$\quad users(t) = \mathrm{ran}\; ru(t) \wedge$

$\quad ur(t) = (ru(t))^{-1}$

Lego version

This is what the above schema looks like when expressed in Lego:

```
[R_U =
   <ru : fun T (universe (QRel_EQ R U))>
   <in_use : fun T (powerset R)        >
   <users : fun T (powerset U)         >
   <ur : fun T (universe (QRel_EQ U R))>
   ({t: set_el T}
   ((QSet (in_use.1 t.1) (dom (ru.1 t.1))) /\
    (QSet (users.1 t.1 ) (rng (ru.1 t.1))) /\
    (QRel (ur.1 t.1 )    (inv (ru.1 t.1)))))]
```

9.3.3 The initial state

Z version

The initial state of the system is defined by making $ru(t)$ the empty relation for each t:

$$\mathit{Init\text{-}R\text{-}U} \mathrel{\widehat=} [\mathit{R\text{-}U} \mid \operatorname{ran} ru = \{\varnothing\}]$$

Lego version

The same schema looks like this when stated in Lego:

```
[Init_R_U =
   <R_U1 : R_U                                    >
   (QSet (f_rng R_U1.1                            )
         (singleton (QRel_EQ R U) (void_rel R U)))]
```

9.3.4 Changing the system state

The operations which can be carried out on this system are concerned with making a new booking, cancelling an existing booking, deleting a resource and deleting a user. These all share the common property of modifying the relation that exists at some particular time. More generally, one can consider the class of operations on *R-U* which cause the image of a *set* of time values to be altered. This class can be described by a schema which involves the following components: a state before, *R-U*, which introduces *ru*, *in-use*, *users*, and *ur*; a state after, *R-U′*, which introduces *ru'*, *in-use'*, *users'* and *ur'*; a set of time values, $t?$ which denotes an input. The schema constraint captures the fact that the operations always leave the function ru unchanged except for times in $t?$.

Z version

This is what this schema looks like in Z notation:

$$
\begin{array}{|l}
\multicolumn{1}{l}{\Delta R\text{-}U} \\
\hline
R\text{-}U \\
R\text{-}U' \\
t? : \mathbb{P}\, T \\
\hline
t? \lhd ru' = t? \lhd ru \\
\hline
\end{array}
$$

Lego version

Expressed in Lego, the above schema looks like this:

```
[deltaR_U =
    <oldR_U, newR_U : R_U              >
    <t_in : dec_pset T                 >
    (Qfun (f_domain_sub oldR_U.1 t_in.1)
          (f_domain_sub newR_U.1 t_in.1))]
```

9.3.5 A booking operation

The general state change operation is now used to build a schema describing a successful booking operation. The new schema inherits all the properties of $\Delta R\text{-}U$. In addition it includes two input parameters $r? : R$ and $u : U$, and is constrained by a predicate which imposes a requirement on the input parameters and also further relates the before and after states.

Z version

This following Z schema represents a successful booking operation.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{R-U-Book}} \\
\hline
\Delta R\text{-}U \\
r? : R \\
u? : U \\
\hline
\forall t : t? \bullet \\
\qquad (r?, u?) \notin ru(t) \land \\
\qquad ru'(t) = ru(t) \cup \{r? \mapsto u?\} \\
\hline
\end{array}
$$

Lego version

This is what the above schema looks like when expressed in Lego:

```
[R_U_Book =
   <dR_U : deltaR_U          >
   <r_in : set_el R          >
   <u_in : set_el U          >
   ([timeset = dR_U.2.2.1.1.1]
    [oldR_U = dR_U.1         ]
    [newR_U = dR_U.2.1       ]
    [old_ru = oldR_U.1.1     ]
    [new_ru = newR_U.1.1     ]
    {time : set_el timeset}
      (and
        (not (is_related (old_ru time.1) r_in.1 u_in.1))
        (QRel (new_ru time.1)
              (rel_union (old_ru time.1)
                         (singleton_rel r_in u_in)))))]
```

9.3.6 Cancelling an existing booking

Cancellation of a booking is said to be successful if the booking to be cancelled is actually an element of the original relation. A schema which describes a successful cancellation operation is here defined.

