12-1997

LSL Traits for using Z with Larch

Hua Zhong

Iowa State University

Follow this and additional works at: http://lib.dr.iastate.edu/cs_techreports

Part of the Programming Languages and Compilers Commons

Recommended Citation

http://lib.dr.iastate.edu/cs_techreports/155

This Article is brought to you for free and open access by the Computer Science at Digital Repository @ Iowa State University. It has been accepted for inclusion in Computer Science Technical Reports by an authorized administrator of Digital Repository @ Iowa State University. For more information, please contact digirep@iastate.edu.
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,


Partions will be Submitted
Copyright © 1997 by Hua Zhong

Department of Computer Science
Iowa State University
Ames, Iowa 50011, USA
A.6 CmpLaws.lsl ................................. 40
A.7 ConcatRev.lsl .............................. 41
A.8 DConcat.lsl ................................ 41
A.9 DisjointPartitions.lsl .................... 41
A.10 DomRan.lsl .............................. 42
A.11 DresRres.lsl ............................ 43
A.12 DsubRsub.lsl ............................ 44
A.13 ExtFilCom.lsl ............................ 44
A.14 Filter.lsl ................................ 44
A.15 FinitePartial.lsl ....................... 45
A.16 FiniteSet.lsl ............................ 45
A.17 Func.lsl ............................... 46
A.18 Func2.lsl .............................. 46
A.19 Function.lsl ............................ 46
A.20 GeneUniInt.lsl ........................... 47
A.21 Graph.lsl ................................ 48
A.22 Graph2.lsl .............................. 48
A.23 HeadLastTailFront.lsl ................. 49
A.24 Id.lsl ................................... 49
A.25 Iteration.lsl ........................... 50
A.26 Pair.lsl ................................ 50
A.27 Pass.lsl ................................ 51
A.28 PrefixSuffix.lsl ....................... 51
A.29 Rel.lsl ................................ 52
A.30 RelImg.lsl .............................. 52
A.31 RelInv.lsl .............................. 52
A.32 RelOvr.lsl .............................. 53
A.33 Sequences.lsl ........................... 53
A.34 SetMap.lsl .............................. 54
A.35 SetMinMax.lsl ........................... 54
A.36 SetUnionIntersectDiff.lsl .......... 55
A.37 TclRtc.lsl .............................. 56
A.38 ZNumber.lsl ............................ 56
A.39 ZPowerSet.lsl ........................... 57
A.40 ZSet.lsl ................................ 58

B Examples for the Larch Prover 59
B.1 ZPowerSet.proof ......................... 59
B.2 State_Basics.proof ...................... 61

BIBLIOGRAPHY 69
List of Tables

2.1 Sets ......................................................... 10
2.2 Relations .................................................. 11
2.3 Functions .................................................. 11
2.4 Misc ......................................................... 11
2.5 Numbers and finiteness ................................. 12
2.6 Sequences .................................................. 12
2.7 Bags ......................................................... 13
List of Figures

1.1 A small example for Z ........................................ 2
1.2 LSL Part .................................................. 4
1.3 Larch/C++ Part ........................................... 5
2.1 ZSet.lsl ................................................... 8
3.1 Z specification for Example 2 .............................. 19
4.1 The facts gained by LP .................................... 31
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 \(= stk \in \text{seq Object}
\)

\text{stkINIT} = \langle \rangle

\[
\begin{align*}
\text{Push} \\
\Delta \text{Stack} \\
o? : \text{Object} \\
stk' = stk \cap \langle o? \rangle
\end{align*}
\]

\[
\begin{align*}
\text{Pop} \\
\Delta \text{Stack} \\
o! : \text{Object} \\
stk' = \text{front} stk \land \\
o! = \text{tail} stk
\end{align*}
\]

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 \(=\) shows that the definition of Stack is syntactically equivalent to \(stk \in \text{seq Object}\).

\text{stkINIT} = \langle \rangle shows the initial state of stk is a empty sequence. \(\langle \rangle\) represents a empty sequence, and notation INIT is used to define the initial state of a object.

