LSL traits for using Z with Larch

Hua Zhong

TR #97-22
December 1997

Keywords: formal methods, specification, Larch, Z, LSL, Larch/C++, mathematical toolkit, debugging,
1997 CR Categories: D.2.1 [*Software Engineering*] Requirements/Specifications — languages, tools; F.3.1 [*Logics and Meanings of Programs*] Specifying and Verifying and Reasoning about Programs — assertions, invariants, Pre- and post-conditions, specification techniques.

Partions will be Submitted

Department of Computer Science
Iowa State University
Ames, Iowa 50011, USA

Contents

List of Tables

List of Figures

ACKNOWLEDGEMENTS

I would like to acknowledge the lots of support and patience of my advisor Dr. Gary T. Leavens. I always find him a real gentleman and very kind person, and helped me a lot on my research. I am also very grateful to my committe members Dr. R. C. Sekar and Dr. Jonathan D.H. Smith.

Thanks for the love of my parents, my husband Jianjun Chen and my newborn son Michael Hanfei Chen. Without them, nothing will really happen.

Abstract

Z and Larch-style languages are two kind of specification languages that are used for software design. Z is more simple and abstract, while Larch-style behavioral interface specification languages can specify more detail about the interface of a module written in a specific programming language. In this paper, I present Larch Shared Language traits that define the equivalent of the Z mathematical toolkit. These traits can be used by people familiar with the Z mathematical toolkit who wish to write interface specifications in a Larch-style language. I also show how to use these traits to easily translate Z specifications into Larch-style interface specifications. Some of the traits were debugged using the Larch Prover, and I present a description and evaluation of that process, with examples.

Chapter 1

INTRODUCTION

1.1 Background

Software design is a very important step for building a software system, especially a large software system. In this process, software designers should not focus on the details of the final implementation, instead they should focus on the overall aspects of the system. Formal specification because it focuses attention on the behavior of a system, can play important role in software design.

Mathematical notations are generally used in formal specification to precisely define the properties that users want to have for their system. But details of how these properties will be implemented are avoided. Thus a good abstraction can be achieved without any restrictions on future implementation.

Natural language is not precise compared to a formal specification language. It has ambiguous meaning in some cases. At the same time, using a formal specification language allows for automated verification and checking. So a formal specification language is required for precisely writing down the thoughts of the designer,

There are many kinds of formal specification languages. In this paper, we only talk about two languages: Z (pronounced "zed") and Larch-style specifications.

Z and Larch-style specifications both follow the idea of model-oriented specification, which defines abstract data types by describing a set of constants and functions. We call these functions trait functions. For example, we can define the abstract data type *Stack* using a constant *emptystack*, plus trait functions like *pop* and *push*. All the other abstract values can be generated by using a sequence of *pop* and *push* applied to the *emptystack*.

1.1.1 Z

"Z is a specification language which is based on the set theory and first order predicate logic" [1]. It has a predefined mathematical toolkit which defines sets, relations, functions, sequences and bags. This mathematical toolkit serves the

$$Stack \mathrel{\widehat=} stk \in \operatorname{seq} Object$$
$$stk\textit{I}\textsc{nit} \mathrel{\widehat=} \langle\,\rangle$$

$$\begin{array}{|l}
\hline
\textit{Push} \\
\Delta Stack \\
o? : Object \\
\hline
stk' = stk \frown \langle o? \rangle \\
\hline
\end{array}$$

$$\begin{array}{|l}
\hline
\textit{Pop} \\
\Delta Stack \\
o! : Object \\
\hline
stk' = \textit{front}\ stk \land \\
o! = \textit{tail}\ stk \\
\hline
\end{array}$$

Figure 1.1: A small example for Z

basis for specification and also for user-defined extensions of its mathematical vocabulary. Take *Stack* as an example. It is defined in Figure 1.1.

In this example, *Stack* is treated as a *sequence*, and *emptystack* is represented by *empty sequence*.

The notation $\mathrel{\widehat=}$ shows that the definition of *Stack* is syntactically equivalent to $stk \in \operatorname{seq} Object$.

$stk\textit{I}\textsc{nit} \mathrel{\widehat=} \langle\,\rangle$ shows the initial state of stk is a empty sequence. $\langle\,\rangle$ represents a empty sequence, and notation $\textit{I}\textsc{nit}$ is used to define the initial state of a object.

Following is a schema of Z. It defines a schema *Push*.

$$\begin{array}{|l}
\hline
\textit{Push} \\
\Delta Stack \\
o? : Object \\
\hline
stk' = stk \frown \langle o? \rangle \\
\hline
\end{array}$$

The part above the center line is the declarations. $\Delta Stack$ shows that the , for this operation *Push*, the state of `stk` before the operation and after the operation have the type as it defines in *Stack*. So they are all sequences. And it also tells *stk* might be changed. The object name follow by "?" means this object is input, and when it is followed by "!", it is a output object. The next line $o? : Object$ tells the input object o has the type *Object*.

The part below the center line is the predicates. It tells which properties should hold after this *Push* operation. In this example, the predicate $stk' =$

stk $\frown \langle o? \rangle$ shows the *stk* after is the *stk* before concatanate the sequence $\langle o? \rangle$.

Then, the operation *Pop* is also defined use a schema. It is quite similar as the *Push* operation.

And the behavior of *Push* is to concatenate the new object at the end of the sequence, and *Pop* is to get the front part of the sequence (the sequence with all the objects of the original sequence except the last one), and output the last object, which is the tail of the sequence.

1.1.2 Larch-style specification languages

A Larch-style specification language is different from other specification languages in its structure. It has two "tiers": a behavioral interface specification language (BISL) and Larch Shared Language (LSL). LSL is shared by all Larch-style specification languages. Its goal is to define all the mathematical vocabulary for each BISL. Each BISL is tailored to a specific programming language. It uses the idea of *preconditions* (the allowed states before the call) and *postconditions* (the result state after the call) to specify the modules of programs.

Larch/C++ is a Larch-style BISL that is tailored to C++. It is used to specify behaviors for C++ modules. Such specifications are convenient for the further software development that uses C++.

As an example for Larch-style language, we also look at a Stack specification written in Larch/C++. It is given by Figures 1.2 and 1.3.

The LSL trait in Figure 1.2 shows the mathematical models for *push*, *pop*, and *top*.

The *introduces* part is used to show the signature of each trait function. The *asserts* part tells the predications for these traits function. The *implies* part shows the claims that follows the assertions in the *asserts* part. Usually, LSL traits might have another *includes* part before the *introduces* part. It defines the traits that is used in this trait.

We take the trait function *push* as a example. the *introduces* part defines it takes two parameters: A STK[OBJ] and a OBJ, and it gives back a STK[OBJ]. In the *asserts* part, the predicate *pop(push(stk,o))== stk;* shows when *pop* after a *push*, it gives back the old stack. The predicate *stk \neq emptystack => push(pop(stk),top(stk)) = stk;* illustrates when *push* the *top(stk)* to the result of *pop(stk)* is gives back the *stk* when *stk* is not empty stack.

The *converts clause* in the *implies* part, the trait functions *push*, *pop* and *top* is converted except *pop(emptystack)* and *top(emptystack)*.

And Larch/C++ in Figure 1.3 specifies a C++ class and the behaviors of the operations *Push* and *Pop*. These definitions clarify that which object can be modified by each operation, and after being modified, what condition it should hold.

In a C++ header file prepared for Larch/C++, C++ comments that begin with `//@` or `/*@` are picked up by Larch/C++ as part of the specification.

The keyword *abstract* shows no constructors are provided by this class *stack*.

Since trait functions *push* and *pop* are used when define the C++ function *Push* and *Pop*, the clause *uses MyStack(int)* tells the where to find the definition

```
MyStack(OBJ): trait
  introduces
    emptystack: ->STK[OBJ]
    push: STK[OBJ], OBJ -> STK[OBJ]
    pop: STK[OBJ] -> STK[OBJ]
    top: STK[OBJ] -> OBJ

  asserts
    STK[OBJ] generated by emptystack,push
    \forall stk: STK[OBJ], o: OBJ
    pop(push(stk,o))== stk;
    stk \neq emptystack =>
      push(pop(stk),top(stk)) = stk;

  implies
     converts
      push: STK[OBJ], OBJ -> STK[OBJ],
      pop: STK[OBJ] -> STK[OBJ],
      top: STK[OBJ] -> OBJ
      exempting pop(emptystack), top(emptystack)
```

Figure 1.2: LSL Part

of trait functions *push* and *pop*.

The next line *spec template⟨ class OBJ⟩ class STK;* gives the hypothesis that the class STK takes a template parameter OBJ. And the *spec* clause *spec STK⟨ int⟩ stk* supposes the object stk has the type *STK⟨ int⟩*, which is a stack of integer.

Then, the file begins to define the member functions of class *stack*.

The *modifies* clause tells only the object *stk* is allowed to change by the member function *Pop*.

The clauses *ensures returns /\ stk'=pop(stk^) /\ result=top(stk^);* shows that after the call the function should throw no exception, stk should get the value pop(stk^) and this function returns the value top(stk^). (stk^ is the value of stk before this function call.)

1.2 Problem definition

Since Larch-style BISLs are tailored to a specific programming language, each BISL can thus specify details of program modules written in the language it is tailored to. And the separation of two tiers can allow users to focus first on the mathematical models (when writing LSL specification) and then on details of the

```
/*@ abstract @*/class stack{
  //@ uses MyStack(int);
  //@ spec template <class OBJ > class STK;
  //@ spec STK<int>  stk;

  virtual int Pop();
  //@ behavior {
  //@  requires stk^ ~=emptystack;
  //@  modifies stk;
  //@  ensures returns /\ stk'=pop(stk^) /\ result=top(stk^);
  //@ }

  virtual void Push(int i);
  //@ behavior {
  //@  modifies stk;
  //@  ensures returns /\ stk'=push(stk^,i);
  //@ }
};
```

Figure 1.3: Larch/C++ Part

interface and the module's behavior. So after finishing the BISL specification, the definition will be more precise and thus easier to use for further software development steps.

For example, in the example that given in Figure 1.3, Larch/C++ can tell the operation *Push* and *Pop* are virtual functions, and by using the notation *returns* in *ensures* clause, it also tells the two operations do not throw exceptions. But Z, there is no way to make these kind of claim, because Z is defined for general purpose language.

With the advantage of using Larch-style specification languages, the users who use Z might want to get the benefit of the two tier separation and still keep using the mathematical vocabulary in the Z mathematical toolkit.

And we also want to explore the differences between Z and Larch, which might be helpful for understanding these two languages and their further improvement. The above two considerations are the motivations for this project.

1.3 Solution

For providing more convenience to Z users to use Larch, I have written LSL traits for all the abstract concepts in the mathematical toolkit of Z. These LSL traits can be used for any one of Larch-style specification languages. In this paper I use Larch/C++ as an example. Chapter 2 tells some more details of these traits. Chapter 3 gives some examples of how to use these traits in Larch/C++. Finally, Chapter 4 gives some examples of how to use theorem proving for LSL traits, which was used to help debug these traits. Chapter 5 gives some ideas gained from my work.

Chapter 2

LSL Traits for Z

The traits I present in this chapter cover the whole mathematical toolkit of Z. Since the purpose of writing these traits is to allow users using Z mathematical toolkit in Larch, most of these traits are defined in the same way as they are defined in Spivey's book [9]. In addition, we also use the notation for Z in LaTeX for each of our operations in these traits. Thus, these traits can be used in the same way as the Z mathematical toolkit.

2.1 Overview of the Traits

*Set*s are the basis of most of the other abstract data types in Z.

A *Relation(X, Y)* is a *set* of pairs. In this set, each element is a pair. The first component has the type X and the second one has the type Y. We call such a pair a *mapping* from X to Y.

A *Function(X, Y)* is a kind of *relation* mapping from X to Y, which has the partial function property. This property means that no element in X will be mapped to two different elements in Y. If f is a *function*, then $f(x)$ can only have one value. But this does not require that all the elements in X have images in Y, hence these mappings are, in general, partial.

A *sequence(X)* is a *function* that maps from 1 to n to an element in X.

A *bag(X)* is a *function* which maps each element of type X in the bag to a positive integer and maps all the other elements in X to zero.

Tables 2.1, 2.2, 2.3, 2.4, 2.5, 2.6, and 2.7 given the names of the traits and the operations defined in the traits.

2.2 One trait

Now, let's take a closer look at one of these traits. Take the trait *ZSet* in Figure 2.1 as an example. It is translated from the definition on page 90 of Spivey's book [9]. *ZSet* defines subset, proper subset, power set, and non-empty power set operations.

```
% @(#)$Id: ZSet.lsl,v 1.13 1997/10/03 02: 12: 45 hzhong Exp $
% Written by Hua Zhong with the help of Dr. Gary T. Leavens
% page 56,90 of "the Z Notation:  A Reference Manual" Second Edition,
%     by Spivey, Prentice Hall,1992

ZSet(X): trait
  includes ZPowerSet(X),ZPowerSet(P[X]),
    DerivedOrders(P[X],\subs for <=,\psubs for <,
         \sups for >=,\psups for >)
  introduces
    __ \subs __ : P[X], P[X] -> Bool   %subset relation
    \pset __: P[X] -> P[P[X]]          %powerset of a set
    \psetone __ : P[X] -> P[P[X]]      %non-empty subsets of a set

  asserts
    \forall s,t: P[X], x: X
      s \subs t == (\A x (x \mem s => x \mem t));
      s \mem (\pset t) == s \subs t;
      t \mem (\psetone s) == t \mem (\pset s) /\ ~(t = emptyset);

  implies
    Reflexive(\subs, P[X] for T),
    Irreflexive(\psubs, P[X] for T),
    Antisymmetric(\subs, P[X] for T),
    Asymmetric(\psubs, P[X] for T),
    Transitive(\subs, P[X] for T),
    Transitive(\psubs, P[X] for T)
    \forall s,t,v: P[X], x,y: X
      emptyset \subs s;
      emptyset \psubs s == ~(s =emptyset);
      (\psetone s) = emptyset == s = emptyset;
      ~(s = emptyset) == s \mem (\psetone s);
    converts
      __ \subs __ : P[X], P[X] -> Bool,
      __ \psubs __ : P[X], P[X] -> Bool,
      \pset __: P[X] -> P[P[X]],
      \psetone __ : P[X] -> P[P[X]]
```

Figure 2.1: ZSet.lsl

The first several lines, which begin with %, are some comments for this trait.

The following line is used to define the name of this trait and its parameters. Usually for parameters, we put those sorts that will be possibly renamed when one uses this trait.

The *includes* section is used to illustrate that from which traits this trait inherits the properties. For example, the trait *ZPowerSet* defines the empty set and universe for a sort X. In *ZSet*, we include *ZPowerSet(X)* and *ZPowerSet(P[X])*. In this way we have defined the empty set and universe for sorts X and P[X]. (Of course, *ZPowerSet* also defines some other operations.)

The *introduces* section defines the signature for all these operations. Usually this part is directly translated from the declaration part (above the center line) from the Z definition.

The *asserts* section gives the axioms that we use to define the abstract data type that this trait represents. The assertions should include all the operations in the *introduces* section. This part usually is translated from the predicate part of a Z schema (below the center line).