Z version

$$
\begin{array}{|l}
\multicolumn{1}{l}{\underline{\quad R\text{-}U\text{-}Cancel \quad}} \\
\Delta R\text{-}U \\
r? : R \\
r? : U \\
\hline
\forall\, t : t? \bullet \\
\qquad (r?, u?) \in ru(t) \wedge \\
\qquad ru'(t) = ru(t) - \{r? \mapsto u?\} \\
\hline
\end{array}
$$

Lego version

This is the translation of the above schema into Lego.

```
[R_U_Cancel =
    <dR_U : deltaR_U          >
    <r_in : set_el R          >
    <u_in : set_el U          >
    ([timeset = dR_U.2.2.1.1.1]
     [oldR_U = dR_U.1         ]
     [newR_U = dR_U.2.1       ]
     [old_ru = oldR_U.1.1     ]
     [new_ru = newR_U.1.1     ]
     {time : set_el timeset}
       (and
         (is_related (old_ru time.1) r_in.1 u_in.1)
         (QRel (new_ru time.1)
               (rel_diff (old_ru time.1)
                         (singleton_rel r_in u_in)))))]
```

9.3.7 Removing a resource from the system

This schema describes an operation which removes from the system an existing resource and all uses of that resource.

Z version

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{R-U-Del-Res}} \\
\hline
\Delta R\text{-}U \\
r? : R \\
\hline
\forall\, t : t? \bullet \\
\qquad r? \in \operatorname{dom} ru(t) \wedge \\
\qquad ru'(t) = \{r?\} \lhd ru(t) \\
\hline
\end{array}
$$

Lego version

This is the translation of the above schema into Lego:

```
[R_U_Del_User =
    <dR_U : deltaR_U                           >
    <u_in : set_el U                           >
    ([timeset = dR_U.2.2.1.1.1                 ]
     [oldR_U = dR_U.1                          ]
     [newR_U = dR_U.2.1                        ]
     [old_ru = oldR_U.1.1                      ]
     [new_ru = newR_U.1.1                      ]
     [u_in_set = (singleton (Q_EQ User) u_in.1)]
     [H = Subset_in_pset (singleton_rule u_in) ]
     [u_in_pset = (u_in_set,H):(pset U)        ]
     {time : set_el timeset}
       (and
         (member  u_in.1 (rng (old_ru time.1)))
         (QRel (new_ru time.1)
               (range_sub (old_ru time.1)
                           u_in_pset       ))))]
```

The somewhat cryptic manipulations of the input parameter `u_in` are necessary in this implementation in order to construct an object from `u_in` that is of the right type to be given as an argument to the `range_sub` operator.

9.3.8 Removing a user from the system

This schema describes an operation which removes from the system an existing user together with all bookings made by that user.

Z version

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{R-U-Del-User}} \\
\hline
\Delta R\text{-}U \\
u? : R \\
\hline
\forall\, t : t? \bullet \\
\qquad u? \in \operatorname{ran} ru(t) \land \\
\qquad ru'(t) = ru(t) \rhd \{u?\} \\
\hline
\end{array}
$$

Lego version

This is the expression in Lego of the above schema:

```
[R_U_Del_Res =
    <dR_U : deltaR_U >
    <r_in : set_el R >
    ([timeset = dR_U.2.2.1.1.1                     ]
     [oldR_U = dR_U.1                              ]
     [newR_U = dR_U.2.1                            ]
     [old_ru = oldR_U.1.1                          ]
     [new_ru = newR_U.1.1                          ]
     [r_in_set = (singleton (Q_EQ Resource) r_in.1)]
     [H = Subset_in_pset (singleton_rule r_in)     ]
     [r_in_pset = (r_in_set,H):(pset R)            ]
     {time : set_el timeset}
       (and
         (member r_in.1 (dom (old_ru time.1)))
         (QRel (new_ru time.1)
               (domain_sub (old_ru time.1)
                           r_in_pset          ))))]
```

9.4 Two theorems

In the course of the development of the Z specification of Caviar in [ref] a number of theorems about the properties of the schemas were given. Their proofs however were omitted. Two of the theorems regarding the general resource-user system are discussed in this section. They are stated in both the original Z notation and in the Lego syntax. They have been proven using Lego and their proofs are here outlined.

9.4.1 Theorem 1.

This theorem proves that the initial state described in 9.3.3 is reasonable and assures the consistency of the definition of $R\text{-}U$.