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

\[
\begin{align*}
\text{Push} \\
\Delta \text{Stack} \\
o? : \text{Object} \\
stk' = stk \cap \langle o? \rangle
\end{align*}
\]

The part above the center line is the declarations. \(\Delta \text{Stack}\) 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 \(o? : \text{Object}\) 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 ⊃ ⟨o?⟩ shows the stk after is the stk before concatenate the sequence ⟨o?⟩.

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 post-conditions (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 predicates 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 an 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^\ast) \result=top(stk^\ast); shows
that after the call the function should throw no exception, stk should get the
value pop(stk^\ast) and this function returns the value top(stk^\ast). (stk^\ast 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
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 \texttt{Push} and \texttt{Pop} are virtual functions, and by using the notation \texttt{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 \( \text{EM}_X \) 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

Sets 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.
ZSet(X): trait
    includes ZPowerSet(X), ZPowerSet(P[X]),
    DerivedOrders(P[X], \sub for <=, \psubs for <,
    \sup for >=, \psups for >)
introduces
    _- \sub _- : 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 \sub t == (\forall x (x \mem s => x \mem t));
    s \mem (\pset t) == s \sub t;
    t \mem (\psetone s) == t \mem (\pset s) \& \& (t \neq \emptyset);

implies
    Reflexive(\sub s, P[X] for T),
    Irreflexive(\psubs s, P[X] for T),
    Antisymmetric(\sub s, P[X] for T),
    Asymmetric(\psubs s, P[X] for T),
    Transitive(\sub s, P[X] for T),
    Transitive(\psubs s, P[X] for T)
\forall s, t, v: P[X], x, y: X
    \emptyset \sub s;
    \emptyset \psubs s == "(s = \emptyset);
    (\psetone s) = \emptyset \psubs s == s = \emptyset;
    "(s = \emptyset) == s \mem (\psetone s);
converts
    _- \sub _- : P[X], P[X] -> Bool,
    _- \psubs _- : P[X], P[X] -> Bool,
    \pset _-_ : P[X] -> P[P[X]],
    \psetone _-_ : P[X] -> P[P[X]]
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 \textit{includes} section is used to illustrate that from which traits this trait inherits the properties. For example, the trait \textit{ZPowerSet} defines the empty set and universe for a sort \(X\). In \textit{ZSet}, we include \textit{ZPowerSet}(\(X\)) and \textit{ZPowerSet}(\(P[X]\)). In this way we have defined the empty set and universe for sorts \(X\) and \(P[X]\). (Of course, \textit{ZPowerSet} also defines some other operations.)

The \textit{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 \textit{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 \textit{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,

\[
\begin{array}{c}
\forall S, T : \mathbb{P} X \cdot \\
(S \subseteq T \Leftrightarrow (\forall x : X \cdot x \in S \Rightarrow x \in T)) \land \\
(s \subseteq T \Rightarrow S \subseteq T \land s \neq T)
\end{array}
\]

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

\(\mathbb{P} (\\setminus\text{pset})\) 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 \(\\setminus\text{subs}\).

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

\[
\mathbb{P}_1 X \equiv \{S : \mathbb{P} X \mid S \neq \emptyset\}
\]

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

The last part is the \textit{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 \textit{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>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>ZPowerSet</td>
<td>89,90</td>
<td>\mem \nem \nemptyset, vemptyset, { _, _ } { a }, { _, _, _ } { a, b, c }</td>
<td>∈ \notin \emptyset</td>
</tr>
<tr>
<td>ZSet</td>
<td>90</td>
<td>\subs \pset \psetone \setminus</td>
<td>⊆</td>
</tr>
<tr>
<td>SetUnionIntersectDiff</td>
<td>91</td>
<td>\int \uni \setminus</td>
<td>∪ ∩</td>
</tr>
<tr>
<td>GeneUniInt</td>
<td>92</td>
<td>\duni \rint</td>
<td>∪ ∩</td>
</tr>
</tbody>
</table>
### Table 2.2: Relations