Originally, $\subseteq, \subset$ are defined in Z like this,

$$
\begin{array}{|l}
[X] \\
\hline
\_ \subseteq \_, \_ \subset \_ : \mathbb{P}\, X \leftrightarrow \mathbb{P}\, Y \\
\hline
\forall\, S, T : \mathbb{P}\, X \bullet \\
(S \subseteq T \Leftrightarrow (\forall\, x : X \bullet x \in S \Rightarrow x \in T)) \wedge \\
(s \subset T \Leftrightarrow S \subseteq T \wedge s \neq T) \\
\hline
\end{array}
$$

For $\subseteq$ (\subs), we use the same definition. Only some syntax is changed; e.g., $\forall$ is converted to \A, and $\Leftrightarrow$ is converted to == . We use a different way of defining $\subset$ (\psubs), because in Guttag and Horning's Larch handbook [3], there is a trait called *DerivedOrders* that can be used. So, we define $\subset$ using DerivedOrders. Meanwhile we add two more operations: $\supseteq$ (\sups) and $\supset$ (\psups).

$\mathbb{P}$ (\pset) is defined in page 56 of Spivey's book [9]. There is no Z definition of it. It is described by the sentence: "If S is a set, $\mathbb{P}$ S is the set of all subsets of S." From the meaning of this sentence, its Larch definition is given by using the operation \subs.

$\mathbb{P}_1$ is originally defined as follows in Z:

$\mathbb{P}_1\, X == \{S : \mathbb{P}\, X \mid S \neq \varnothing\}$

So we change it to LSL as shown in the definition of $\mathbb{P}_1$ in Figure 2.1.

The last part is the *implies* section, in which one gives the theorems that this abstract data type should have. This part is used for checking that the trait defines the desired theory. We can use proof assistants like the Larch Prover (LP) to prove these theorems. We will talk in more detail about LP in Chapter 4. In the Z traits I have written, the *implies* section usually comes from the laws part in Spivey's book.

All the other traits are written in a similar way. Because of the limitation of time, some of Spivey's laws have not being added to our traits.

2.3 Summary of These Traits

Thus far, I have defined all of the Z mathematical toolkit using LSL traits. These traits appear in Appendix A. Thus users can use the Z mathematical toolkit for writing Larch/C++ specifications.

A summary of these traits is given in Tables 2.1, 2.2, 2.3, 2.5, 2.4, 2.6, and 2.7.

Table 2.1: Sets

Trait Name	Page in [4]	Operations	Symbols in Z
ZPowerSet	89,90	\mem	$\in$
		\nem	$\notin$
		emptyset,varemptyset	$\varnothing$
		`{ __ }`	$\{a\}$
		`{ __ , __ }`	$\{a, b\}$
		`{ __, __ ,__ }`	$\{a, b, c\}$
ZSet	90	\subs	$\subseteq$
		\pset	$\mathbb{P}$
		\psetone	$\mathbb{P}_1$
SetUnionIntersectDiff	91	\int	$\cap$
		\uni	$\cup$
		\setminus	$\setminus$
GeneUniInt	92	\duni	$\bigcup$
		\dint	$\bigcap$

Table 2.2: Relations

Trait Name	Page in [4]	Operations	Symbols in Z
Rel	95	\rel	$\leftrightarrow$
		\map	$\mapsto$
DomRan	96	\dom	dom
		\ran	ran
Id	97	\id	id
Cmp	97	\fcmp	$\fatsemi$
		\cmp	$\circ$
DresRres	98	\dres	$\lhd$
		\rres	$\rhd$
DsubRsub	99	\dsub	$\ndres$
		\rsub	$\nrres$
RelInv	100	\inv	$R^{\sim}$
RelImg	101	\img	$(\!\mid\,\mid\!)$
RelOvr	102	\fovr	$\oplus$
TclRtcl	103	\tcl	$^{+}$
		\rtcl	*

Table 2.3: Functions

Trait Name	Page in [4]	Operations	Symbols in Z
Function	105	\pfun	$\pfun$
		\tfun	$\rightarrow$
		\pinj	$\pinj$
		\tinj	$\rightarrowtail$
		\psur	$\psurj$
		\tsur	$\twoheadrightarrow$
		\bij	$\bij$
		apply	$f(x)$

Table 2.4: Misc

Trait Name	Page in [4]	Operations	Symbols in Z
Func		func	
Pass		pass	
SetMap		map	
Pair	93	first	
		second	
		\prod	$\times$

Table 2.5: Numbers and finiteness

Trait Name	Page in [4]	Operations	Symbols in Z
ZNumber	108	nat	$\mathbb{N}$
		integer	$\mathbb{Z}$
		natone	$\mathbb{N}_1$
		\succ	*succ*
		\upto	n .. m
Iteration	110	iter(0,R)	$iter\,0\,R$
		\iter	R^k
FiniteSet	111	\fset	$\mathbb{F}$
		\fsetone	$\mathbb{F}_1$
		#	$\#$
FinitePartial	112	\ffun	$\mathrel{+\!\!\!\!\!\!\rightarrow\!\!\!\!\!\!\!+}$
		\finj	$\rightarrowtail\!\!\!\!\!\!\!\!+\!\!\!\!+$
SetMinMax	113	\min	*min*
		\max	*max*

Table 2.6: Sequences

Trait Name	Page in [4]	Operations	Symbols in Z
Sequences	115	\seq	seq
		\seqone	seq_1
		\iseq	iseq
ConcatRev	116	\cat	$\frown$
		\rev	*rev*
HeadLastTailFront	117	\head	*head*
		\last	*last*
		\tail	*tail*
		\front	*front*
ExtFilCom	118	\ires	$\upharpoonleft$
		\sres	$\upharpoonright$
		\squash	*squash*
PrefixSuffixIn	119	\prefix	$\subseteq$
		\suffix	suffix
		\inseq	in
DConcat	121	\dcat	$\frown/$

Table 2.7: Bags

Trait Name	Page in [4]	Operations	Symbols in Z
Bags	124	\bag	bag
		\bagcount	*count*
		#	$\sharp$
		\otimes	$\otimes$
BagMemSub	125	\inbag	in
		\bagsubs	$\sqsubseteq$
BagUniDiff	126	\buni	$\uplus$
		\bdiff	
BagItems	127	\items	*items*
DisjointPartitions	122	\disjoint	disjoint
		\partitions	partitions
Filter		filter	

Chapter 3

Using Z with Larch

Now, let's see an example of how to use those traits to translate a specification written in Z to Larch/C++.

3.1 Example 1: Symbol Table

The first example is taken from the example first given in page 38-42 of Hayes' book [4]. It defines symbol tables, which are used to record attributes of the symbols in a compiler. In this example, symbol tables are defined as a partial function from type `SYM` to type `VAL`. `SYM` is the type for the symbol, and `VAL` is the type for its attribute.

Since the Z is based of the set theory, the notation for power set is widely used in Z specifications. The file `predefined.h` contains the following lines:

```
//@ uses ZPowerSet(X);
//@ uses Pair(X,Y);
//@ spec template<class X> class P;
//@ spec template<class X, class Y> class pair;
```

This specifies a class `P` that takes one template parameter, and a class pair that takes two template parameters. This makes the class `P` and class `pair` correspond to the sorts `P[X]` and `pair[X,Y]` in the Z traits.

We do not want to put restriction on the types of the symbols and values, so in Larch/C++, we use a template with parameters `SYM` and `VAL`.

The *expects* clause says the trait function contained_objects is defined on the abstract values of `SYM` and `VAL`.

The keyword *abstract* shows that the class `symboltable` is an abstract class. It means no constructors are provided by this class.

```
template <class SYM, class VAL>
  //@ expects contained_objects(SYM), contained_objects(VAL);
  //@ where SYM is {
```

```
  //@   bool operator == (SYM x, SYM y);
  //@   behavior {
  //@     ensures returns /\ result = (x = y);
  //@   }
  //@ };
/*@ abstract @*/class symboltable {
```

The *uses* clause tells what LSL traits this class uses. These traits give the mathematical vocabulary for this class, so the following \pfun, \map ,\mem , and \dom are defined in Function.lsl or those traits it includes, and \fovr is defined in RelOvl.lsl.

The other two *spec* clauses are put here because it is invalid syntax in Larch/C++ to write `universe:P<SYM>`, so we have to rename `P<SYM>` as SetOfSym and SetOfVal.

Then we use another *spec* clause to specify that we suppose `st`, the internal state of symbol table, has the type of `P<pair<SYM,VAL> >`. Thus, in the further specifications, we can talk about how `st` is changed for the *Update*, *LookUp*, and *Delete* operations.

```
  //@ uses Function(SYM,VAL);
  //@ uses RelOvl(SYM,VAL);
  //@ spec typedef P<SYM> SetOfSym;
  //@ spec typedef P<VAL> SetOfVal;
  //@ spec P<pair<SYM,VAL> > st;
```

The *invariant* clause shows what invariant properties always be hold for this class. The following *invariant* clause is converted from the Z definition: $st, st' : SYM \pfun VAL$. It means st is always a partial function that maps from a SYM to a VAL.

```
  //@ invariant st\any \mem (universe:SetOfSym \pfun universe:SetOfVal);
```

In the original example in Hayes's book [4], *Update* is defined in following way:

Update

$$st, st' : SYM \pfun VAL$$
$$s? : SYM$$
$$v? : VAL$$

$$st' = st \oplus \{s? \mapsto v?\}$$

In Larch/C++, we write the *Update* operation in this way:

```
  virtual Update(SYM s,VAL v) throw();
  //@ behavior {
  //@  modifies st;
  //@  ensures st' = (st^ \fovr { s \map v });
  //@ }
```

Larch/C++ requires explicit declaration of all changed objects. Since in the *Update* operation, the state st may be changed, a modifies clause is required, `modifies st;`.

Because s? and v? are input variables, they can be translated to the parameters of the function *Update*. Thus they are defined as formal parameters of the types SYM and VAL in the standard C++ syntax.

Technically, the *requires* clause is what is needed to make the Z predicate well-defined. That is, this is calculated to make the whole specification *protective* in the sense of Leavens's and Wing's paper [8]. Then we can drop those parts of what would be the postcondition that are implied by this precondition. As a rule of thumb: conjuncts that only involve the pre-state variables go into the precondition. However sometimes more is needed.

In the *Update* function, since for any values of s and v, the postcondition is all well-defined, we have no *requires* clause in our translation. The final state st' is the relational override ($\oplus$) of the state st^ before the update operation by the pair $s \mapsto v$. The $\oplus$ operation is defined in the trait RelOvr.lsl (Appendix A.32). The definition is that when b(x) is defined, then $(a \oplus b)(x) = b(x)$, otherwise $(a \oplus b)(x)$ has the value of a(x), if any.

The notation \map is the notation of LaTeX that corresponds to the Z symbol $\mapsto$; and similarly \fovr is used for $\oplus$. As I mentioned in Chapter 2, the LaTeX notation has been used in my traits for the operation name if possible.

In the original example in Hayes's book[4], *LookUp* is defined in following way:

LookUp

$st, st' : SYM \rightarrow\!\!\!\!\!+ VAL$
$s? : SYM$
$v? : VAL$

$s? \in \mathrm{dom}\, st \land$
$v! = st(s?) \land$
$st' = st$

In Larch/C++, this becomes,

```
virtual VAL LookUp(SYM s) throw();
//@ behavior {
//@    requires s \mem (\dom st^);
//@    ensures result = apply(st^,s);
//@  }
```

Because *LookUp* operation requires the input symbol is in the domain of st, we need to have a precondition, otherwise `apply(st^,s)` would not be well-defined. Thus, we need to have a requires clause. In this case, the calculated precondition corresponds to the conjunct $s? \in \mathrm{dom}\, st$ in the Z specification; hence the resulting Larch/C++ specification follows the rule of thumb mentioned earlier.

There is no modifies clause here because no objects should be changed by the *Lookup* operation. The default for Larch/C++ is that an omitted modifies clause means no objects may be changed.

In the original example [4], *Delete* is defined in following way:

Delete

$$
\begin{array}{l}
st, st' : SYM \nrightarrow VAL \\
s? : SYM \\
\hline
s? \in \operatorname{dom} st \wedge \\
st' = \{s?\} \ntriangleleft st
\end{array}
$$

We translate it to the following,

```
virtual void Delete(SYM s) throw();
//@ behavior {
//@    requires s \mem (\dom st^);
//@    modifies st;
//@    ensures st’ = {s} \dsub st^;
//@  }
```

This translation is very similar to the previous operation *Update* and *Lookup*. The poststate value of st is the domain anti-restriction of the set { s } and the state st before the operation *Delete*. Domain anti-restriction ($\ntriangleleft$) is defined like this: “An object x is related to an object y by the relation $S \ntriangleleft R$ holds if and only if x is related to y by R and x is not a member of S. ”[9]

The following is the whole content of our example:

```
#include "predefined.h"
// symbol table
template <class SYM, class VAL>
  //@ expects contained_objects(SYM), contained_objects(VAL);
  //@ where SYM is {
  //@   bool operator == (SYM x, SYM y);
  //@   behavior {
  //@     ensures returns /\ result = (x = y);
  //@   }
  //@ };
/*@ abstract @*/class symboltable {
  //@ uses Function(SYM,VAL);
  //@ spec typedef P<SYM> SetOfSym;
  //@ spec typedef P<VAL> SetOfVal;
  //@ spec P<P<pair<SYM,VAL> > > st;
  //@ invariant st\any \mem (universe:SetOfSym \pfun universe:SetOfVal);
  virtual Update(SYM s,VAL v) throw();
  //@ behavior {
  //@  modifies st;
```

```
  //@  ensures st' = (st^ \fovr { s \map v });
  //@ }
  virtual VAL LookUp(SYM s) throw();
  //@ behavior {
  //@    requires s \mem (\dom st^);
  //@    ensures result = apply(st^,s);
  //@  }
  virtual void Delete(SYM s) throw();
  //@ behavior {
  //@    requires s \mem (\dom st^);
  //@    modifies st;
  //@    ensures st' = {s} \dsub st^;
  //@  }
};
```

3.2 Example 2: Symbol Table with Error Handling

The second example is taken from page 52-57 of Hayes' book [4]. It still talks about symbol tables. But different notations of Z are used for the Z specification, and error handling is added into the specification.

The Z specification is given in Figure 3.1.

In Z, when one uses symbol Δ, one usually means that this object may be modified. And when one uses $\equiv$, one tells this object will not be changed. So very often, the schema with symbol Δ is translated to a function with modifies clause of this object. And the schema with symbol $\equiv$ illustrates that object should not be put into the *modifies* clause of Larch/C++.

In fact, Δ and $\equiv$ are syntactic sugars in Z. For *LookUp*, *Delete*, and *Update*, this desugared forms are as the Z specification in the first example. So, they would get the same translations if error handling were not added.

Usually, Translating the Z specification of the form $X \vee Y$, one can use `also` in Larch/C++ to combine the specifications that translate X and Y. The reason is given as following: suppose we have the following schema:
$X \mathrel{\widehat{=}} A \wedge B$
$Y \mathrel{\widehat{=}} \neg\, A \wedge C$
The disjunction ($\vee$) of two schemas means merging their declarations together, and disjoining their predicate parts (the common variables of the two schemas must have the same type) [9]. So, $Z \mathrel{\widehat{=}} X \vee Y$ is equal to definition $Z \mathrel{\widehat{=}} (A \wedge B) \vee \neg\, A \wedge C$.