Z version

This is the statement of this theorem in Z notation:

Theorem 1. $\vdash \exists\, R\text{-}U \bullet \mathit{Init}\text{-}R\text{-}U$

Lego version

In Lego, the schema `Init_R_U` is considered to be a type. The Lego version of this theorem is therefore the assertion that this type is inhabited, i.e., that an object of this type exists. This is what it looks like:

```
Goal Init_R_U
```

This theorem has been proven in Lego. The proof, which is far from being as concise as the statement of the theorem, consists of the building up of an object in Lego. which is of the type `Init-R-U`. The first step is the definition of `init_ru`. This is defined as a function which, for all time values `t`, returns the empty relation.

```
[init_ru_aux = [t:Time] void_rel R U : Time -> (rel R U)]

Goal is_fun T (universe (QRel_EQ R U)) init_ru_aux
...init_ru_aux2

[init_ru =
(init_ru_aux, init_ru_aux2): fun T (universe (QRel_EQ R U))]
```

`init_in_use` is defined as the function which, for all time values `t`, gives the empty set of resources:

```
[init_in_use_aux = [t:Time] (void (Q_EQ Resource))]

Goal is_fun T (powerset R) init_in_use_aux
...init_in_use_aux2

[init_in_use =
(init_in_use_aux, init_in_use_aux2): fun T (powerset R)]
```

`init_users` is defined similarly to `init_in_use`:

```
[init_users_aux = [t:Time] (void (Q_EQ User))]

Goal is_fun T (powerset U) init_users_aux
...init_users_aux

[init_users =
(init_users_aux, init_users_aux2): fun T (powerset U)]
```

`init_ur` is defined similarly to `init_ru`:

```
[init_ur_aux = [t:Time] void_rel U R : (Time -> (rel U R))]

Goal is_fun T (universe (QRel_EQ U R)) init_ur_aux

[init_ur =
(init_ur_aux, init_ur_aux2): fun T (universe (QRel_EQ U R))]
```

Having defined all these components, it is now necessary to prove that they conform to the schema constraints. First it is shown that they obey the constraint enforced by R_U:

```
Goal {t: set_el T}
   (QSet (init_in_use.1 t.1) (dom (init_ru.1 t.1))) /\
   (QSet (init_users.1 t.1)   (rng (init_ru.1 t.1))) /\
   (QSet (init_ur.1 t.1).1 (inv (init_ru.1 t.1)).1)
...init_lemma1
```

It is then shown that the extra constraint imposed by Init_R_U is also obeyed:

```
Goal QSet (f_rng init_ru)
          (singleton (QRel_EQ R U) (void_rel R U))
...init_lemma2
```

Finally all these definitions and proofs are packaged into a tuple which has the type Init_R_U. This completes the proof.

```
[Init_R_U_example =
  (((init_ru, init_in_use, init_users, init_ur, init_lemma1):R_U),
    (init_lemma2)):Init_R_U]
```

9.4.2 Theorem 2.

The second theorem concerns the booking operation of 9.3.5. In this schema the predicate

$\forall\, t : t? \cdot (r?, u?) \notin ru(t)$

is made a precondition for a successful booking. Theorem 2 states that this condition is sufficient, that is, that if a valid system state is present, and if the required input parameters of the correct type are available, and if furthermore the above condition holds, then there exists a resulting valid system state which is related to the above according to the *R-U-Book* schema.

Z version

This is the statement of theorem 2 in the original Z notation:

Theorem 2.

$$R\text{-}U \wedge [t? : \mathbb{P}\,T;\ r? : R;\ u? : U \mid \forall\, t : t? \bullet (r?, u?) \notin ru(t)]$$

$$\vdash$$

$$\exists\, R\text{-}U' \bullet R\text{-}U\text{-}Book$$

Lego version

Theorem 2 is expressed in Lego in the following way:

```
Goal {R_U_given : R_U}
     {other_conditions :
            <t_in : dec_pset T       >
            <r_in : set_el R         >
            <u_in : set_el U         >
            ({time :set_el t_in.1.1 }
               (not (member
                       (r_in.1, u_in.1)
                       (R_U_given.1.1 time.1).1)))}
      R_U_Book
```

This theorem has been proven in Lego. The proof is very long and, like that of theorem 1, is constructed in a number of stages. First all of the conditions given in the proof are assumed. Then a schema of type `R_U_Book` is constructed piece by piece using the set-theoretic, relational and functional operators that have been defined to build the components of the new schema. It is then shown that these components satisfy the constraints imposed by `R_U`, `deltaR_U` and `R_U_Book`. All these definitions and proofs are then combined into a tuple which has the type `R_U_Book`. Finally the assumptions are discharged from the environment to complete the proof of the theorem as stated.