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Rel</td>
<td>95</td>
<td>\rel</td>
<td>$\leftrightarrow$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>\map</td>
<td>$\mapsto$</td>
</tr>
<tr>
<td>DomRan</td>
<td>96</td>
<td>\dom</td>
<td>dom</td>
</tr>
<tr>
<td></td>
<td></td>
<td>\ran</td>
<td>ran</td>
</tr>
<tr>
<td>Id</td>
<td>97</td>
<td>\id</td>
<td>id</td>
</tr>
<tr>
<td>Cmp</td>
<td>97</td>
<td>\fcmp</td>
<td>$;$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>\cmp</td>
<td>$\circ$</td>
</tr>
<tr>
<td>DresRres</td>
<td>98</td>
<td>\dres</td>
<td>$\triangleleft$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>\rres</td>
<td>$\triangleright$</td>
</tr>
<tr>
<td>DsubRsub</td>
<td>99</td>
<td>\dsub</td>
<td>$\leftarrow$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>\rsub</td>
<td>$\leftrightarrow$</td>
</tr>
<tr>
<td>RelInv</td>
<td>100</td>
<td>\inv</td>
<td>$R^\sim$</td>
</tr>
<tr>
<td>RelImg</td>
<td>101</td>
<td>\img</td>
<td>$||$</td>
</tr>
<tr>
<td>RelOvr</td>
<td>102</td>
<td>\fovr</td>
<td>$\odot$</td>
</tr>
<tr>
<td>TclRtcl</td>
<td>103</td>
<td>\tcl</td>
<td>$+$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>\rtcl</td>
<td>$\ast$</td>
</tr>
</tbody>
</table>

### Table 2.3: Functions

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Function</td>
<td>105</td>
<td>\pfun</td>
<td>$\rightarrow$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>\tfun</td>
<td>$\rightarrow$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>\pinj</td>
<td>$\mapsto$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>\tinj</td>
<td>$\mapsto$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>\psur</td>
<td>$\mapsto$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>\tsur</td>
<td>$\mapsto$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>\bij</td>
<td>$\mapsto$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>apply</td>
<td>$f(x)$</td>
</tr>
</tbody>
</table>

### Table 2.4: Misc

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Func</td>
<td></td>
<td>func</td>
<td></td>
</tr>
<tr>
<td>Pass</td>
<td></td>
<td>pass</td>
<td></td>
</tr>
<tr>
<td>SetMap</td>
<td></td>
<td>map</td>
<td></td>
</tr>
<tr>
<td>Pair</td>
<td>93</td>
<td>first</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>second</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>\prod</td>
<td>$\times$</td>
</tr>
</tbody>
</table>
Table 2.5: Numbers and finiteness

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>ZNumber</td>
<td>108</td>
<td>nat $\mathbb{N}$</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>integer $\mathbb{Z}$</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>natone $\mathbb{N}_1$</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>\texttt{\textbackslash succ} $\texttt{succ}$</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>\texttt{\textbackslash upto} $\texttt{n..m}$</td>
<td></td>
</tr>
<tr>
<td>Iteration</td>
<td>110</td>
<td>iter(0,R) $\texttt{iter 0 R}$</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>\texttt{\textbackslash iter} $\texttt{R}^k$</td>
<td></td>
</tr>
<tr>
<td>FiniteSet</td>
<td>111</td>
<td>\texttt{\textbackslash set} $\texttt{F}$</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>\texttt{\textbackslash setone} $\texttt{F}_1$</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>\texttt{#} $#$</td>
<td></td>
</tr>
<tr>
<td>FinitePartial</td>
<td>112</td>
<td>\texttt{\textbackslash ffun} $\Rightarrow$</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>\texttt{\textbackslash finj} $\Rightarrow^+$</td>
<td></td>
</tr>
<tr>
<td>SetMinMax</td>
<td>113</td>
<td>\texttt{\textbackslash min} $\texttt{min}$</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>\texttt{\textbackslash max} $\texttt{max}$</td>
<td></td>
</tr>
</tbody>
</table>