In Larch/C++, the definition is

```
void Z()
{
  requires A;
  modifies ...;
```

$ST \mathrel{\widehat=} [\, st : SYM \pfun VAL \,]$
$stI\text{NIT} \mathrel{\widehat=} \{\,\}$

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{LookUp}} \\ \hline
\Xi ST \\
s? : SYM \\
v? : VAL \\ \hline
s? \in \operatorname{dom} st \land \\
v! = st(s?) \\ \hline
\end{array}$$

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{Delete}} \\ \hline
\Delta ST \\
s? : SYM \\ \hline
s? \in \operatorname{dom} st \land \\
st' = \{s?\} \lhd st \\ \hline
\end{array}$$

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{Update}} \\ \hline
\Delta ST \\
s? : SYM \\
v? : VAL \\ \hline
st' = st \oplus \{s? \mapsto v?\} \\ \hline
\end{array}$$

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{NotPresent!}} \\ \hline
\Xi ST \\
s? : SYM \\
rep! : Report \\ \hline
s \notin \operatorname{dom} st \land \\
rep! ='' \textit{Symbol not present}'' \\ \hline
\end{array}$$

$Success \mathrel{\widehat=} [\, rep! : Report \mid rep! = \text{``}OK'' \,]$
$STLookUp \mathrel{\widehat=} (LookUp \land Success) \lor NotPresent!$
$STUpdate \mathrel{\widehat=} (Update \land Success)$
$STDelete \mathrel{\widehat=} (LookUp \land Success) \lor NotPresent!$

Figure 3.1: Z specification for Example 2

```
  ensures B;
 also
  requires ~A;
  modifies ...;
  ensures C;
}
```

This definition is equal to the following definition:

```
void Z()
{
  requires A \/ ~A
  modifies ...;
  ensures (A => B) /\ (~ A => C);
}
```

Since `A => B` = `~A \/ B`, and `~A => C` =`A \/ C`,
the predicate `(A => B) /\ (~A => C)`
= `(~A \/ B) /\ (A \/ C)`
= `((~A \/ B) /\ A) \/ ((~A \/ B) /\ C))`
= `(~A /\ A) \/ (B /\ A) \/ (~A /\ C) \/ (B /\ C)`
= `(~A /\ C) \/ (B /\ A) \/ (B /\ C)`
Compare this result with the Z predicate, there are one more condition `(B /\ C)`. However, since B and C are the results that get from different case `A` and `~A`. Usually `B /\ C = false`, because the reason for the use of A is to separate the two cases that get two different results. And when `B /\ C = false`, the above two specifications for Z and Larch/C++ is equivalent.

The conjunction ($\wedge$) of two schemas means merging their declarations together, and conjoining their predicate parts (the common variables of the two schemas must have the same type) [9].

There is no trivial way to translate Z specification of the form $A \wedge B$ when A and B are two schema that are already defined. But if they have the same declarations, you can translate separately and use *also* in Larch/C++.

One way to do that is to manually merge the declaration of the two schemas, conjunct their predicates, and then do the translation. Another way is first to write a LSL trait for each of the already defined schema, and then use both of them. We will go into more detail of the method in the third example.

In this example, since the schema *Success* is very simple, I combine it with the other operations.

Another thing we need to consider is: how to output those objects that need for output? In this example, the object *rep!* is for output. I define it as a public instance variable of the class *symboltable*.

So, the translation becomes as the following:

```
#include "predefined.h"
// symbol table with error handling
enum RepSignal{OK,Symbol_not_present};
```

```
template <class SYM, class VAL>
  //@ expects contained_objects(SYM), contained_objects(VAL);
  //@ where SYM is {
  //@   bool operator == (SYM x, SYM y);
  //@   behavior {
  //@     ensures returns /\ result = (x = y);
  //@   }
  //@ };
/*@ abstract @*/class symboltable {
public:
  //@ uses Function(SYM,VAL);
  //@ spec typedef P<SYM> SetOfSym;
  //@ spec typedef P<VAL> SetOfVal;
  //@ spec P<P<pair<SYM,VAL> > > st;
  //@ enum RepSignal rep;
  //@ invariant st\any \mem (universe:SetOfSym \pfun universe:SetOfVal);

  virtual VAL STLookUp(SYM s) throw();
  //@ behavior {
  //@    requires s \mem (\dom st^);
  //@    modifies rep;
  //@    ensures result = apply(st^,s) /\ rep =OK;
  //@   also
  //@     requires s \nem {\dom st^};
  //@     ensures rep = Symbol_not_present;
  //@  }

  virtual STUpdate(SYM s,VAL v) throw();
  //@ behavior {
  //@  modifies st, rep;
  //@  ensures st' = (st^ \fovr { s \map v }) /\ rep =OK;
  //@ }

  virtual void STDelete(SYM s) throw();
  //@ behavior {
  //@    requires s \mem (\dom st^);
  //@    modifies st, rep;
  //@    ensures st' = {s} \dsub st^ /\ rep =OK;
  //@   also
  //@     requires s \nem {\dom st^};
  //@     ensures rep = Symbol_not_present;
  //@  }
};
```

Another way to translate the specification with error handling is to use exceptions.

The translation with exceptions is given as following:

```
#include "predefined.h"
// symbol table with error handling
enum RepSignal{OK,Symbol_not_present};
template <class SYM, class VAL>
  //@ expects contained_objects(SYM), contained_objects(VAL);
  //@ where SYM is {
  //@   bool operator == (SYM x, SYM y);
  //@   behavior {
  //@     ensures returns /\ result = (x = y);
  //@   }
  //@ };
/*@ abstract @*/class symboltable {
public:
  //@ uses Function(SYM,VAL);
  //@ spec typedef P<SYM> SetOfSym;
  //@ spec typedef P<VAL> SetOfVal;
  //@ spec P<P<pair<SYM,VAL> > > st;
  //@ enum RepSignal rep;
  //@ invariant st\any \mem (universe:SetOfSym \pfun universe:SetOfVal);

  virtual VAL STLookUp(SYM s) throw(RepSignal);
  //@ behavior {
  //@    requires s \mem (\dom st^);
  //@    ensures returns /\ result = apply(st^,s);
  //@   also
  //@     requires s \nem {\dom st^};
  //@     ensures throws(RepSignal) /\
  //@        thrown(RepSignal)= Symbol_not_present;
  //@  }

  virtual STUpdate(SYM s,VAL v) throw();
  //@ behavior {
  //@  modifies st, rep;
  //@  ensures st' = (st^ \fovr { s \map v });
  //@ }

  virtual void STDelete(SYM s) throw(RepSignal);
  //@ behavior {
  //@    requires s \mem (\dom st^);
  //@    modifies st, rep;
  //@    ensures st' = {s} \dsub st^ /\ returns;
  //@   also
```

```
  //@     requires s \nem {\dom st^};
  //@     ensures throws(RepSignal) /\
  //@        thrown(RepSignal)= Symbol_not_present;
  //@  }
};
```

3.3 Example 3: Phone Network

The third example is taken from a complex example that given in page 370-388 of Jackson's paper [5]. It gives a specification of phone network.

In Jackson's paper, he talks about using views to simplify the writing of specification. Two views have been used for the example of a phone network: a phone view and a switch view. The phone view is used to define the state change of a phone. The switch view handles connections between phones.

For simplicity, I only translate the specifications for the operation `Net_Req`. The other operations can be translated in the same way.

The specification of the phone view given as following:

$$[\,Id\,]$$
$$Status ::= ringing \mid idle \mid waiting \mid connected \mid dialtone \mid busytone \mid ringtone$$
$$Phone \mathrel{\widehat{=}} [\,ps : Id \pfun Status\,]$$
$$Phone\_Dial \mathrel{\widehat{=}} [\,\Delta Phones;\ i : Id \mid ps(i) = dialtone \land ps' = waiting\,]$$

$$\begin{array}{|l}
\underline{\ Phones_Frame\ } \\
\Delta Phone \\
c : \mathbb{P}\, Id \\
\hline
\forall\, i : (\mathrm{dom}\, phones \setminus c) \bullet ps(i) = ps'(i) \\
\hline
\end{array}$$

In this example, I give another way of translating Δ. After I defined trait function *Phone*, I defined a trait function *Delta_Phone* that models the syntactic sugar in Z for $\Delta Phone$. Then, one can use *Delta_Phone* in the specification of other operations, or in the predicates in Larch/C++.

I have not put the *Delta_Phone* into each operation in the phone view trait, because it is an invariant property of the phone network. Later we will put it into the *invariant* clause of class phone network of Larch/C++.

The trait function *Phone_Dial_pre* is defined for the use of Larch/C++, it is the precondition of operation *Phone_Dial*.

The following LSL trait is the translation of the phone view:

```
Phoneview(Id):trait

  includes
   Function(Id, STATUS)
```

```
 STATUS enumeration of ringing, idle, waiting, connected,
    dialtone, busytone, ringtone

introduces
  Phone: P[pair[Id,STATUS]] -> Bool
  Delta_Phone: P[pair[Id,STATUS]],P[pair[Id,STATUS]] -> Bool
  Phone_Dial: P[pair[Id,STATUS]],P[pair[Id,STATUS]], Id -> Bool
  Phones_Frame: P[pair[Id,STATUS]],P[pair[Id,STATUS]], P[Id] -> Bool
  Phone_Dial_pre: P[pair[Id,STATUS]], Id -> Bool

asserts
 STATUS generated by ringing, idle, waiting, connected,
    dialtone, busytone, ringtone
\forall ps,ps': P[pair[Id,STATUS]], i: Id,c:P[Id]
 Phone(ps) = ps \mem (universe:P[Id] \pfun universe:P[STATUS]);
 Delta_Phone(ps,ps') = Phone(ps) /\ Phone(ps');
 Phone_Dial(ps,ps',i) == (apply(ps,i) = dialtone) /\
                          (apply(ps',i) = waiting);
 Phones_Frame(ps, ps', c) ==
    \A i ((i \mem ((\dom ps) \setminus c)) =>
       apply(ps,i) = apply(ps',i));
 Phone_Dial_pre(ps,i) == apply(ps,i) = dialtone;
```

Then comes the *Switch* view. It is originally defined as follows:

Switch

$reqconns : Id \pfun Id$
$conns : Id \pfun Id$

$conns \subseteq reqconns$
$\mathrm{dom}\, conns \cap \mathrm{ran}\, conns = \varnothing$

Switch_Request

$\Delta Switch$
$from, to : Id$

$reqconns' = reqconns \cup \{from \mapsto to\}$
$conns' = conns$

In the same way as the *Phone* view, I also put it into a LSL trait. For the same reason as the *Phone* view, I have not put the *Delta_Switch* into each operation in the switch view trait, because it is also an invariant property of the phone network.

There is no precondition for the operations in the switch view, so there is no need to generate extra trait functions for the switch view.

```
switchview(Id):trait

   includes
     Function(Id,Id)

   introduces
     Switch: P[pair[Id,Id]],P[pair[Id,Id]]-> Bool
     Delta_Switch: P[pair[Id,Id]],P[pair[Id,Id]],P[pair[Id,Id]],
              P[pair[Id,Id]]-> Bool
     Switch_Request: P[pair[Id,Id]],P[pair[Id,Id]],P[pair[Id,Id]],
              P[pair[Id,Id]], Id, Id -> Bool
   asserts
     \forall conns,reqconns,conns',reqconns' : P[pair[Id,Id]],
        to,from : Id
       Switch(conns,reqconns) == (conns \subs reqconns) /\
           ((\dom conns) \int (\ran conns)) =emptyset:P[Id];
       Delta_Switch(conns,conns',reqconns,reqconns') ==
         Switch(conns,reqconns) /\ Switch(conns',reqconns');
       Switch_Request(conns,conns',reqconns,reqconns',from,to) ==
         reqconns'=reqconns \uni {from \map to} /\
         conns'=conns;
```

Now, after we have these two views, we get our specifications for the phone network.

The following is the original Z specification of the phone network.

$$
\begin{array}{l}
\mathit{Net} \mathrel{\widehat{=}} \mathit{Phones} \land \mathit{Switch} \\
\mathit{Net\_Req} \mathrel{\widehat{=}} \\
\qquad \Delta \mathit{Net} \land (\exists\, c : \mathbb{P}\, \mathit{Id} \mid c = \{\mathit{from}\} \bullet \mathit{Phone\_Frame}) \\
\qquad \land\ \mathit{Switch\_Request} \land \mathit{Phone\_Dial}\,[\mathit{from}/i\,]
\end{array}
$$

In Larch/C++ BISL, there is no way to and two defined operations together to generate another operation as the Z specification of phone network does. However, in LSL, one can find a way to do this. So, one can use traits to define the views and use these notations in the views in other LSL traits or Larch/C++. But in LSL there is no concept of state, they are only pure mathematical specifications. So one needs to put the states into the parameters of operations.

Until this step, the main task for writing the Larch/C++ specification is to calculate the objects that can be modified and the properties that are invariant for the class. Other parts come from directly translation.

```
#include "predefined.h"

// phone network
template <class Id>
  //@ expects contained_objects(Id);
```

```
  //@ where Id is {
  //@   bool operator == (Id x, Id y);
  //@   behavior {
  //@     ensures returns /\ result = (x = y);
  //@   }
  //@ };
/*@ abstract @*/class phone_net {
  //@ uses Phoneview(Id);
  //@ uses switchview(Id);
  //@ spec P<pair<Id,Id> >  conns;
  //@ spec P<pair<Id,Id> >  reqconns;
  //@ spec P<pair<Id,Status> >  ps;
  //@ invariant Phone(ps\any)/\ Switch(reqconns\any,conns\any);

  virtual Net_Req(Id from, Id to) throw();
  //@ behavior {
  //@  requires Phone_Dial_pre(ps,from);
  //@  modifies ps,reqconns;
  //@  ensures Phone_Frame(ps,ps',{from})/\
  //@   Switch_Request(conns,conns',reqconns,reqconns',from,to)/\
  //@   Phone_Dial (ps,ps',from);
  //@ }

  //  ...

};
```

3.4 General Steps for Translation

The following is a summary of the steps that are used to translate Z specification into Larch/C++:

- If there are some views or some predefined concepts in Z, use LSL traits to define those.
- Give the necessary traits in the *uses* clauses.
- For all the given types used in the Z specification,make them parameters of the C++ *template*.
- If there are properties that should hold for all operations, for example properties of Z types, put them into *invariant* clauses.
- Use the *spec* clauses to specify the type of internal state of the class's object.
- Then, for each function:

- Put the schema's predicate into the *ensures* clause as the postcondition, changing `o` to `o^` for all object o.
- Calculate the precondition so as to make all predicates in the postcondition well-defined, and put these preconditions into the function's *requires* clause.
- Find out all the objects that can be changed by the function and list them in the *modifies* clause.

• The "optimizations" below are optional. They help the specification looks better.

- Eliminate conjuncts in the post-condition that are redundant with the precondition.
- Eliminate conjuncts of the form `o^ = o'` in the post-condition that are redundant, since the *modifies* clause must have already put restriction on that.
- Use *also* if possible, to break the specification into cases.
- Move the common predicates into *invariant* clauses.

Chapter 4

Debugging Traits with the Larch Prover

As we already know, specification languages are used for the design phase of software development. If one uses the "waterfall" model, then all the other phases of software development are based on the result of design phase, so these specifications are extremely important. Even if one uses more evoluterary approaches, specifications are still important for recording designs.

The goal of specification is to convert the features of a software system, which are ideas in a person's mind, into precise language. Obviously, there is no way to formally ensure the correctness of the specification. No absolute standard can be used to judge correctness. So what can one do?

Parsing and type-checking are one way to find some errors in specifications. But they can not find logical errors.