9.5 Discussion

Two main observations were made during the course of this translation exercise.

- Quite obviously, the representation of schemas in Lego does not possess the visual elegance of the original Z notation. One of the reasons for this is that extraction of the components of a sigma type in Lego is a cumbersome operations. This would be improved if Lego included some method of indexing the components of tuples and sigma types, and then referring to these components by index number.

  Another, more subtle, reason arises because there is in Z a syntactic ambiguity in the use of names of sets and schemas. *Set* names are also as the name of the *type* of elements of the named set. Schema names denote either a *schema* (that is, some grouping of variables with a predicate relating them), or the *type* of *system states* which satisfy the named schema. No actual ambiguity arises from this usage because the human user of Z can infer from the context the correct semantic interpretation. Lego, however, is not so clever. In Lego, these different usages of set and schema names must be defined separately. This leads to some visual cluttering of the Lego code.

- Proving things in Lego can be a tedious task. Small proof steps which are taken for granted by human mathematicians must be painstakingly stepped through in Lego. The proofs outlined in this chapter are cases in point.

  Occasionally, however, this can have its rewards. Sometimes Lego's proof-checking can bring to light certain assumptions which may escape notice otherwise. An example of this is thought to have occured during the proof of 'Theorem 1' in this chapter. Proving this theorem in Lego required the (minor) assumption that the set T of time-values was non-empty. It is thought that this assumption would also be required in a traditional proof of this theorem; yet it was never stated in the original specification.

9.6 Summary

The CAVIAR specification in Z describes a system for recording and retrieving data about arrangements for visitors and meetings at a large industrial site. The specification itself is built up from a specification of an abstract general resource-user system.

The schemas describing the general resource-user system have been translated into Lego. Two theorems about this specification have been proven. An assumption implicit in the original specification was brought to light during one of these proofs.

It is observed that the Lego syntax is quite inelegant compared to the original Z notation, and that Lego proofs of seemingly obvious results are long and laborious. However occasionally stepping through the detailed steps of a proof in Lego can be rewarding, as it can expose assumptions which might otherwise be missed.

Chapter 10

Conclusion

This project has been a very interesting and challenging exercise. Each chapter of this report describes a different challenge in representing some new mathematical concept in Lego. Lego itself is an exciting toy/tool, combining just the right proportions of difficulty and reward to make it immensely addictive. Its expressive powers continued to surprise the author throughout the duration of the project. Very often it was discovered that a difficulty in proving something in Lego was due, not to a lack of skill or familiarity with Lego, nor to limitations within Lego itself, but to the fact that the proof goal itself was untrue. The author has been left with a feeling of respect and admiration for the Lego system and its creators.

Comments about the present work

Some general comments can be made about the work that has been done:

The original project outline called for the translation of a Z specification case study into Lego, the formulation and proofs of some theorems about this specification, the translation of a number of useful features of the Z notation into Lego, and the formulation of some general rules for translating Z into Lego. This outline was necessarily quite open-ended, since it was impossible to predict at the outset how difficult a task had been set and how much could be accomplished in the allotted time. All of these goals have now been realised to some extent and it is thought that the original outline has been satisfied.

Despite this, it cannot be said that a full implementation of Z in Lego has been achieved. This, even if it turned out to be possible, would have been too large a task for the given time. What has been achieved is the modelling of a subset of the Z notation, large enough to permit the translation of some Z schemas and some proofs about these schemas and to indicate some of the possibilities and the problems of this endeavour. Enough has been done to suggest that the goal of implementing Z in Lego might yet be achieved, if certain identifiable problems can be overcome somehow. More will be said about this in the next subsection.

Future prospects for this work

Early in the course of this project, the author experienced many doubts about whether anything worthwhile could possibly be accomplished. There were so many difficulties involved in expressing in Lego the mathematical concepts (viz. numbers, sets, relations, and functions) that were the basic fabric of the Z notation that it seemed as if the work would never get to the stage where a Z schema could actually be expressed in Lego. It therefore came almost as a (very pleasant) surprise when this stage was reached and not only was it possible to translate some schemas, but some theorems about these schemas could actually be proved. The author now believes that this work is capable of being extended and even more features of the Z notation can be expressed within Lego. It is also thought to be possible to refine or to extend the existing implementations of various features. Some suggestions for this are listed here.