Table 2.6: Sequences

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequences</td>
<td>115</td>
<td>\texttt{\textbackslash seq} $\texttt{seq}$</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>\texttt{\textbackslash seqone} $\texttt{seq}_1$</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>\texttt{\textbackslash iseq} $\texttt{iseq}$</td>
<td></td>
</tr>
<tr>
<td>ConcatRev</td>
<td>116</td>
<td>\texttt{\textbackslash cat} $\sim$</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>\texttt{\textbackslash rev} $\texttt{rev}$</td>
<td></td>
</tr>
<tr>
<td>HeadLastTailFront</td>
<td>117</td>
<td>\texttt{\textbackslash head} $\texttt{head}$</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>\texttt{\textbackslash last} $\texttt{last}$</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>\texttt{\textbackslash tail} $\texttt{tail}$</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>\texttt{\textbackslash front} $\texttt{front}$</td>
<td></td>
</tr>
<tr>
<td>ExtFilCom</td>
<td>118</td>
<td>\texttt{\textbackslashires} $\mid$</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>\texttt{\textbackslash sires} $\mid$</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>\texttt{\textbackslash squash} $\texttt{squash}$</td>
<td></td>
</tr>
<tr>
<td>PrefixSuffixIn</td>
<td>119</td>
<td>\texttt{\textbackslash prefix} $\subset$</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>\texttt{\textbackslash suffix} $\texttt{suffix}$</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>\texttt{\textbackslash inseq} $\texttt{in}$</td>
<td></td>
</tr>
<tr>
<td>DConcat</td>
<td>121</td>
<td>\texttt{\textbackslash dcat} $\sim^/$</td>
<td></td>
</tr>
</tbody>
</table>
Table 2.7: Bags

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Bags</td>
<td>124</td>
<td>\text{bag}</td>
<td>bag</td>
</tr>
<tr>
<td></td>
<td></td>
<td>\text{\textbackslash bagcount}</td>
<td>count</td>
</tr>
<tr>
<td></td>
<td></td>
<td>#</td>
<td>\sharp</td>
</tr>
<tr>
<td></td>
<td></td>
<td>\text{\textbackslash otimes}</td>
<td>\odot</td>
</tr>
<tr>
<td>BagMemSub</td>
<td>125</td>
<td>\text{\textbackslash inbag}</td>
<td>in</td>
</tr>
<tr>
<td></td>
<td></td>
<td>\text{\textbackslash bagsubs}</td>
<td>\subseteq</td>
</tr>
<tr>
<td>BagUniDiff</td>
<td>126</td>
<td>\text{\textbackslash buni}</td>
<td>\sqcup</td>
</tr>
<tr>
<td></td>
<td></td>
<td>\text{\textbackslash bdiff}</td>
<td></td>
</tr>
<tr>
<td>BagItems</td>
<td>127</td>
<td>\text{\textbackslash items}</td>
<td>items</td>
</tr>
<tr>
<td>DisjointPartitions</td>
<td>122</td>
<td>\text{\textbackslash disjoint}</td>
<td>disjoint</td>
</tr>
<tr>
<td></td>
<td></td>
<td>\text{\textbackslash partitions}</td>
<td>partitions</td>
</tr>
<tr>
<td>Filter</td>
<td></td>
<td></td>
<td>filter</td>
</tr>
</tbody>
</table>
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 on the set theory, the notation for power set is widely used in Z specifications. The file predefined.h contains the following lines:

```c
//@ 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.

```c
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, \nmem, 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 SetOf-
  Sym 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 \to 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:

<table>
<thead>
<tr>
<th>Update</th>
</tr>
</thead>
<tbody>
<tr>
<td>st, st' : SYM \to VAL</td>
</tr>
<tr>
<td>s? : SYM</td>
</tr>
<tr>
<td>v? : VAL</td>
</tr>
<tr>
<td>st' = st \oplus {s? \map v?}</td>
</tr>
</tbody>
</table>

  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 (⊔) of the state st^ before the update operation by the pair s → v. The ⊔ operation is defined in the trait RelOvr.lsl (Appendix A.32). The definition is that when b(x) is defined, then (a ⊔ b)(x) = b(x), otherwise (a ⊔ b)(x) has the value of a(x), if any.