The specification is written abstractly. So in general there is no way to test the correctness of the specification by executing it. However, one can "test" the specification against one's intuition to "debug" it. After writing the specification, one can give some theorems that intuitively follow from it. This makes it possible for one to find the bugs in a specification by trying to prove these theorems from the original axioms of the specification[3].

The Larch Prover (LP) is an interactive theorem proof assistant that can help in debugging Larch-style specification. Galand and Guttag's paper [2] and Chapter 7 of Guttag and Horning's book [3] talk in more detail of how to debug traits using LP.

I have used LP to help debug some of the traits I present in this paper. I now describe the process of using LP for debugging LSL traits.

After the LSL checker finishes checking for the parse and type errors of a trait, it also generates some files for LP to use. These files are LP script files. LP scripts add the assertions of the trait to LP and instruct LP to try to prove each of the trait's implications. LP might stop at some place, and wait for the person to give it more guidance. Because LP is not designed for handling difficult proofs automatically, it requires the person's assistance at such points.

4.1 Example 1: prove for ZPowerSet.lsl

The following is a proof I have done for the trait: ZPowerSet.lsl (Appendix A.39) . The whole content of the proof is given by Appendix B.1.

After input of all the axioms, LP has gained the following facts, labeled with names of the form ZPowerSet.n:

```
ZPowerSet.2: x \mem emptyset -> false
ZPowerSet.3: varemptyset -> emptyset
ZPowerSet.4: x \mem universe -> true
ZPowerSet.5: x \nem p -> ~(x \mem p)
ZPowerSet.6: x \mem {y} -> x = y
ZPowerSet.7: x \mem {y1, y2} -> x = y1 \/ x = y2
ZPowerSet.8: x \mem {y1, y2, y3} -> x = y1 \/ x = y2 \/ x = y3
```

And then when proving the theorem:

```
sort P[X] partitioned by \nem
```

LP generated the following subgoal (this can be printed by typing the command `display proof-status`):

```
Conjecture ZPowerSetTheorem.1: when \A x (x \nem p <=> x \nem p1)
  yields p = p1,  try a proof of deduction rule.
Level 2 subgoal for proof of deduction rule:
  \A x (x \nem p <=> x \nem p1) => p = p1
  Current subgoal: \A x (x \mem p1 <=> x \mem p) => p = p1
  Attempt a proof by normalization.
```

So the current subgoal is : `\A x (x \mem p1 <=> x \mem p) => p = p1`. After we observe this subgoal, we can see that it has the form $A => B$. Thus, using the tips that Garland gives in his LP guide [2], we first try "`resume by =>`". Then this conjecture is proved.

The other part of this proof is easier. It can be finished by using the command `resume by contradiction` and then the command `critical-pairs *Hyp with *`.

4.2 Example 2: proof for State_Basic.lsl

This trait is one of the build-in LSL traits for Larch/C++. I did this proof before I began to write traits for Z. I think this proof is a good example for illustrating how to use LP.

The whole proof is given by Figure B.2. We skip some lines and begin to look at the proof of the following claim:

```
prove
 (isBottom(st)) = (st = bottom)
 ..
```

First by observing this conjecture, we can see that it contains a free variable `st`. And using the LP command `display`, we can find out there is an induction rule for its sort. So I tried the command `resume by induction`. Other commands might cause a free variable to be converted into a constant. (e.g., `resume by case ...`). Thus, when induction is the method for proving the conjecture, most of the time it is the first step of the proof.

The base case is proved automatically. Then the current subgoal is the following claim:

```
(stc = bottom) = (bind(stc, o, v, s2) = bottom)
```

This looks a little complex, and LP seldom can handle complex situations by itself. The goal needs to be simplified. So I used the step `resume by case stc =bottom`.

LP proved the case $stc = bottom$ automatically, and generated the other case ~($stc = bottom$). At this step, the current subgoal becomes :

`~(bind(stc, o, v, s2) = bottom).`

It can be simply proved by trying the command `resume by contradiction` and the command `critical-pairs *Hyp with *`. These two steps are usually used together. Because when we use `resume by contradiction`, LP negates the current subgoal and adds it as a hypothesis; the second command is used to help LP to find some conflicts among the rules. Sometimes, but not often, LP can find the conflicts itself only by normalizing the facts.

The proof of the following theorem is a complex one:

```
(bind(bind(st, obj, v, typs), obj, v1, typs1)) =
        (bind(st, obj, v1, typs1))
```

Before proving this theorem, LP has gained the facts in Figure 4.1.

There is no rule for directly proving when two bind operations get the same result. However, the sort of the result of the bind operation is `State`, so what we are proving is that one `State` equals another; we can think of it as proof of the form `st1 = st2`, where st1 is the left hand side above, and st2 the right. After observing these facts and rules, the one we can use is the deduction rule State_Basics.2. When using the deduction rule, we need to use the `apply` command. However, this command should be the last command of the proof. So, we need to first prepare the conditions for using it. That is, we first need to prove

`\A obj (allocated(obj, st1) <=> allocated(obj, st2))`

and

`\A obj (eval(obj, st1) = eval(obj, st2))`

then we might let LP draw the conclusion that `st1 = st2`.

So, to obtain the second of these, I began with proving the following lemma:

```
prove
```

```
Induction rules:

State_Basics.1: sort State generated by emptyState, bottom, bind


Deduction rules:

State_Basics.2: when \A o (allocated(o, s) <=> allocated(o, s1)),
                      \A o (eval(o, s) = eval(o, s1)),
                      s = bottom <=> s1 = bottom
                yield s = s1


Rewrite rules:

State_Basics.3: allocated(obj, emptyState) -> false
State_Basics.4: allocated(obj, bottom) -> false
State_Basics.5: allocated(obj, bind(st, obj1, v, typs))
                  -> ~(st = bottom) /\ (obj = obj1 \/ allocated(obj, st))
State_Basics.6: ~(st = bottom)
                  => eval(obj1, bind(st, obj, v, typs))
                       = (if obj1 = obj then v else eval(obj1, st))
                  -> true
State_Basics.9: bind(st, obj, v, typs) = bottom -> st = bottom
State_BasicsTheorem.1: emptyState = bottom -> false
State_BasicsTheorem.2: isBottom(st) -> st = bottom
State_BasicsTheorem.4: ~(st = bottom)
                         => eval(obj1, bind(st, obj1, v, typs)) = v
                         -> true
State_BasicsTheorem.5: ~(st = bottom)
                         => eval(obj1, bind(st, obj, v, typs))
                              = (if obj1 = obj
                                    then eval(obj1,
                                              bind(emptyState, obj1, v, typs))
                                    else eval(obj1, st))
                         -> true
State_BasicsTheorem.6: ~(st = bottom)
                         => (~(st = bottom)
                               /\ (obj1 = obj \/ allocated(obj1, st)))
                              = (obj1 = obj \/ allocated(obj1, st))
                         -> true
```

Figure 4.1: The facts gained by LP

```
eval(obj1,bind(bind(st, obj, v, typs), obj, v1, typs1))
   =eval(obj1,bind(st, obj, v1, typs1))
..
```

Again I resorted to searching the facts and rules, to try to find a rule fit for proving this lemma, and State_Basics.6 fell into my eyes. It says that when a certain condition holds, then
`eval(obj1, bind(st, obj, v, typs))`
is equal to
`eval(obj1, st)`.
So, I rewrote it to the form I want by using the command `instantiate st by bind(st, obj, v2, typs2) in State_Basics.6`. The rule State_Basics.6 has been transformed to the following:

```
~(st = bottom)
  => eval(obj1, bind(bind(st, obj, v2, typs2), obj, v, typs))
    = (if obj1 = obj then v else eval(obj1, bind(st, obj, v2, typs2)))
```

Now I separated these cases by using the command `resume by case st~=bottom` and `resume by case obj1 = obj`. And then I used the command `critical-pair * with *Hyp` to tell LP to combine these hypothesis with the fact rules. The case *st~ = bottom* has been successfully proved.

For the case *st = bottom*, the current subgoal is:

```
Current subgoal:
  eval(obj1, bind(bind(bottom, obj, v, typs), obj, v1, typs1))
    = eval(obj1, bind(bottom, obj, v1, typs1))
```

Obviously, if I had a lemma: `bind(bottom, obj, v, typs)= bottom`, this would be easily proved by LP. However, after the proof it is automatically deleted by LP, since it is "normalized to true". This is the time that the command `set immunization on` can show its power. Thus, I used this command before I proved the following lemma.
`bind(bottom, obj, v, typs)= bottom`
Then I turned immunization off next. At this time, the case *st = bottom* is also proved.

Then, I executed the command:

```
prove
  allocated(obj1,bind(bind(st, obj, v, typs), obj, v1, typs1))
     =allocated(obj1,bind(st, obj, v1, typs1))
  ..
```

I separated it to two cases: *st = bottom* and *st~=bottom*. And after that, the command `apply State_Basics.2 to conjecture` leads to the end of the proof of this theorem.

The other part of the proof is similar, and I do not want to talk too much detail about it.

4.3 Discussion

The following is some experience that I gained from my work.

There is often more than one way to prove a theorem in LP, just as in mathematics. Some of them might be very simple and others might be much more complex. If the right way can be found, it will save the person a lot of time.

First, I think the following idea is very important for one who uses LP: do not think LP does its job automatically. It is the person that does the proving work. LP can only finish proving a conjuncture by following the person's guidance. If it can get the result, it means the person's reasoning is absolutely right.

However, when LP has finished proving a set of implications from a trait's assertions, all one can say is that the implications agree with the trait.

Otherwise, if LP gets stuck, then the person may need to find some other way to prove the conjecture. So the person should always keep checking the facts and the current subgoals of LP, and trying to find the rules in the facts that can lead to the current subgoals.

Second, the person should not always believe the correctness of what is being proved. But it is also possible that after a long time, the person still can not find a way to finish the proof. Then, this is the time for him/her to think about whether those assertions can really prove these implications.

Furthermore, the person should pay attention to the difference between free variables and constants. After using command `resume by case ...` some free variables have been changed to constants, and LP adds a "c" character at the end of each such variable. The difference becomes important when proving lemmas. A lemma $xc = y$ is different from $x = y$. Because, in LP, a theorem $f(xc) = g(xc)$ only means for constant xc, $f(xc) = g(xc)$, and a theorem $f(x) = g(x)$ has the same meaning as $\forall x(f(x) = g(x))$. Thus, the x can be replaced by any other variables and constants, while the xc can not. If one can prove $f(x) = g(x)$, one can always prove $f(xc) = g(xc)$. But $f(xc) = g(xc)$ tells much less than $f(x) = g(x)$.

The following step might also be helpful for proving theorems using LP:

- When there are some induction rules in the facts, use the command `resume by induction` first, if possible. Many other commands might cause a free variable to be converted into a constant. (e.g., `resume by cases ...`). Thus, when induction is the method used for proving a conjecture, most of the time it is the first step of the proof.

- When there are some deduction rules in the facts, using the command `apply x to conjecture` might be a good choice. This choice can only be used at the last step of the proof of the current subgoal. The reason is that if this command can not prove the current subgoal, it will have no effect on the proof process. So, all the conditions for using this deduction rule should be prepared first.

- Using the `prove` command to add lemmas is also very helpful. As I have mentioned earlier, be careful when putting constants into the lemma.
- The command `resume by cases ...` is a very useful command, because it can simplify the subgoal. LP can seldom normalize a complex subgoal automatically. For example, when the subgoal has the form $f(a = b) = g(a = b)$, the command `resume by case a = b`, changes the conjecture to two subgoals: $f(true) = g(true)$, and $f(false) = g(false)$. These two subgoal will be more simple than the previous one, and easy to work out for LP.
- Using the command `set immunity on` to avoid having the system automatically use some rewrite rules, and then delete the theorem that it just proved. Sometimes it is useful to use the command `set immunity on` before adding a new lemma, so that this lemma will not be reduced to true and deleted. Then, after proving the lemma, using the command `set immunity off` is also important. If immunization state is kept "on", too many claims will enter the set of facts and this will cause the proof process to be much slower. Sometimes this may even make it impossible to prove a theorem that LP can prove with less facts.

Even after the proof of the trait, there might still have problems in the trait, because it is possible that the assertions of the trait are not consistent. They might contain claims which can prove $true = false$. If this happens to be the case, any implication can be proved. But unfortunately, there is no general way to help us always write consistent traits.

For more details about using LP, see Garland and Guttag's guide for LP [2] and Leavens's Larch FAQ [7].

Chapter 5

Discussion

When I wrote these traits, I found the following differences between Z and Larch that are described in the next two sections.

5.1 LSL differences

The following is some differences between Z and LSL:

- First, a big difference is that the mathematical model of Z is based on set theory. Thus the type system of Z is more flexible [6]. For example, in Z, schema a: X actually has the same meaning as $a \in X$. On the other hand, in LSL the declaration a : X means that, X is the name of a sort. X can not be treated as a set at any time.

  For the convenience of using the Set for a certain sort X, in the LSL trait `ZPowerSet`, there is a constant *universe*. The *universe* is the set that contains all the element of that sort.

  An example is taken from Spivey's book [9, page 97], where the function $\_ \mathbin{;} \_$ is defined as following:

  $$[X, Y, Z]$$
  $$\_ \mathbin{;} \_ : (X \leftrightarrow Y) \times (Y \leftrightarrow Z) \rightarrow (X \leftrightarrow Z)$$
  $$\forall Q : X \leftrightarrow Y;\ R : Y \leftrightarrow Z \bullet$$
  $$Q \mathbin{;} R = \{x : X;\ y : Y;\ z : Z \mid x \underline{Q}\, y \land y \underline{R}\, z \bullet x \mapsto z\}$$

  Following this, in spiveys book, there is a law $\mathrm{id}\, X \mathbin{;} P = P$, and $\mathrm{id}\, X$ has the type $X \leftrightarrow X$. Obviously, Z treats the X,Y, and Z as variables for standing for sets, not the names for types. But LSL is not so flexible. When one talks about `f : A -> C`, the parameter of f can only be A. If the sort of b is B (B and A is not equal), f(b) can not be used without another declaration for f on B.

- Second, in Z all functions can also be treated as sets. For example, $min : \mathbb{P}\, X \to X$ is defined as a set of all the mapping from a subset of X to a minimum element of X in this subset. In LSL, $min : P[X] \to X$ only means that the function min is mapped from a set of sort X to an element of sort X. It is not possible to use min as a set. Another limitation is that in LSL, a parameter of a function can not be another function.

  In my traits, I defined functions in two ways: either as a function in LSL or as a set of pairs that map from one sort to another. Most of the time I only define them as functions. When it is necessary to use the other form, I define them in both ways. There is a trait Graph.lsl (given in Appendix A.21) which can help to do this job. When one needs the set for a function g, he can use command *includes Graph(g)* in his traits and *graph* is the set of all pairs of the form (x,g(x)), where x is a element of x.

- A syntactic problem is that `'\'` followed by a string can not be a constant in LSL. For example, LSL does not allow one to use \emptyset as the symbol for $\varnothing$. So I have used word `emptyset` in my traits instead.

5.2 Larch/C++ BISL differences

In Z, one does not explicitly define what variables can be modified. One can put restriction on the variables that should not be changed, by putting an equation $a = a'$ into the predicate part of a schema, but by default there is no such restriction on modifying any variable. On the contrary, the Larch/C++ modifies clause gives the set of the objects that can be changed. Larch/C++ seems more precise in this aspect of specifications.