- The definition of 'set' which has been implemented is syntactically quite neat and is, in the author's opinion, powerful enough to continue to be used even in an implementation of the full Z notation. It has been observed that certain sequences of proof steps, particularly certain sequences of refinements by the various properties (reflexivity, symmetry, and transitivity) of the given equality over the elements of a set, frequently recur in proofs. Further analysis of this would quite probably make it possible to prove some key lemmas incorporating these sequences of proof steps, thereby shortening many proofs that involve sets.

- A novel method of representing integers in Lego was experimented with during this project. A full implementation of Z in Lego would need some representation of the integers and this was the best that could be found. These "integers" appeared to behave as they were meant to; yet proofs that they possessed certain desirable properties were not found. It would be interesting to devote some more time to these integers and to settle the questions which were raised.

- Two implementations of "functions" were presented in this report; the first corresponded directly to Z's definitions but presented some difficulties when defining function application, while the second differed considerably from the definitions used by Z. The second method was used since in the schema translation examples because function application was required and it was easy to do with this representation. However more thought needs to be given to the problem of function representation. It remains to be determined whether or not using a definition different from that used by Z results in an implementation which is logically inconsistent with Z. It might be possible to find some way of defining function application for the first kind of function, by redefining the existential quantifier (the crux of the problem) in Lego.

  The second method has only been used to represent total functions; partial functions can be represented by an adaptation of this method and this also remains to be done.

- For the purposes of this project Z schemas were represented in Lego as Σ-types; this method was easy and straightforward and certain operations of the schema calculus could be translated directly as operations over Σ-types. However this representation cannot easily be extended to include the remaining operations of the schema calculus. It is necessary to find some way of representing "schema" itself as a datatype in Lego, so that general operations over schemas can be defined.

Appendix A

Mathematical definitions used by Z

This is a selection from the glossary of Z notation in [1].

A.1 Definitions and declarations

Let x, x_k be identifiers and let T, T_k be sets.

$LHS \mathrel{\widehat{=}} RHS$	Definition of LHS as syntactically equivalent to RHS.
$x : T$	Declaration of x as type T.
$x_1 : T_1;\ x_2 : T_t;\ \ldots;\ x_n : Tn$	List of declarations.

A.2 Logic

Let P, Q be predicates and let D be a declaration.

$true, false$	Logical constants.
$\neg P$	Negation: "not P."
$P \wedge Q$	Conjunction: "P and Q."
$P \vee Q$	Disjunction: "P or Q ".
$P \Rightarrow Q$	Implication: "P implies Q."
$P \Leftrightarrow Q$	Equivalence: "P is logically equivalent to Q."
$P \rightarrow Q, R$	Conditional: "if P then Q else R."
$\forall x : T \cdot P$	Universal quantification: "for all x of type T, P holds."
$\exists x : T \cdot P$	Existential quantification: "there exists an x of type T such that P."
$\exists ! x : T \cdot P_x$	Unique existence: "there exists a unique x of type T such that P." $\widehat{=} (\exists x : T \cdot P_x \wedge \neg (\exists y : T \mid y \neq x \cdot P_y))$

A.3 Sets

Let S, T and X be sets; t,t_k terms; P a predicate and D declarations.

$t \in s$	Set membership: "t is an element of S."
$t \notin S$	$\widehat{=} \neg(t \in S)$.
$S \subseteq T$	Set inclusion: $\widehat{=}(\forall x : S \cdot x \in T)$.
$S \subset T$	Strict set inclusion: $\widehat{=} S \subseteq T \wedge S \neq T$.
$\{\}$	The empty set.
(t_1, t_2)	Ordered pair of t_1 and t_2.
$\{x : T \mid P\}$	The set containing exactly those x of type T for which P holds.
$T_1 \times T_2$	Cartesian product: the set of all pairs such that the first element is of type T_1 and the second is of type T_2.
$\{D \mid P \cdot t\}$	The set of t's such that given the declarations D, P holds.
$\{D \cdot t\}$	$\widehat{=}\{D \mid true \cdot t\}$.

$\mathbb{P} S$ Powerset: the set of all subsets of S.

$\mathbb{F} S$ Set of finite subsets of S:
$\widehat{=} \{ T : \mathbb{P} S \mid Tisfinite \}$.