The notation \map is the notation of LaTeX that corresponds to the Z symbol ⊔; and similarly \fovr is used for ⊔. 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:

\begin{verbatim}
LookUp
st, st' : SYM → VAL
s? : SYM
v? : VAL

s? ∈ dom st ∧
v! = st(s?) ∧
st' = st
\end{verbatim}

In Larch/C++, this becomes,

```c++
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? ∈ 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

st, st' : SYM -> VAL
s? : SYM

s? \in dom st \land
st' = \{ s? \} \leftrightarrow st
```

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 (\leftrightarrow) is defined like this: "An object `x` is related to an object `y` by the relation `S \leftrightarrow 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:

```c++
#include "predefined.h"
//@ symbol table
#include <predefined.h>
//@ template <class SYM, class VAL>
//@ expects contained_objects(SYM), contained_objects(VAL);
//@ where SYM is {
//@    bool operator == (SYM x, SYM y);
//@ }
//@ invariant st\any \mem (universe:SetOfSym \pfun universe:SetOfVal);
//@ virtual Update(SYM s, VAL v) throw();
//@ behavior {
//@    modifies st;
```
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 ∆, one usually means that this object may be modified. And when one uses ≡, one tells this object will not be changed. So very often, the schema with symbol ∆ is translated to a function with modifies clause of this object. And the schema with symbol ≡ illustrates that object should not be put into the modifies clause of Larch/C++.

In fact, ∆ and ≡ 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 \( X \lor Y \), 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:

\[
X \equiv A \land B \\
Y \equiv \neg A \land C
\]

The disjunction (\( \lor \)) 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, \( Z \equiv X \lor Y \) is equal to definition \( Z \equiv (A \land B) \lor \neg A \land C \).

In Larch/C++, the definition is

```cpp
void Z()
{
  requires A;
  modifies ...;
```
\[ ST \triangleq [st : SYM \rightarrow VAL] \]
\[ st\text{INIT} \triangleq \{ \} \]

\[ \text{LookUp} \]
\[ \equiv ST \]
\[ s? : SYM \]
\[ v? : VAL \]
\[ s? \in \text{dom} st \land v! = st(s?) \]

\[ \Delta ST \]
\[ s? : SYM \]
\[ s? \in \text{dom} st \land st' = \{ s? \} \uplus st \]

\[ \text{Update} \]
\[ \Delta ST \]
\[ s? : SYM \]
\[ v? : VAL \]
\[ st' = st \uplus \{ s? \mapsto v? \} \]

\[ \text{NotPresent!} \]
\[ \equiv ST \]
\[ s? : SYM \]
\[ \text{rep!} : \text{Report} \]
\[ s \notin \text{dom} st \land \text{rep! = "Symbol not present"} \]

\[ \text{Success} \triangleq [\text{rep!} : \text{Report} \mid \text{rep!} = \text{"OK"}] \]
\[ ST\text{LookUp} \triangleq (\text{LookUp} \land \text{Success}) \lor \text{NotPresent!} \]
\[ ST\text{Update} \triangleq (\text{Update} \land \text{Success}) \]
\[ ST\text{Delete} \triangleq (\text{LookUp} \land \text{Success}) \lor \text{NotPresent!} \]

Figure 3.1: Z specification for Example 2
This definition is equal to the following definition:

```c
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 (\) of two schemas means merging their declarations to-
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 \ 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:

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

20
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<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:

```c++
#include "predefined.h"
// symbol table with error handling
enum RepSignal{OK,Symbol_not_present};
template <class SYM, class VAL>
//0 expects contained_objects(SYM), contained_objects(VAL);
//0 where SYM is {
//0   bool operator == (SYM x, SYM y);
//0   behavior {
//0     ensures returns \ result = (x == y);
//0   }
//0 };
/*@ 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:
//0 uses Function(SYM,VAL);
//0 spec typedef P<SYM> SetOfSym;
//0 spec typedef P<VAL> SetOfVal;
//0 spec P<<pair<SYM,VAL> >> st;
//0 enum RepSignal rep;
//0 invariant st\any \mem (universe:SetOfSym \pfun universe:SetOfVal);

virtual VAL STLookUp(SYM s) throw(RepSignal);
//0 behavior {
//0   requires s \mem \{\dom st^\};
//0   ensures returns \ result = apply(st^\,s);
//0   also
//0   requires s \mem \{\dom st^\};
//0   ensures throws(RepSignal) \_
//0   thrown(RepSignal)== Symbol_not_present;
//0 }

virtual STUpdate(SYM s,VAL v) throw();
//0 behavior {
//0  modifies st, rep;
//0  ensures st' = (st^\forallv \{s \map v\});
//0 }