Z also does not explicitly define the precondition of an opertion's specification. So when converting a specification from Z to Larch/C++, one needs to calculate the precondition by calculating the domains that make all operations in the schema well-defined, and then one puts the restriction for the domains into the precondition.

In Z, a state of a variable in the predicate part of a function schema, if it is not followed by character `'?'`, `'!'`, or `'''`, refers the state of the variable before calling of this function. It is same as the notation of `'^'` in Larch/C++.

Because of this, in Z, there is also no way to refer to object identity. For example, in Larch/C++, one can have the following specification:

```
void f(int &x,int &y);
  //@ behavior {
  //@  requires x ~= y;
  //@  modifies x,y;
  //@  ensures x'=2 /\ y'=3;
  //@ }
```

But, in Z, there is no way to easily write the constraint `x~=y`, because in Z the names x and y only refer to the value of the two objects, not their identities.

5.3 My recommendations

For LSL, it might be better to have all the functions take a tuple as input parameter instead of taking several parameters, and to allow the more complex sorts to appear in the parameters. For example, let LSL allow functions as parameters. (However, this would be a rather fundamental change in LSL and hence in LP.)

A simple but useful change would be to add the LSL the ability to treat notation \ followed by a string (\emptyset etc.) as a constant.

I would also like to recommend following method to a software designer.

As we have seen before, Z has a more simple model and is more flexible than Larch-style specification languages. Almost all of its concepts are based on the concept of *Set*.

So when a person begins to design software, it is easier to start with writing a Z specification. One does not need to consider too many details about the behavior of the program module, and by using Z one can get a more abstract specification.

But as the design goes on, one will begin to think about more details about the behaviors of the program module, and maybe it is helpful for one to translate it into a Larch-style specification language. So, one can get the power of specifying more detail of the behaviors, and get a more precise specification. This makes the further step of software development easier.

By using the traits in this paper (see Appendix A), the translation from Z to Larch-style specification languages, should not be a big task. It is true that the person needs to consider several more points, but those are the exactly points Larch-style specification languages use to make the specification more precise.

And after the translation, one can use the tools for Larch-style specification languages (for example the theorem prover LP) to help debug the problems the specification.

Appendix A

LSL Traits for Z mathmatical toolkit

In this appendix I present all of the traits that are used to model the Z mathematical toolkit. See Chapter 2 for an overview of these traits.

A.1 BagItems.lsl

```
% @(#)$Id: BagItems.lsl,v 1.2 1997/10/07 02: 55: 51 hzhong Exp $
% Written by Hua Zhong with the help of Dr. Gary T. Leavens
% page 127 of "the Z Notation:  A Reference Manual" Second Edition, by Spivey,
%    Prentice Hall,1992

BagItems(Z, X): trait
includes
  Bags, Sequences, DresRres(Z, X), FiniteSet(Z)
introduces
  \items __ :  P[pair[Z,X]] -> P[pair[X, Z]]

asserts
  \forall x,y: X, i,j: Z, s: P[pair[Z, X]]
    (s \mem (\seq (universe: P[X])))=>
     (\items s) # x = #(\dom(s \rres {x}));
```

A.2 BagMemSub.lsl

```
% @(#)$Id: BagMemSub.lsl,v 1.2 1997/10/07 02: 32: 52 hzhong Exp $
% Written by Hua Zhong with the help of Dr. Gary T. Leavens
% page 125 of "the Z Notation:  A Reference Manual" Second Edition, by Spivey,
%    Prentice Hall,1992

BagMemSub(Z,X): trait
includes
```

```
  Bags
introduces
  __ \inbag __ :  X, P[pair[X,Z]] -> Bool               % Bag Membership
  __ \bagsubs __:  P[pair[X,Z]], P[pair[X,Z]] -> Bool    % Sub-bag relation

asserts
  \forall x: X, b,c: P[pair[X,Z]]
    x \inbag b == x \mem (\dom b);
    b \bagsubs c == \A x ((b # x) <= (c # x));
```

A.3 Bags.lsl

```
% @(#)$Id: Bags.lsl,v 1.4 1997/10/07 02: 28: 27 hzhong Exp $
% Written by Hua Zhong with the help of Dr. Gary T. Leavens
% page 124 of "the Z Notation:  A Reference Manual" Second Edition, by Spivey,
%      Prentice Hall,1992

Bags(Z, X): trait
includes
  ZNumber, Function(X,Z),DresRres(X,Z),RelOvr(X,Z)
introduces
  \bag __:  P[X] -> P[P[pair[X, Z]]]           % Bags
  \bagcount __ :  P[pair[X,Z]]-> P[pair[X,Z]] % Multiplicity
  __ # __ : P[pair[X,Z]], X -> Z
  __ \otimes __ : Z,P[pair[X,Z]]->P[pair[X,Z]] %bag scaling

asserts
  \forall x: X,p: P[X],n: Z, b: P[pair[X,Z]]
    \bag p == p \pfun natone;
    \bagcount b ==  (universe \prod {0}) \fovr b;
    b # x = apply(\bagcount b, x);
    (n \otimes b) # x == n *(b # x);
```

A.4 BagUniDiff.lsl

```
% @(#)$Id: BagUniDiff.lsl,v 1.2 1997/10/07 02: 36: 22 hzhong Exp $
% Written by Hua Zhong with the help of Dr. Gary T. Leavens
% page 126 of "the Z Notation:  A Reference Manual" Second Edition, by Spivey,
%      Prentice Hall,1992

BagUniDiff(Z,X): trait
includes
  Bags,SetMinMax
introduces
  __ \buni __ :  P[pair[X,Z]],P[pair[X,Z]] -> P[pair[X,Z]]    % bag union
  __ \bdiff __ :  P[pair[X,Z]],P[pair[X,Z]] -> P[pair[X,Z]]   % bag difference

asserts
```

```
  \forall x: X, b,c: P[pair[X,Z]]
    (b \buni c) # x == ((b # x)+(c # x));
    (b \bdiff c) # x == \max{(b # x)-(c # x),0};
```

A.5 Cmp.lsl

```
% @(#)$Id: Cmp.lsl,v 1.8 1997/10/03 03: 13: 46 hzhong Exp $
% Written by Hua Zhong with the help of Dr. Gary T. Leavens
% page 97 of "the Z Notation:  A Reference Manual" Second Edition, by Spivey,
%     Prentice Hall,1992

Cmp(X,Y,Z) : trait

  includes Rel(X,Y),Rel(Y,Z),Rel(X,Z),DomRan

  introduces
    % Relational composition
    __ \fcmp __ : P[pair[X,Y]] , P[pair[Y,Z]] -> P[pair[X,Z]]
    % Backward relational composition
    __ \cmp __ : P[pair[Y,Z]] , P[pair[X,Y]] -> P[pair[X,Z]]

  asserts
    \forall x: X, y: Y , z: Z , p: P[X],
            r: P[pair[X,Y]] , q: P[pair[Y,Z]]
      [x,z] \mem (r \fcmp q) == \E y ([x,y] \mem r /\ [y,z] \mem q);
      [x,z] \mem (q \cmp r) == \E y ([x,y] \mem r /\ [y,z] \mem q);

  implies
     \forall r: P[pair[X,Y]] , q: P[pair[Y,Z]]
      r \fcmp q == q \cmp r;
```

A.6 CmpLaws.lsl

```
% @(#)$Id: CmpLaws.lsl,v 1.5 1997/10/07 15: 23: 47 hzhong Exp $
% Written by Hua Zhong with the help of Dr. Gary T. Leavens


CmpLaws(X,Y,Z) : trait

  includes Cmp,Cmp(Y,Z,W),Cmp(X,Z,W),Cmp(X,Y,W),
           Id(X),Id(Y),Cmp(X,Y,Y),Cmp(X,X,Y),Cmp(X,X,X)

  implies
     \forall x: X, y: Y , z: Z , p,p1,p2: P[X],t: P[Y],
            r: P[pair[X,Y]] , q: P[pair[Y,Z]],s: P[pair[Z,W]]
      (r \fcmp q) \fcmp s == r \fcmp (q \fcmp s);
      (\id p) \fcmp r = r;
      r \fcmp (\id t) = r;
```

```
    (\id p1) \fcmp (\id p2) == \id ( p1 \int p2);
```

A.7 ConcatRev.lsl

```
% @(#)$Id: ConcatRev.lsl,v 1.3 1997/10/06 02: 21: 06 hzhong Exp $
% Written by Hua Zhong with the help of Dr. Gary T. Leavens
% page 116 of "the Z Notation:  A Reference Manual" Second Edition, by Spivey,
%      Prentice Hall,1992

ConcatRev(Z,X): trait
includes
  ZNumber, FiniteSet(X),FiniteSet(pair[Z,X])
introduces
  __ \cat __:  P[pair[Z,X]],P[pair[Z,X]]  -> P[pair[Z,X]] % Concatenation
  \rev __:  P[pair[Z,X]] -> P[pair[Z,X]]                  % Reverse

asserts
  \forall s,t: P[pair[Z,X]], x: X, n: Z
    ([n,x] \mem (s \cat t)) ==
        ([n,x] \mem s) \/ ((n \mem (\dom t)) /\([n-(# s), x] \mem t));
    [n,x] \mem (\rev s) == [((#s) -n)+1,x] \mem s;
```

A.8 DConcat.lsl

```
% @(#)$Id: Bags.lsl,v 1.4 1997/10/07 02: 28: 27 hzhong Exp $
% Written by Hua Zhong with the help of Dr. Gary T. Leavens
% page 124 of "the Z Notation:  A Reference Manual" Second Edition, by Spivey,
%      Prentice Hall,1992

Bags(Z, X): trait
includes
  ZNumber, Function(X,Z),DresRres(X,Z),RelOvr(X,Z)
introduces
  \bag __:  P[X] -> P[P[pair[X, Z]]]           % Bags
  \bagcount __ :  P[pair[X,Z]]-> P[pair[X,Z]] % Multiplicity
  __ # __ : P[pair[X,Z]], X -> Z
  __ \otimes __ : Z,P[pair[X,Z]]->P[pair[X,Z]] %bag scaling

asserts
  \forall x: X,p: P[X],n: Z, b: P[pair[X,Z]]
    \bag p == p \pfun natone;
    \bagcount b ==  (universe \prod {0}) \fovr b;
    b # x = apply(\bagcount b, x);
    (n \otimes b) # x == n *(b # x);
```

A.9 DisjointPartitions.lsl

```
% @(#)$Id: DisjointPartitions.lsl,v 1.2 1997/10/07 02: 09: 02 hzhong Exp $
```

```
% Written by Hua Zhong with the help of Dr. Gary T. Leavens
% page 122 of "the Z Notation:  A Reference Manual" Second Edition, by Spivey,
%      Prentice Hall,1992

DisjointPartitions(Z,X): trait
includes
  ZNumber, FinitePartial(Z, P[X]),ZPowerSet(I),SetUnionIntersectDiff(X),
  Function(I,P[X]), SetMap(second for func,pair[I,P[X]] for X, P[X] for Y), Pair,
  GeneUniInt(X)
introduces
  \disjoint __:  P[pair[I,P[X]]] -> Bool           % Disjointness
  __ \partitions __:  P[pair[I,P[X]]], P[X]-> Bool % Partitions

asserts
  \forall s: P[pair[I,P[X]]], x,y,t: P[X], i,j: I
    (s \mem (universe: P[I] \pfun (\pset (universe: P[X])))) =>
      (\disjoint s) =
          (\A i \A j ((i \neq j)=>
             (apply(s,i) \int apply(s,j)) = emptyset));
    (s \mem ((universe: P[I]) \pfun (\pset (universe: P[X])))) =>
       ((s \partitions t) = (\disjoint s) /\ (\duni (map (s)) = t));
```

A.10 DomRan.lsl

```
% @(#)$Id: DomRan.lsl,v 1.8 1997/10/03 02: 29: 35 hzhong Exp $
% Written by Hua Zhong with the help of Dr. Gary T. Leavens
% page 96 of "the Z Notation:  A Reference Manual" Second Edition, by Spivey,
%     Prentice Hall,1992

DomRan(X,Y): trait
  includes
    Rel(X,Y),SetUnionIntersectDiff(pair[X,Y]),
    SetUnionIntersectDiff(X),SetUnionIntersectDiff(Y)
  introduces
\dom __: P[pair[X,Y]] -> P[X]   % Domain of relation
\ran __: P[pair[X,Y]] -> P[Y]   % Range of relation

  asserts
    \forall x: X, y: Y, r: P[pair[X,Y]]
x \mem (\dom r) == \E y ([x,y] \mem r);
y \mem (\ran r) == \E x ([x,y] \mem r);
  implies
    \forall x: X, y: Y, q,r: P[pair[X,Y]]
[x,y] \mem r => x \mem (\dom r);
[x,y] \mem r => y \mem (\ran r);
(x \nem (\dom r)) => \A y ([x,y] \nem r);
(y \nem (\ran r)) => \A x ([x,y] \nem r);

\dom(q \uni r) == (\dom q) \uni (\dom r);
```

```
\ran(q \uni r) == (\ran q) \uni (\ran r);
(\dom(q \int r)) \subs ((\dom q) \int (\dom r));
(\ran(q \int r)) \subs ((\ran q) \uni (\ran r));
\dom emptyset == emptyset;
\ran emptyset == emptyset;
   converts
\dom __: P[pair[X,Y]] -> P[X],
\ran __: P[pair[X,Y]] -> P[Y]
```

A.11 DresRres.lsl

```
% @(#)$Id: DresRres.lsl,v 1.6 1997/10/03 03: 32: 33 hzhong Exp $
% Written by Hua Zhong with the help of Dr. Gary T. Leavens
% page 98 of "the Z Notation:  A Reference Manual" Second Edition, by Spivey,
%     Prentice Hall,1992

DresRres(X,Y): trait
  includes
    Rel(X,Y),Id(X),Id(Y),Cmp(X,X,Y),Cmp(X,Y,Y),DomRan(X,Y)

  introduces
    __ \dres __ : P[X],P[pair[X,Y]] -> P[pair[X,Y]]   % Domain restriction
    __ \rres __ : P[pair[X,Y]], P[Y] -> P[pair[X,Y]]  % Range restriction

  asserts
    \forall x: X,y: Y,s: P[X],t: P[Y], r: P[pair[X,Y]]
      [x,y] \mem (s \dres r) == x \mem s /\ [x,y] \mem r;
      [x,y] \mem (r \rres t) == y \mem t /\ [x,y] \mem r;

  implies
    \forall s,v: P[X],t,w: P[Y],r: P[pair[X,Y]]
    s \dres r == (\id s) \fcmp r;
    s \dres r == (s \prod universe) \int r;
    r \rres t == r \fcmp ( \id t);
    r \rres t == r \int (universe \prod t);

    \dom (s \dres r) == s \int (\dom r);
    \ran (r \rres t) == (\ran r) \int t;

    (s \dres r) \subs r;
    (r \rres t) \subs r;

    (s \dres r) \rres t == s \dres (r \rres t);
    s \dres (v \dres r) == (s \int v) \dres r;
    (r \rres t) \rres w == r \rres (t \int w);
```