$S \cap T$ Set intersection: given $S,\ T : \mathbb{P} X$,
$\widehat{=} \{ x : X \mid x \in S \wedge x \in T \}$.

$S \cup T$ Set union: given $S,\ T : \mathbb{P} X$,
$\widehat{=} \{ x : X \mid x \in S \vee x \in T \}$.

$S - T$ Set difference: given $S,\ T : \mathbb{P} X$,
$\widehat{=} \{ x : X \mid x \in S \wedge x \notin T \}$.

$\bigcap SS$ Distributed set intersection: given $SS : \mathbb{P}(\mathbb{P} X)$,
$\widehat{=} \{ x : X \mid (\forall S : SS \cdot x \in S) \}$.

$\bigcup SS$ Distributed set union: given $SS : \mathbb{P}(\mathbb{P} X)$,
$\widehat{=} \{ x : X \mid (\exists S : SS \cdot x \in S) \}$.

A.4 Numbers

$\mathbb{N}$ The set of natural numbers (non-negative integers)

$\mathbb{Z}$ The set of integers (positive, zero and negative)

A.5 Relations

A relation is defined by a set of ordered pairs. Hence operators defined for sets can be used on relations.

Let X, Y, and Z be sets; $x : X$; $y : Y$; and $R : X \leftrightarrow Y$.

$X \leftrightarrow Y$ The set of relations from X to Y:

$\widehat{=} \mathsf{P}(X \times Y)$.

xRy x is related to y:

$\widehat{=} (x, y) \in R$.

$x \mapsto y$ $\widehat{=} (x, y)$

$\mathsf{dom} R$ The domain of a relation:

$\widehat{=} \{x : X \mid (\exists\, y : Y \cdot xRy)\}$.

$\mathsf{rng} R$ The range of a relation:

$\widehat{=} \{y : Y \mid (\exists\, x : X \cdot xRy)\}$.

$R_1;\ R_2$ Forward relational composition:

given $R_1 : X \leftrightarrow Y;\ R_2 : Y \leftrightarrow Z$,

$\widehat{=} \{x : X;\ z : Z \mid (\exists\, y : Y \cdot xR_1y \wedge yR_2z)\}$.

$R_1 \circ R_2$ Relational composition:

$\widehat{=} R_2;\ R_1$

R^{-1} Inverse of relation R:

$\widehat{=} \{y : Y;\ x : X \mid xRy\}$.

$\mathsf{id}X$ Identity function on the set X

$\widehat{=}\{x : X \cdot x \mapsto x\}$.

$R(S)$ Image: given S : **P** X,

$\widehat{=}\{y : Y \mid (\exists\, x : S \cdot xRy)\}$.

$S \lhd R$ Domain restriction to S: given $S : \mathsf{P}X$,

$\widehat{=}\{x : X;\ y : Y \mid x \in S \wedge xRy\}$.

$S \lhd R$ Domain subtraction : given $S : \mathsf{P}X$,

$\widehat{=}(X - S) \lhd R$.

$R \rhd T$ Range restriction to T: given $T : \mathbf{P}\,Y$,

$\widehat{=}\{x : X;\ y : Y \mid xRy \wedge y \in T\}$.

$R \rhd T$ Range subtraction of T: given $T : \mathbf{P}\,Y$,

$\widehat{=}R \rhd (Y - T)$.

$R_1 \oplus R_2$ Overriding: given $R_1, R_2 : X \leftrightarrow Y$,

$\widehat{=}(\mathbf{dom}R_2 \lhd R_1) \cup R_2$.

A.6 Functions

A function is a relation with the property that for each element in its domain there is a unique element in its range related to it. As functions are relations all the operators defined above for relations also apply to functions.

$X \nrightarrow Y$ The set of partial funcions from X to Y:

$\widehat{=} \{ f : X \leftrightarrow Y \mid (\forall\, x : \mathbf{dom} f \cdot (\exists!\, y : Y \cdot xfy)) \}.$

$X \rightarrow Y$ The set of total functions from X to Y:

$\widehat{=} \{ f : X \nrightarrow Y \mid \mathbf{dom} f = X \}.$

ft The function f applied to t.

Appendix B

Schema notation

This is a selection from the glossary of schema notation in [1].

Schema definition A schema groups together some declarations of variables and a predicate relating these variables. There are two ways of writing schemas: vertically, for example:

$$S \mathrel{\widehat{=}} \begin{array}{|l} x : \mathsf{N} \\ S : set\mathsf{N} \\ \hline \forall\, z : s \cdot z < x \end{array}$$

or horizontally, for the same example,

$S \mathrel{\widehat{=}} [x : \mathsf{N};\ s : set\mathsf{N} \mid \forall\, z : s \cdot z < x]$.

Schemas as types When a schema name S is used as a type it stands for the set of all objects described by the schema.

Projection functions The component names of a schema may be used as projection (or selector) functions.

tupleS This is the tuple formed from a schema's variables.

predS This is the predicate part of a schema.

Inclusion A schema S may be included within the declarations of a schema T, in which case the declarations of S are merged with the other declarations of T (variables declared in both S and T must be of the same type) and the predicates of S and T are conjoined.

$S \mid P$ The schema S with predicate P conjoined to its predicate part.

$S;\ D$ The schema S with the declarations D merged with the declarations of S.

$S[new/old]$ Renaming of components: the schema S in which the component *old* has been renamed to *new* both in the declaration and at its every free occurrence in the predicate.

Decoration Decoration with subsript, superscript, prime, etc.; this results in the systematic renaming of the variables declared in the schema.

$\neg S$ The schema S with its predicate part negated.

$S \wedge T$ The schema formed from schemas S and T by merging their declarations (as in inclusion) and conjoining their predicates.

$S \vee T$ The schema formed from schemas S and T by merging their declarations and disjoining their predicates.

$S \Rightarrow T$ The schema formed from schemas S and T by merging their declarations and taking $predS \Rightarrow T$ as the predicate.

$S \Leftrightarrow T$ The schema formed from schemas S and T by merging their declarations and taking $predS \Leftrightarrow T$ as the predicate.

$S \setminus (v_1, v_2, \ldots, v_n)$ Hiding: the schema S with the variables $v_1, v_2, \ldots$, and v_n hidden: the variables listed are removed from the declarations and are existentially quantified in the predicated.

The following conventions are used for variable names in those schemas which represent operations—that is, which are written as descriptions of operations on some state:

undashed	state before the operation
dashed	state after the operation
ending in "?"	inputs to (arguments for) the operation
ending in "!"	outputs from (results of) the operation

The following schema operations only apply to schemas following the above conventions.

preS Precondition: all the state after components (dashed) and the outputs (ending in "!") are hidden.

postS Postcondition: this is similar to precondition except all the state before components (undashed) and inputs (ending in "?") are hidden.

$S \oplus T$ Overriding: $\widehat{=}(S \wedge \neg pre T) \vee T$.

$S; T$ Schema composition: if we consider an intermediate state that is both the final state of the operation S and the initial state of the operation T then the composition of S and T is the operation which relates the initial state of S to the final state of T through the intermediate state. To form thje composition of S and T we take the state-after components of S and the state-before components of T that have a basename[1] in common, rename both to new variables, take the schema which is the "and" ($\wedge$) of the resulting schemas, and hide the new variables.

$S \gg T$ Piping: this schema operation is similar to schema composition; the difference is that, rather than identifying the state after components of S with the state before components of T, the output components of S are identified with the input components of T that have the same basename.

[1]basename is the name with any decoration ("′", "!", "?", etc,) removed

Appendix C

A Glossary of Some Lego Terms

This glossary has been selected from [13].

Identifiers `x` stands for an arbitrary variable (for both declarations and definitions) which can be any identifier.

Type universes `Prop` and `Type` are called *type universes*. Intuitively one can consider types as sets. In general, any term which has a universe as its type is called a *type*. When a term has type `Prop` it is also called a *proposition*.

Π-types and λ-abstraction `{x:A}B` is the Lego notation for ordinary dependent Π-types ($\Pi x : A.B$) in type theory. `{x:A}B` is often written as `A->B` when the identifier `x` does not occur free in `B`. When `{x:A}B` is a proposition, it intuitively stands for the logical formula '$\forall\, x : A.B$'. When it is not a proposition, it intuitively denotes a set of functions f with domain `A` such that f`(a)` is of type `[a/x]B` [1]. Particularly, if both `A` and `B` are propositions, the proposition `A->B` intuitively means "`A` implies `B`"; if neither `A` nor `B` is a proposition, `A->B` intuitively stands for the function type from `A` to `B`.