A.12 DsubRsub.lsl

```
% @(#)$Id: DsubRsub.lsl,v 1.5 1997/10/03 03: 33: 35 hzhong Exp $
% Written by Hua Zhong with the help of Dr. Gary T. Leavens
% page 99 of "the Z Notation:  A Reference Manual" Second Edition, by Spivey,
%     Prentice Hall,1992

DsubRsub(X,Y): trait
  includes
    Rel(X,Y)

  introduces
    __ \dsub __ : P[X], P[pair[X,Y]] -> P[pair[X,Y]]   % Domain anti-restriction
    __ \rsub __ : P[pair[X,Y]], P[Y] -> P[pair[X,Y]]   % Range anti-restriction

  asserts
    \forall x: X,y: Y,s: P[X],t: P[Y], r: P[pair[X,Y]]
      [x,y] \mem (s \dsub r) == x \nem s /\ [x,y] \mem r;
      [x,y] \mem (r \rsub t) == y \nem t /\ [x,y] \mem r;
```

A.13 ExtFilCom.lsl

```
% @(#)$Id: ExtFilCom.lsl,v 1.4 1997/10/06 21: 54: 50 hzhong Exp $
% Written by Hua Zhong with the help of Dr. Gary T. Leavens
% page 118 of "the Z Notation:  A Reference Manual" Second Edition, by Spivey,
%      Prentice Hall,1992

ExtFilCom(Z,X): trait
includes
  ZNumber, Function(Z,X),DresRres(Z,X),FiniteSet(pair[Z,X]),Cmp(Z,Z,Z),
  Function(Z,Z),RelInv(Z,Z), Graph2(\succ, Z, Z, SuccSet for graph),
  Graph2(<, Z, Z)
introduces
  __ \ires __:  P[Z], P[pair[Z,X]] -> P[pair[Z,X]] % Extraction from a sequence
  __ \sres __:  P[pair[Z,X]], P[X] -> P[pair[Z,X]] % Filtering
  \squash __:  P[pair[Z,X]]-> P[pair[Z,X]]         % Compaction
  compacted_seq: P[pair[Z,X]] -> P[pair [Z, Z]]

asserts
  \forall u: P[X],v: P[Z],f,s: P[pair[Z,X]],p: P[pair[Z,Z]],x,y: Z
    v \ires s == \squash (v \dres s);
    s \sres u == \squash (s \rres u);
    compacted_seq(f) \mem ((1 \upto (# f)) \bij (\dom s));
    (compacted_seq(f) \cmp SuccSet \cmp compacted_seq(f)) \subs graph;
    \squash f = f \cmp compacted_seq(f);
```

A.14 Filter.lsl

```
% @(#)$Id: Filter.lsl,v 1.6 1997/09/28 03: 20: 00 hzhong Exp $
```

```
% Written by Hua Zhong with the help of Dr. Gary T. Leavens

Filter(pass,X): trait

  assumes Pass
  includes ZPowerSet(X)

  introduces
    filter: P[X]->P[X]

  asserts
    \forall x,y: X, p: P[X]
      x \mem filter(p) == pass(x) /\ (x \mem p)

  implies
    \forall x,y: X, p: P[X]
      filter(emptyset) == emptyset;
      (\A x ( (x \mem p) => pass(x)= false))
          => (filter(p)=emptyset);
      x \nem filter(p) == ~pass(x) \/ (x \nem p);
      (\A x ( (x \mem p) => (pass(x)= true)))
          => (filter(p) = p);
    converts filter
```

A.15 FinitePartial.lsl

```
% @(#)$Id: FinitePartial.lsl,v 1.2 1997/10/07 15: 23: 48 hzhong Exp $
% Written by Hua Zhong with the help of Dr. Gary T. Leavens
% page 112 of "the Z Notation:  A Reference Manual" Second Edition, by Spivey,
%      Prentice Hall,1992

FinitePartial(X,Y): trait
  includes
    SetUnionIntersectDiff(P[pair[X,Y]]),
    FiniteSet(X),Function(X,Y)
  introduces
    __ \ffun __ : P[X], P[Y] -> P[P[pair[X,Y]]]    % Finite Partial functions
    __ \finj __ : P[X], P[Y] -> P[P[pair[X,Y]]]    % Finite partial injection

  asserts
    \forall s: P[X],t: P[Y], f: P[pair[X,Y]]
      f \mem (s \ffun t) ==
        f \mem (s \pfun t) /\ ((\dom f) \mem (\fset s));
      s \finj t ==  (s \ffun t) \int (s \pinj t);
```

A.16 FiniteSet.lsl

```
% @(#)$Id: FiniteSet.lsl,v 1.4 1997/10/03 03: 59: 42 hzhong Exp $
```

```
% Written by Hua Zhong with the help of Dr. Gary T. Leavens
% page 111 of "the Z Notation:  A Reference Manual" Second Edition, by Spivey,
%      Prentice Hall,1992

FiniteSet(X): trait
  includes ZSet,ZNumber,
    Function(Z,X),DomRan(P[Z],P[X])

  introduces
    \fset __ : P[X] -> P[P[X]]        % Finite sets
    \fsetone __ : P[X] -> P[P[X]]     % Non-empty finite sets
    # __: P[X] -> Z                   % Number of members of a set

  asserts
    \forall n: Z, s,x: P[X], f: P[pair[Z,X]]
      s \mem (\fset x)
          == (s \subs x) /\ (\E n (\E f ( f \mem ((1 \upto n) \tfun s))
                                        /\ (\ran f = s)));
      s \mem (\fsetone x) == (s \mem (\fset x)) /\ (s \neq emptyset);
      s \mem (\fset x) =>
        (n = (# s)) = \E f ((f \mem ((1 \upto n) \tinj s))
                             /\ ((\ran f) = s));
```

A.17 Func.lsl

```
% @(#)$Id: Func.lsl,v 1.4 1997/09/28 03: 48: 53 hzhong Exp $
% Written by Hua Zhong with the help of Dr. Gary T. Leavens

Func(func,X,Y): trait

  introduces
   func: X ->Y
```

A.18 Func2.lsl

```
% @(#)$Id: Func.lsl,v 1.4 1997/09/28 03: 48: 53 hzhong Exp $
% Written by Hua Zhong with the help of Dr. Gary T. Leavens

Func2(func,X,Y): trait

  introduces
   func: X, Y -> Bool
```

A.19 Function.lsl

```
% @(#)$Id: Function.lsl,v 1.4 1997/10/06 02: 01: 53 hzhong Exp $
% Written by Hua Zhong with the help of Dr. Gary T. Leavens
% page 105 of "the Z Notation:  A Reference Manual" Second Edition, by Spivey,
```

```
%      Prentice Hall,1992

Function(X,Y): trait
  includes
    DomRan(X,Y), SetUnionIntersectDiff(P[pair[X,Y]])
  introduces
    __ \pfun __ : P[X], P[Y] -> P[P[pair[X,Y]]]    %Partial functions
    __ \tfun __ : P[X], P[Y] -> P[P[pair[X,Y]]]      % total functions
    __ \pinj __ : P[X], P[Y] -> P[P[pair[X,Y]]]      % Partial injections
    __ \tinj __ : P[X], P[Y] -> P[P[pair[X,Y]]]      % Total injections
    __ \psur __ : P[X], P[Y] -> P[P[pair[X,Y]]]      % Partial surjections
    __ \tsur __ : P[X], P[Y] -> P[P[pair[X,Y]]]      % Total surjections
    __ \bij __ : P[X], P[Y] -> P[P[pair[X,Y]]]      % Bijections
    apply: P[pair[X,Y]], X -> Y                 % apply x to function

  asserts
    \forall x,x1,x2: X, y,y1,y2: Y,s: P[X],t: P[Y], f: P[pair[X,Y]]
      f \mem (s \pfun t) ==
        \A x \A y1 \A y2 ([x,y1] \mem f /\ [x,y2] \mem f => (y1=y2));
      f \mem (universe: P[X] \pfun universe: P[Y]) =>
         (apply(f,x) = y) = ([x,y] \mem f);
      f \mem (s \tfun t) ==  (f \mem (s \pfun t)) /\ (\dom f = universe);
      f \mem (s \pinj t) ==
        (f \mem (s \pfun t)) /\
        (\A x1 \A x2 ((x1 \mem (\dom f))/\ (x2 \mem (\dom f))) =>
            apply(f,x1)=apply(f,x2));
      s \tinj t == (s \pinj t) \int (s \tfun t);
      f \mem (s \psur t) == (f \mem (s \pinj t)) /\ ((\ran f) = universe);
      s \tsur t == (s \psur t) \int (s \tfun t);
      s \bij t == (s \tsur t) \int (s \tinj t);
```

A.20 GeneUniInt.lsl

```
% @(#)$Id: GeneUniInt.lsl,v 1.5 1997/10/03 02: 21: 51 hzhong Exp $
% Written by Hua Zhong with the help of Dr. Gary T. Leavens
% page 92 of "the Z Notation:  A Reference Manual" Second Edition, by Spivey,
%     Prentice Hall,1992

GeneUniInt(X): trait
  includes SetUnionIntersectDiff(X),SetMap(func,P[X],P[X]),
           SetUnionIntersectDiff(P[X]),Func(func,P[X],P[X])
  introduces
    \duni __ : P[P[X]] -> P[X] % Generalized Union
    \dint __ : P[P[X]] -> P[X] % Generalized Intersection

  asserts
    \forall a: P[P[X]], s: P[X],x: X
      x \mem (\duni a) == \E s (x \mem s /\ s \mem a);
      x \mem (\dint a) == \A s (s \mem a => x \mem s);
```

```
  implies
    \forall a,b : P[P[X]], s,t : P[X], x: P[X]
      \duni( a \uni b) == ( \duni a) \uni (\duni b);
      \dint ( a \uni b) == (\duni a) \int (\duni b);
      \duni emptyset: P[P[X]] == emptyset;
      (\dint emptyset: P[P[X]]) == universe;
      \A x (func(x)=(x \int s)) => (s \int (\duni a)) = (\duni (map(a: P[P[X]])));
      \A x (func(x)=(x \uni s)) => s \uni (\dint a) = (\dint map(a: P[P[X]]));
      \A x (func(x)=(s \setminus x)) =>
           (\duni a)\setminus s = (\duni map(a: P[P[X]]));
      \A x (func(x)=(x \setminus s)) =>
            s \setminus (\duni a) = (\duni map(a: P[P[X]]));
      (a \psubs b) => (\duni a) \psubs (\duni b);
      (a \psubs b) => (\duni b) \psubs (\duni a);
    converts
      \duni __ : P[P[X]] -> P[X],
      \dint __ : P[P[X]] -> P[X]
```

A.21 Graph.lsl

```
Graph(f, X , Y):trait

   includes
     Function(X, Y)

   assumes
    Func(f, X, Y)

   introduces
     graph: -> P[ pair[ X , Y ]]

   asserts
     \forall x:X, y:Y
       (([x, y]) \mem graph) == f(x) = y;

   implies
     \forall x:X
       apply(graph, x) == f(x);
```

A.22 Graph2.lsl

```
Graph2(f, X , Y):trait

   includes
     Function(X, Y)

   assumes
```

```
 Func2(f, X, Y)

 introduces
   graph: -> P[ pair[ X , Y ]]

 asserts
   \forall x:X, y:Y
     (([x, y]) \mem graph) == f(x, y);

 implies
   \forall x:X, y:Y
     apply(graph, x) = y == f(x, y);
```

A.23 HeadLastTailFront.lsl

```
% @(#)$Id: HeadLastTailFront.lsl,v 1.4 1997/10/06 02: 37: 27 hzhong Exp $
% Written by Hua Zhong with the help of Dr. Gary T. Leavens
% page 117 of "the Z Notation:  A Reference Manual" Second Edition, by Spivey,
%      Prentice Hall,1992

HeadLastTailFront(N,X): trait
includes
  ZNumber, Sequences(N,X),DresRres(N,X)
introduces
  \head __:  P[pair[N,X]]-> X
  \last __:  P[pair[N,X]]-> X
  \tail __:  P[pair[N,X]]-> P[pair[N,X]]
  \front __:  P[pair[N,X]]-> P[pair[N,X]]

asserts
  \forall s: P[pair[N,X]], x: P[pair[N,X]],n: N
    (s \mem (\seqone universe)) => (\head s) = apply(s,1);
    (s \mem (\seqone universe)) => (\last s) = apply(s, (#s));
    (s \mem (\seqone universe)) =>
      (\A n ((n \mem (1 \upto ((#s)-1))) => ([n,apply(s,n)]\mem (\tail s))));
    (s \mem (\seqone universe)) => (\front s) = ( 1 \upto ((# s) -1)) \dres s;
```

A.24 Id.lsl

```
% @(#)$Id: Id.lsl,v 1.4 1997/10/03 02: 27: 32 hzhong Exp $
% Written by Hua Zhong with the help of Dr. Gary T. Leavens
% page 97 of "the Z Notation:  A Reference Manual" Second Edition, by Spivey,
%     Prentice Hall,1992

Id(X) : trait
  includes Rel(X,X)
  introduces
    \id  __ : P[X] ->P[pair[X,X]] % Identily Relation
```

```
  asserts
    \forall x1,x2: X,p: P[X]
      [x1,x2] \mem (\id p) == x1 = x2 /\ (x1 \mem p);

  implies
    \forall x1,x2: X,p: P[X]
      (x1 \map x2) \mem (\id p) == (x1=x2)/\ (x1 \mem p);
```

A.25 Iteration.lsl

```
% @(#)$Id: Iteration.lsl,v 1.4 1997/10/03 03: 54: 31 hzhong Exp $
% Written by Hua Zhong with the help of Dr. Gary T. Leavens
% page 110 of "the Z Notation:  A Reference Manual" Second Edition, by Spivey,
%      Prentice Hall,1992

Iteration(X): trait

  includes Integer, Cmp(X,X,X), Id, RelInv(X,X)
  introduces
    iter: Int, P[pair[X,X]] -> P[pair[X,X]]
    __ \iter __ : P[pair[X,X]], Int -> P[pair[X,X]]

  asserts
   \forall r: P[pair[X,X]], k: Int
     iter(0,r) == \id universe;
     iter(k+1, r) == r \fcmp iter(k, r);
     iter(-k, r)== iter(k,(r \inv));
     r \iter k == iter(k, r);
```

A.26 Pair.lsl

```
% @(#)$Id: Pair.lsl,v 1.3 1997/09/28 03: 22: 44 hzhong Exp $
% Written by Hua Zhong with the help of Dr. Gary T. Leavens
% page 93 of "the Z Notation:  A Reference Manual" Second Edition, by Spivey,
%     Prentice Hall,1992

Pair(X,Y): trait

  pair[X,Y] tuple of x: X, y: Y
  includes
    ZPowerSet(X), ZPowerSet(Y), ZPowerSet(pair[X,Y])

  introduces
    first: pair[X,Y] -> X
    second: pair[X,Y] -> Y
    __ \prod __: P[X],P[Y] ->P[pair[X,Y]]   % cross product
```

```
  asserts
    \forall x: X, y: Y, s: P[X], t: P[Y]
      first([x,y]) == x;
      second([x,y]) == y;
      [x,y] \mem (s \prod t) == x \mem s /\ y \mem t;

  implies
    sort pair[X,Y] generated by [ __ , __ ]
    \forall p: pair[X,Y]
      [first(p),second(p)] == p;
    converts
      first: pair[X,Y] -> X,
      second: pair[X,Y] -> Y,
      __ \prod __ : P[X],P[Y] ->P[pair[X,Y]]
```

A.27 Pass.lsl

```
% @(#)$Id: Pass.lsl,v 1.4 1997/09/28 03: 23: 33 hzhong Exp $
% Written by Hua Zhong with the help of Dr. Gary T. Leavens
%This trait is defined for Filter.lsl

Pass(pass,X): trait

introduces
 pass : X ->Bool
```

A.28 PrefixSuffixIn.lsl

```
% @(#)$Id: PrefixSuffixIn.lsl,v 1.5 1997/10/06 22: 00: 17 hzhong Exp $
% Written by Hua Zhong with the help of Dr. Gary T. Leavens
% page 119 of "the Z Notation:  A Reference Manual" Second Edition, by Spivey,
%      Prentice Hall,1992

PrefixSuffixIn(Z,X): trait
includes
  ZNumber, Function(Z, X), DresRres(Z, X), ConcatRev
introduces
  __ \prefix __:  P[pair[Z,X]], P[pair[Z,X]]-> Bool % Prefix relation
  __ \suffix __:  P[pair[Z,X]], P[pair[Z,X]]-> Bool % Suffix relation
  __ \inseq __:   P[pair[Z,X]], P[pair[Z,X]]-> Bool % Segment relation


asserts
  \forall s,t,u,v: P[pair[Z,X]], x: P[pair[Z,X]]
    s \prefix t == \E v ((s \cat v) = t);
    s \suffix t == \E v ((v \cat s) = t);
    s \inseq t == \E v(\E u (((u \cat s )\cat v)= t));
```

A.29 Rel.lsl

```
% @(#)$Id: Rel.lsl,v 1.5 1997/09/29 01: 34: 47 hzhong Exp $
% Written by Hua Zhong with the help of Dr. Gary T. Leavens
% page 95 of "the Z Notation:  A Reference Manual" Second Edition, by Spivey,
%     Prentice Hall,1992

Rel(X,Y): trait

  includes Pair(X,Y), ZSet(pair[X,Y])
  introduces
    __ \rel __: P[X] , P[Y] -> P[P[pair[X,Y]]]  % relation
    __ \map __: X , Y -> pair[X,Y]              % maplet

  asserts
    \forall s: P[X], t: P[Y], x: X, y: Y
      x \map y == [x,y];
      (s \rel t) == (\pset (s \prod t));

  implies
    \forall s: P[X], t: P[Y], x: X, y: Y, p: P[pair[X,Y]]
      ([x,y] \mem p) => ((p \mem (s \rel t)) => x \mem s /\ y \mem t);

    converts
      __ \rel __: P[X] , P[Y] -> P[P[pair[X,Y]]],
      __ \map __: X , Y -> pair[X,Y]
```

A.30 RelImg.lsl

```
% @(#)$Id: RelImg.lsl,v 1.6 1997/10/03 03: 36: 13 hzhong Exp $
% Written by Hua Zhong with the help of Dr. Gary T. Leavens
% page 101 of "the Z Notation:  A Reference Manual" Second Edition, by Spivey,
%      Prentice Hall,1992

RelImg(X,Y): trait
  includes
    Rel(Y,X), Rel(X,Y)

  introduces
    __ \img __ : P[pair[X,Y]], P[X] -> P[Y]  % relation image

  asserts
    \forall x: X, y: Y,s: P[X],t: P[Y], r: P[pair[X,Y]]
      y \mem (r \img s ) == ([x,y] \mem r) /\ (x \mem s) ;
```

A.31 RelInv.lsl

```
% @(#)$Id: RelInv.lsl,v 1.4 1997/10/03 03: 34: 58 hzhong Exp $
% Written by Hua Zhong with the help of Dr. Gary T. Leavens
```

```
% page 100 of "the Z Notation:  A Reference Manual" Second Edition, by Spivey,
%      Prentice Hall,1992

RelInv(X,Y): trait
  includes
    Rel(Y,X), Rel(X,Y)

  introduces
    __ \inv: P[pair[X,Y]] -> P[pair[Y,X]]  % Relation inverse

  asserts
    \forall x: X, y: Y,s: P[X],t: P[Y], r: P[pair[X,Y]]
      [y,x] \mem (r \inv) == [x,y] \mem r;
```

A.32 RelOvr.lsl

```
% @(#)$Id: RelOvr.lsl,v 1.5 1997/10/03 03: 38: 49 hzhong Exp $
% Written by Hua Zhong with the help of Dr. Gary T. Leavens
% page 102 of "the Z Notation:  A Reference Manual" Second Edition, by Spivey,
%      Prentice Hall,1992

RelOvr(X,Y): trait
  includes
    DomRan(X,Y), DsubRsub(X,Y)

  introduces
    __ \fovr __ : P[pair[X,Y]], P[pair[X,Y]] -> P[pair[X,Y]] % function overide

  asserts
    \forall x: X, y: Y,s: P[X],t: P[Y], r,s: P[pair[X,Y]]
     r \fovr s == ((\dom s) \dsub r)\uni s;
```

A.33 Sequences.lsl

```
% @(#)$Id: Sequences.lsl,v 1.3 1997/10/06 02: 15: 14 hzhong Exp $
% Written by Hua Zhong with the help of Dr. Gary T. Leavens
% page 115 of "the Z Notation:  A Reference Manual" Second Edition, by Spivey,
%      Prentice Hall,1992

Sequences(Z,X): trait
includes
  ZNumber, FiniteSet(pair[Z,X]), FinitePartial(Z,X)

introduces
  \seq __:  P[X] -> P[P[pair[Z,X]]]        %Finite sequences
  \seqone __:  P[X] -> P[P[pair[Z,X]]]     %Non-empty finite sequences
  \iseq __:  P[X] -> P[P[pair[Z,X]]]        %Injective sequences
  \langle \rangle : -> P[P[pair[Z,X]]]
```

```
  \langle __ \rangle :X -> P[pair[Z,X]]
  \langle __ , __ \rangle :X ,X -> P[pair[Z,X]]
  \langle __ , __ , __ \rangle :X ,X ,X -> P[pair[Z,X]]
asserts
  \forall f: P[pair[Z,X]], x: P[X], y, y1, y2, y3:X
    f \mem (\seq x) == (f \mem (nat \ffun x)) /\ ((\dom f) = (1 \upto (#f)));
    f \mem (\seqone x) == f \mem (\seq x) /\ ((# f) >0);
    \iseq x == (\seq x) \int (nat \pinj x);
    \langle \rangle == emptyset:P[P[pair[Z,X]]];
    \langle y \rangle == {[1, y]};
    \langle y1, y2 \rangle == {[1, y1], [2, y2]};
    \langle y1, y2, y3 \rangle == {[1, y1], [2, y2], [3, y3]};
```

A.34 SetMap.lsl

```
% @(#)$Id: SetMap.lsl,v 1.4 1997/09/28 03: 30: 17 hzhong Exp $
% Written by Hua Zhong with the help of Dr. Gary T. Leavens

SetMap(func,X,Y): trait
  includes
    ZPowerSet(X),ZPowerSet(Y)

  assumes
    Func(func,X,Y)

  introduces
    map: P[X] -> P[Y]   % use func to map from one set to another

  asserts
    \forall x: X, y: Y, p: P[X], q: P[Y]
      y \mem map(p) == \E x (x \mem p /\ func(x)=y);

  implies
   \forall x: X, p: P[X]
     x \mem p => func(x) \mem map(p);
     converts map
```

A.35 SetMinMax.lsl

```
% @(#)$Id: SetMinMax.lsl,v 1.2 1997/10/06 02: 03: 05 hzhong Exp $
% Written by Hua Zhong with the help of Dr. Gary T. Leavens
% page 113 of "the Z Notation:  A Reference Manual" Second Edition, by Spivey,
%      Prentice Hall,1992

SetMinMax(Z): trait
includes
  ZNumber, Function(P[Z],Z)
introduces
```

```
  minAsSet: -> P[pair[P[Z],Z]]    % Minimum of a set of numbers
  maxAsSet: -> P[pair[P[Z],Z]]    % Maximum of a set of numbers
  \min __: P[Z] -> Z   % Minimum of a set of numbers
  \max __: P[Z] -> Z    % Maximum of a set of numbers

asserts
  \forall s: P[Z], m,n: Z
    minAsSet \mem ((\psetone integer) \pfun integer);
    maxAsSet \mem ((\psetone integer) \pfun integer);
    ([s,m] \mem minAsSet) == (m \mem s) /\ (\A n ((n \mem s) => (m <=n)));
    \min s == apply(minAsSet, s);
    [s,m] \mem maxAsSet == (m \mem s) /\ (\A n ((n \mem s) => (m >=n)));
    \max s == apply(maxAsSet ,s);
```

A.36 SetUnionIntersectDiff.lsl

```
% @(#)$Id: SetUnionIntersectDiff.lsl,v 1.10 1997/09/28 03: 30: 50 hzhong Exp $
% Written by Hua Zhong with the help of Dr. Gary T. Leavens
% page 91 of "the Z Notation:  A Reference Manual" Second Edition, by Spivey,
%     Prentice Hall,1992

SetUnionIntersectDiff(X): trait
includes ZSet(X)
  introduces
    __ \uni __ : P[X],P[X] -> P[X]         %set union
    __ \int __ : P[X],P[X] -> P[X]         %set intersection
    __ \setminus __  : P[X],P[X] -> P[X]   %set difference

  asserts
    \forall s,t: P[X], x: X
    x \mem (s \uni t) == (x \mem s \/ x \mem t);
    x \mem (s \int t) == (x \mem s /\ x \mem t);
    x \mem (s \setminus t) == (x \mem s /\ ~(x \mem t));

  implies
    Distributive(\uni,\int,P[X]),
    Distributive(\int,\uni,P[X]),
    Semilattice(emptyset for \bot, \int for \glb , \uni for \lub ,
       \subs for <=, \psubs for <, \sups for >=, \psups for >,
       P[X] for T)

    \forall s,t,v: P[X]
      s \uni s == s;
      s \int s == s;
      s \int emptyset == emptyset;
      s \setminus emptyset == s;
      s \setminus s == emptyset;
      emptyset \setminus s == emptyset;
      (s \setminus t) \int t == emptyset;
```

```
      s \int (t \setminus v) == (s \int t) \setminus v;
      s \setminus (t \setminus v) == (s \setminus t) \uni (s \int v);
      (s  \uni t) \setminus v == (s \setminus v) \uni (t \setminus v);
      (s \setminus t) \setminus v == s \setminus (t \uni v);
      s \setminus (t \int v) == (s \setminus t) \uni (s \setminus v);
    converts
        __ \uni __ : P[X],P[X] -> P[X],
__ \int __ : P[X],P[X] -> P[X],
__ \setminus __  : P[X],P[X] -> P[X]
```

A.37 TclRtcl.lsl

```
% @(#)$Id: TclRtcl.lsl,v 1.4 1997/10/03 03: 40: 04 hzhong Exp $
% Written by Hua Zhong with the help of Dr. Gary T. Leavens
% page 103 of "the Z Notation:  A Reference Manual" Second Edition, by Spivey,
%      Prentice Hall,1992

TclRtcl(X): trait
  includes
    Rel(X,X)

  introduces
    __ \tcl : P[pair[X,X]] -> P[pair[X,X]]     % Transitive closure
    __ \rtcl : P[pair[X,X]] -> P[pair[X,X]]   % Reflexive-transitive closure

  asserts
    \forall x,y,z: X, s: P[X], r,s: P[pair[X,X]]
      [x,y] \mem (r \tcl) ==
         ([x,y] \mem r) \/ (\E z ([x,z] \mem r /\ ([z,y] \mem (r \tcl))));
      [x,y] \mem (r \rtcl) ==
         (x=y) \/
         ([x,y] \mem r) \/
         (\E z ([x,z] \mem r /\ ([z,y] \mem (r \tcl))))
```

A.38 ZNumber.lsl

```
% @(#)$Id: ZNumber.lsl,v 1.6 1997/10/07 15: 23: 50 hzhong Exp $
% Written by Hua Zhong with the help of Dr. Gary T. Leavens
% page 108,109 of "the Z Notation:  A Reference Manual" Second Edition, by
%     Spivey, Prentice Hall,1992


ZNumber(Z): trait

  includes Integer(Z), ZPowerSet(Z),Pair(Z,Z)
  introduces
    nat : -> P[Z]             % Natural Numbers
    integer : -> P[Z]         % Integers
```

```
    natone : -> P[Z]          % Strictly positive integers
    \succ __ : Z -> Z     % Successor function
    succAsSet : -> P[pair[Z, Z]]
    __ \upto __ : Z , Z -> P[Z] % Number range

  asserts
    \forall n,a,b: Z
      n \mem integer;
      n >= 0 == n \mem nat;
      n \mem natone == (n \mem nat) /\ n \neq 0;
      \succ n == n+1;
      [n,n+1] \mem succAsSet;
      n \mem ( a \upto b) == (a <= n) /\ (n <= b);
```

A.39 ZPowerSet.lsl

```
% @(#)$Id: ZPowerSet.lsl,v 1.11 1997/10/03 02: 06: 37 hzhong Exp $
% Written by Hua Zhong with the help of Dr. Gary T. Leavens
% This sort is the basic sort for set in Z.
% page 89,90 of "the Z Notation:  A Reference Manual" Second Edition,
%     by Spivey, Prentice Hall,1992

ZPowerSet(X): trait
introduces
  universe: ->P[X]
  __ \mem __ : X , P[X] -> Bool  %membership of a set
  __ \nem __ : X , P[X] -> Bool  %non-membership of a set
  emptyset: -> P[X]
  varemptyset: -> P[X]
  { __ } : X -> P[X]
  { __ , __ } : X , X -> P[X]
  { __ , __ , __ } : X , X , X -> P[X]

asserts
  sort P[X] partitioned by \mem
  \forall x,y,y1,y2,y3: X, p: P[X]
     ~(x \mem emptyset);
     varemptyset == emptyset;
     x \mem  universe;
     x \nem p == ~(x \mem p);
     x \mem { y } == x = y;
     x \mem {y1,y2} == x = y1 \/ x = y2;
     x \mem {y1,y2,y3} == x = y1 \/ x = y2 \/ x = y3

implies
  sort P[X] partitioned by \nem
  \forall x: X, s,t: P[X]
    x \nem emptyset;
    s = t => \A x (x \mem s = x \mem t);
```

```
    \A x (x \mem s = x \mem t) => s = t;
    x \mem s /\ x \nem s == false;
  converts
    __ \nem __ : X, P[X] -> Bool
```

A.40 ZSet.lsl

```
% @(#)$Id: ZSet.lsl,v 1.13 1997/10/03 02: 12: 45 hzhong Exp $
% Written by Hua Zhong with the help of Dr. Gary T. Leavens
% page 56,90 of "the Z Notation:  A Reference Manual" Second Edition,
%     by Spivey, Prentice Hall,1992

ZSet(X): trait
  includes ZPowerSet(X),ZPowerSet(P[X]),
    DerivedOrders(P[X],\subs for <=,\psubs for <,
         \sups for >=,\psups for >)
  introduces
    __ \subs __ : P[X], P[X] -> Bool   %subset relation
    \pset __: P[X] -> P[P[X]]          %powerset of a set
    \psetone __ : P[X] -> P[P[X]]      %non-empty subsets of a set

  asserts
    \forall s,t: P[X], x: X
      s \subs t == (\A x (x \mem s => x \mem t));
      s \mem (\pset t) == s \subs t;
      t \mem (\psetone s) == t \mem (\pset s) /\ ~(t = emptyset);

  implies
    Reflexive(\subs, P[X] for T),
    Irreflexive(\psubs, P[X] for T),
    Antisymmetric(\subs, P[X] for T),
    Asymmetric(\psubs, P[X] for T),
    Transitive(\subs, P[X] for T),
    Transitive(\psubs, P[X] for T)
    \forall s,t,v: P[X], x,y: X
      emptyset \subs s;
      emptyset \psubs s == ~(s =emptyset);
      (\psetone s) = emptyset == s = emptyset;
      ~(s = emptyset) == s \mem (\psetone s);
    converts
      __ \subs __ : P[X], P[X] -> Bool,
      __ \psubs __ : P[X], P[X] -> Bool,
      \pset __: P[X] -> P[P[X]],
      \psetone __ : P[X] -> P[P[X]]
```

Appendix B

Examples for the Larch Prover

In this appendix I present two illustractive proofs using LP. See Chapter 4 for an overview of these proofs.