`[x:A]M` is the Lego notation for ordinary λ-form ($\lambda\, x : A.M$) which intuitively denotes a function which, whenever applied to an element a of type `A`, returns the value of $[a/$`x`$]$`M` as its value.

Σ-types, pairs and projections `<x:A>B` is the Lego notation for ordinary (dependent) strong sum types (Σ-types $\Sigma x : A.B$). `<x:A>B` intuitively denotes the set of pairs (a, b) such that a is in the set denoted by `A` and b is in that

[1] `[a/x]B` is the term got by substituting all free occurrences of `x` by `a` in `B`.

denoted by B(a). A#B is the abbreviation of <x:A>B when x does not occur free in B which denotes the usual product type ($A \times B$ in the usual notation).

a,b is the form of a pair in Lego which is formed by operator ,. The principal type of a,b is A#B where A and B are the principal types of a and b, respectively. One can use *type-casting* to ensure that a pair has intended dependent Σ-type as its principal type.

M.1 and M.2 are the two projections which intuitively extract the first and second component of M respectively.

Strong sum types can be used to express abstract structures and mathematical theories. See this dissertation for a number of examples.

Local definitions [x=M]N is the Lego notation for local definitions (cf., the let-construct *let* $x = M$ *in* N in functional languages).

Type-castings Every (well-typed) term M can be cast with one of its types A to form a new (well-typed) term M:A. Intuitively, M:A is just the term M with the only exception that its principal type is A which may be different from that of M.

Leibnitz equality (Q) This equality, along with proofs that it is reflexive, symmetric, and transitive, are part of the 'Logic' file that is customarily loaded into Lego. It has the following definition:

```
[Q [A|Type] [x,y:A] = {P:A->Prop}(P x)->(P y)]
```

Discharging of identifiers Proofs may become tedious and definitions unreadable if they are cluttered by irrelevant polymorphism. Lego makes it possible to avoid this by declaring identifiers and then *discharging* these declared identifiers after developing the desired proofs or definitions. The effect of the command Discharge A, where A is some previously declared identifier, is that all identifiers declared back to A are forgotten (removed from the context) but at the same time, all defined constants which make use of the discharged identifiers are 'parameterised' over these identifiers.

Bibliography

[1] Hayes, I.J. (Ed.). *Specification case studies.* Prentice-Hall International 1987.

[2] Hayes, I.J. *Examples of Specification Using Mathematics.* in [1].

[3] Flinn, B., Sorensen, I.H. *CAVIAR: A case study in specification.* in [1].

[4] Spivey, J.M. *Understanding Z: a specification language and its formal semantics.* Cambridge University Press 1988.

[5] Spivey, J.M. *An introduction to Z and formal specifications* Software Engineering Journal, January 1989.

[6] Spivey, J.M. *The Z notation: a reference manual.* Prentice-Hall International 1989.

[7] Sufrin, B. (Ed.). *The Z handbook.* Programming Research Group, Oxford University Computing Laboratory, Oxford University 1986.

[8] Taylor, Paul.*Playing with Lego: Some Examples of Developing Mathematics in the Calculus of Constructions.* Laboratory for Foundations of Computer Science, University of Edinburgh 1989.

[9] Coquand, T., Huet, G. *The Calculus of Constructions.* Information and Computation, vol 76, 1988.

[10] Coquand, T., Huet, G. *Constructions: A Higher Order Proof System for Mechanising Mathematics.* Proc. EUROCAL 85, LNCS 203, Springer 1985.

[11] Coquand, T., *An Analysis of Girard's Paradox*, LICS'87

[12] Luo, Z. *ECC, An Extended Calculus of Constructions.* 4th Symposium on Logic in Computer Science, IEEE 1989.

[13] Luo, Z., Pollack, R., Taylor, P. *How to Use Lego: a preliminary user's manual.* Preprint, LFCS, Department of Computer Science, Edinburgh University 1989.

[14] Pollack, R. *The Theory of Lego*, Manuscript. 1988.

[15] Burstall, R., *Computer Assisted Proof for Mathematics: an Introduction, using the Lego Proof System.* Draft version, LFCS, Department of Computer Science, Edinburgh University 1989.

[16] Burstall, R., McKinna J.H. *Deliverables: an idea for correct program development.* Proc. 2nd Annual Meeting of the BRA "Logical Frameworks" Antibes 1990.

[17] Harper R., Honsell F., Plotkin G. *A Framework for defining Logics.* LICS'87.