B.1 ZPowerSet.proof

```
set script ZPowerSet
set log ZPowerSet

%%% Proof Obligations for trait ZPowerSet

execute ZPowerSet_Axioms

%% Implications

declare variables
  x: X
  s: P[X]
  t: P[X]
  ..


% main trait: ZPowerSet

set name ZPowerSetTheorem

prove
  sort P[X] partitioned by \nem
  ..
  resume by =>
    <> => subgoal
```

```
    [] => subgoal
  [] conjecture
qed
prove
  (x \nem emptyset)
  ..
  [] conjecture
prove
  (s = t => \A x (x \mem s = (x \mem t)))
  ..
  [] conjecture
qed
prove
  (\A x (x \mem s = (x \mem t)) => s = t)
  ..
  res by =>
    <> => subgoal
    apply ZPowerSet.1 to conj
    [] => subgoal
  [] conjecture
qed
prove
  (x \mem s /\ x \nem s) = (false)
  ..
  [] conjecture
qed

%% Conversions

freeze ZPowerSet

%% converts __ \nem __: X, P[X] -> Bool

thaw ZPowerSet

declare operators
  __ \nem' __: X, P[X] -> Bool
  ..


% subtrait 0: ZPowerSet (__ \nem' __: X, P[X] -> Bool for __ \nem __: X, P[X]
%   -> Bool)

set name ZPowerSet

assert
  (x \nem' p) = (~ (x \mem p))
  ..

declare variables
```

```
  _x1_: X
  _x2_: P[X]
  ..

set name conversionChecks

prove (_x1_ \nem _x2_) = (_x1_ \nem' _x2_)
  [] conjecture
qed
prove (_x1_ \nem _x2_) = (_x1_ \nem' _x2_)
  [] conjecture
qed
```

B.2 State_Basics.proof

```
set script State_Basics
set log State_Basics

%%% Proof Obligations for trait State_Basics

execute State_Basics_Axioms

%% Implications

declare variables
  obj: Object
  obj1: Object
  st: State
  v: Value
  typs: Set[TYPE]
  v1: Value
  typs1: Set[TYPE]
  ..


% main trait: State_Basics

set name State_BasicsTheorem

prove
  (emptyState ~= bottom) by con
  ..
    <> contradiction subgoal
    [] contradiction subgoal
  [] conjecture
qed
prove
  (isBottom(st)) = (st = bottom)
  ..
```

```
  res by ind
    <> basis subgoal
    [] basis subgoal
    <> basis subgoal
    [] basis subgoal
    <> induction subgoal
    res by case stc = bottom
      <> case stc = bottom
      [] case stc = bottom
      <> case ~(stc = bottom)
      res by con
        <> contradiction subgoal
        crit *Hyp with *
        [] contradiction subgoal
      [] case ~(stc = bottom)
    [] induction subgoal
  [] conjecture
qed

prove
  (~ isBottom(st) => bind(st, obj, v, typs) ~= bottom)
  ..
  [] conjecture
qed

prove
  (~ isBottom(st) => eval(obj1, bind(st, obj1, v, typs)) = v)
  ..
  instantiate obj by obj1 in State_Basics.6
  [] conjecture
qed

prove
  (~ isBottom(st) => eval(obj1, bind(st, obj, v, typs)) = (if obj1 = obj then
    eval(obj1, bind(emptyState, obj1, v, typs)) else eval(obj1, st)))
  ..
  res by =>
    <> => subgoal
    res by case obj1 = obj
      <> case obj1c = objc
      p eval(objc, bind(stc, objc, v, typs))=v
        res by con
          <> contradiction subgoal
          crit *Hyp with *
          [] contradiction subgoal
        [] conjecture
      res by con
        <> contradiction subgoal
        crit *Hyp with *
        [] contradiction subgoal
```

```
      [] case obj1c = objc
      <> case ~(obj1c = objc)
      res by con
        <> contradiction subgoal
        crit *Hyp with *
        [] contradiction subgoal
      [] case ~(obj1c = objc)
    [] => subgoal
  [] conjecture
qed

prove
  (~ isBottom(st) => allocated(obj1, bind(st, obj, v, typs)) = (if obj1 = obj
    then allocated(obj1, bind(emptyState, obj1, v, typs)) else allocated(obj1,
    st)))
  ..
  res by =>
    <> => subgoal
    [] => subgoal
  [] conjecture
qed

prove
  (bind(bind(st, obj, v, typs), obj, v1, typs1)) = (bind(st, obj, v1, typs1))
  ..
  prove
    eval(obj1,bind(bind(st, obj, v, typs), obj, v1, typs1))
       =eval(obj1,bind(st, obj, v1, typs1))
    ..
    declare variables v2: Value typs2: Set[TYPE]
    ins st by bind(st, obj, v2, typs2) in State_Basics.6
    res by case st~=bottom
      <> case stc ~= bottom
      res by case obj1 = obj
        <> case obj1c = objc
        crit * with *Hyp
        [] case obj1c = objc
        <> case ~(obj1c = objc)
        crit * with *Hyp
        [] case ~(obj1c = objc)
      [] case stc ~= bottom
      <> case ~(stc ~= bottom)
qed
      set immu on
      prove bind(bottom, obj, v, typs) = bottom
        [] conjecture
      [] case ~(stc ~= bottom)
    [] conjecture
  set immu off
  prove
```

```
    allocated(obj1,bind(bind(st, obj, v, typs), obj, v1, typs1))
       =allocated(obj1,bind(st, obj, v1, typs1))
    ..
    res by case st=bottom
      <> case stc = bottom
      [] case stc = bottom
      <> case ~(stc = bottom)
      [] case ~(stc = bottom)
    [] conjecture
  apply State_Basics.2 to conjecture
  [] conjecture
qed

prove
  (bind(bind(st, obj, v, typs), obj1, v1, typs1)) = (if obj1 = obj then
    bind(st, obj1, v1, typs1) else bind(bind(st, obj1, v1, typs1), obj, v,
    typs))
  ..
  declare variables obj2: Object v2: Value typs2: Set[TYPE]
  prove allocated(obj2,bind(bind(st, obj, v, typs), obj1, v1, typs1))
   =allocated(obj2,if obj1 = obj then bind(st, obj1, v1, typs1)
     else bind(bind(st, obj1, v1, typs1), obj, v,typs))
  ..
    res by case st = bottom
      res by case obj1 = obj
        <> case obj1c = objc
        [] case obj1c = objc
        <> case ~(obj1c = objc)
        [] case ~(obj1c = objc)
      [] case stc = bottom
      <> case ~(stc = bottom)
      res by case obj1 = obj
        <> case obj1c = objc
        [] case obj1c = objc
        <> case ~(obj1c = objc)
        [] case ~(obj1c = objc)
      [] case ~(stc = bottom)
    [] conjecture
  p eval(obj2,bind(bind(st, obj, v, typs), obj1, v1, typs1))
   =eval(obj2,(if obj1 = obj then bind(st, obj1, v1, typs1)
     else bind(bind(st, obj1, v1, typs1), obj, v, typs)))
  ..
    res by case st=bottom
      <> case stc = bottom
      set immu on
      prove bind(bottom, obj, v, typs)=bottom
        [] conjecture
      [] case stc = bottom
      <> case ~(stc = bottom)
      set immu off
```

```
    res by case obj1 = obj
      <> case obj1c = objc
      [] case obj1c = objc
      <> case ~(obj1c = objc)
      ins st by stc in State_Basics.6
      res by case obj2=obj1c
        <> case obj2c = obj1c
        p eval(obj1c, bind(bind(stc, objc, v, typs), obj1c, v1, typs1))=v1
          res by con
            <> contradiction subgoal
            crit *Hyp with *
            [] contradiction subgoal
          [] conjecture
        ins st by bind(stc, obj1c, v1, typs1) in State_Basics.6
        [] case obj2c = obj1c
        <> case ~(obj2c = obj1c)
        res by case obj2c= objc
          <> case obj2c = objc
          p eval(objc, bind(bind(stc, obj1c, v1, typs1), objc, v, typs))=v
            res by con
              <> contradiction subgoal
              crit *Hyp with *
              [] contradiction subgoal
            [] conjecture
          ins st by bind(stc, objc, v2, typs2)in State_Basics.6
          [] case obj2c = objc
          <> case ~(obj2c = objc)
          p eval(obj2c, bind(bind(stc, objc, v, typs), obj1c, v1, typs1))=
               eval(obj2c, bind(stc, objc, v, typs))
          ..
            ins st by bind(stc, objc, v1, typs1) in State_Basics.6
            [] conjecture
          ins st by bind(stc, obj1c, v1, typs1) in State_Basics.6
          [] case ~(obj2c = objc)
        [] case ~(obj2c = obj1c)
      [] case ~(obj1c = objc)
    [] case ~(stc = bottom)
  [] conjecture
p bind(bind(st, obj, v, typs), obj1, v1, typs1)=bottom
    <=> (if obj1 = obj then bind(st, obj1, v1, typs1) else
         bind(bind(st, obj1, v1, typs1), obj, v, typs))=bottom
..
  res by case obj1 = obj
    <> case obj1c = objc
    [] case obj1c = objc
    <> case ~(obj1c = objc)
    [] case ~(obj1c = objc)
  [] conjecture
apply State_Basics.2 to conjecture
[] conjecture
```

```
qed


%% Conversions

freeze State_Basics

%% converts allocated, eval, isBottom exempting \forall obj: Object eval(obj,
%%   bottom), eval(obj, emptyState)

thaw State_Basics

declare operators
  allocated': Object, State -> Bool
  , isBottom': State -> Bool
  , eval': Object, State -> Value
  ..


% subtrait 0: State_Basics (allocated': Object, State -> Bool for allocated:
%   Object, State -> Bool, eval': Object, State -> Value for eval: Object,
%   State -> Value, isBottom': State -> Bool for isBottom: State -> Bool)

set name State_Basics

assert
  sort State partitioned by allocated', eval', isBottom'
  ;(~ allocated'(obj, emptyState))
  ;(~ allocated'(obj, bottom))
  ;(allocated'(obj, bind(st, obj1, v, typs))) = (~ isBottom'(st) /\ (obj =
    obj1 \/ allocated'(obj, st)))
  ;(~ isBottom'(st) => eval'(obj1, bind(st, obj, v, typs)) = (if obj1 = obj
    then v else eval'(obj1, st)))
  ;(~ isBottom'(emptyState))
  ;(isBottom'(bottom))
  ;(isBottom'(bind(st, obj, v, typs))) = (isBottom'(st))
  ..

declare variables
  obj: Object
  _x1_: Object
  _x1_: State
  _x2_: State
  ..

set name exemptions

assert
  eval(obj, bottom) = eval'(obj, bottom);
  eval(obj, emptyState) = eval'(obj, emptyState)
```

```
  ..

set name conversionChecks
prove
  (isBottom'(st)) = (st = bottom)
  ..
  res by ind
    <> basis subgoal
    [] basis subgoal
    <> basis subgoal
    [] basis subgoal
    <> induction subgoal
    [] induction subgoal
  [] conjecture
qed

prove (allocated(_x1_:Object, _x2_)) = (allocated'(_x1_:Object, _x2_))
  res by ind
    <> basis subgoal
    [] basis subgoal
    <> basis subgoal
    [] basis subgoal
    <> induction subgoal
    [] induction subgoal
  [] conjecture
qed

prove (allocated(_x1_:Object, _x2_)) = (allocated'(_x1_:Object, _x2_))
  [] conjecture
qed

prove (eval(_x1_:Object, _x2_)) = (eval'(_x1_:Object, _x2_))
  res by ind
    <> basis subgoal
    [] basis subgoal
    <> basis subgoal
    [] basis subgoal
    <> induction subgoal
    res by case _x2_c=bottom
      <> case _x2_c = bottom
      set immu on
      p bind(bottom, o, v, s2)=bottom
        [] conjecture
      [] case _x2_c = bottom
      <> case ~(_x2_c = bottom)
      res by case _x1_=o
        <> case _x1_c = oc
        p eval(oc, bind(_x2_c, oc, v, s2))=v
          res by con
            <> contradiction subgoal
```

```
            crit *Hyp with *
            [] contradiction subgoal
          [] conjecture
        res by con
          <> contradiction subgoal
          crit **Hyp with *
          [] contradiction subgoal
        [] case _x1_c = oc
        <> case ~(_x1_c = oc)
        p eval(_x1_c, bind(_x2_c, oc, v, s2))=eval(_x1_c,_x2_c)
          res by con
            <> contradiction subgoal
            crit *Hyp with *
            [] contradiction subgoal
          [] conjecture
        res by con
          <> contradiction subgoal
          crit *Hyp with *
          [] contradiction subgoal
        [] case ~(_x1_c = oc)
      [] case ~(_x2_c = bottom)
    [] induction subgoal
  [] conjecture
qed

prove (eval(_x1_:Object, _x2_)) = (eval'(_x1_:Object, _x2_))
  [] conjecture
qed

prove (isBottom(_x1_:State)) = (isBottom'(_x1_:State))
  [] conjecture
qed
```

BIBLIOGRAPHY

[1] Jonathan Bowen. Z Frequently Asked Questions. Available in ftp://ftp.comlab.ox.ac.uk/archive/z.html, September 1997.

[2] Stephen J. Garland and John V. Guttag. A guide to lp, the larch prover. Technical Report 82, Digital Equipment Corporation, Systems Research Center, 130 Lytton Avenue, Palo Alto, CA 94301, December 1991. Order from src-report@src.dec.com.

[3] John V. Guttag and James J. Horning, editors. *Larch: Languages and Tools for Formal Specification.* Texts and Monographs in Computer Science. Springer-Verlag, 1993. With Stephen J. Garland, Kevin D. Jones, Andrés Modet, and Jeannette M. Wing.

[4] I. Hayes, editor. *Specification Case Studies.* International Series in Computer Science. Prentice Hall, 1987.

[5] Daniel Jackson. Structuring Z specifications with views. *ACM Transactions on Software Engineering and Methodology*, 4(4):365–389, October 1995.

[6] Leslie Lamport and Lawrence C. Paulson. Should your specification language be typed? Technical report, Digital Equipment Corporation, Systems Research Center, 130 Lytton Avenue, Palo Alto, CA 94301, May 1997.

[7] Gary T. Leavens. Larch Frequently Asked Questions. Version 1.88. Available in http://www.cs.iastate.edu/~leavens/larch-faq.html, October 1997.

[8] Gary T. Leavens and Jeannette M. Wing. Protective interface specifications. In Michel Bidoit and Max Dauchet, editors, *TAPSOFT '97: Theory and Practice of Software Development, 7th International Joint Conference CAAP/FASE, Lille, France*, volume 1214 of *Lecture Notes in Computer Science*, pages 520–534. Springer-Verlag, New York, NY, 1997.

[9] J. Michael Spivey. *The Z Notation: A Reference Manual.* International Series in Computer Science. Prentice-Hall, New York, NY, second edition, 1992.