virtual void STDelete(SYM s) throw(RepSignal);
//0 behavior {
//0  requires s \mem \{\dom st^\};
//0  modifies st, rep;
//0  ensures st' = {s} \dsub st^\ /\ returns;
//0  also
```
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:

\[
\begin{align*}
[Id] \\
\text{Status} &::= \text{ringing} | \text{idle} | \text{waiting} | \text{connected} | \text{dialtone} | \text{busytone} | \text{ringtone} \\
\text{Phone} &\triangleq [ps : Id \rightarrow \text{Status}] \\
\text{Phone}_{-Dial} &\triangleq [\Delta \text{Phones}; \ i : Id | ps(i) = \text{dialtone} \land ps(i) = \text{waiting}] \\
\end{align*}
\]

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

I have not put the \(\Delta\text{Phone}\) 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 \(\text{Phone}_{-Dial}\_pre\) is defined for the use of Larch/C++, it is the precondition of operation \(\text{Phone}_{-Dial}\).

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

\[
\text{Phoneview}(Id) : \text{trait}
\]

\[
\text{includes}
\]

\[
\text{Function}(Id, \text{STATUS})
\]
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]);
Phone.Dial(ps,ps') = Phone(ps) \setminus Phone(ps');
Phone.Dial.pre(ps,ps',i) == (apply(ps,i) = dialtone) \setminus
  (apply(ps',i) = waiting);
Phones.Frame(ps, ps', c) ==
  \forall 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:

\[\begin{array}{l}
  \text{Switch} \\
  \text{reqconn} : Id \rightarrow Id \\
  \text{conn} : Id \rightarrow Id \\
  \text{conn} \subseteq \text{reqconn} \\
  \dom \text{conn} \cap \ran \text{conn} = \emptyset \\
\end{array}\]

\[\begin{array}{l}
  \text{Switch.Request} \\
  \Delta \text{Switch} \\
  \text{from}, \text{to} : Id \\
  \text{reqconn}' = \text{reqconn} \cup \{\text{from} \mapsto \text{to}\} \\
  \text{conn}' = \text{conn} \\
\end{array}\]

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.

24
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:

\[
\begin{align*}
Net & \triangleq \text{Phones} \land \text{Switch} \\
Net_{\text{Req}} & \triangleq \\
& \quad \Delta Net \land (\exists c : \text{Id} \mid c = \{\text{from}\} \bullet \text{Phone_Frame}) \\
& \quad \land \text{Switch_Request} \land \text{Phone_Dial} [\text{from}/i]
\end{align*}
\]

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);
abstract **class phone_net {**
  uses Phoneview(Id);
  uses switchview(Id);
  spec P<Id,Id> > conns;
  spec P<Id,Id> > reqconns;
  spec P<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);
  }

  // ...
- Put the schema’s predicate into the \texttt{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 \texttt{requires} clause.

- Find out all the objects that can be changed by the function and list them in the \texttt{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 \texttt{modifies} clause must have already put restriction on that.

  - Use \texttt{also} if possible, to break the specification into cases.

  - Move the common predicates into \texttt{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 evolutary 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].


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 \rightarrow false
ZPowerSet.3: varemptyset \rightarrow emptyset
ZPowerSet.4: x \ mem universe \rightarrow true
ZPowerSet.5: x \ mem p \rightarrow ~(x \ mem p)
ZPowerSet.6: x \ mem \{y\} \rightarrow x = y
ZPowerSet.7: x \ mem \{y1, y2\} \rightarrow x = y1 \lor x = y2
ZPowerSet.8: x \ mem \{y1, y2, y3\} \rightarrow x = y1 \lor x = y2 \lor x = y3

And then when proving the theorem:

sort P[X] partitioned by \ mem

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

Conjecture ZPowerSetTheorem.1: when \ A x (x \ mem p <=> x \ mem p1) yields p = p1, try a proof of deduction rule.
Level 2 subgoal for proof of deduction rule:
\ A x (x \ mem p <=> x \ mem 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 built-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:
First by observing this conjecture, we can see that it contains a free variable \( st \). And using the LP command \texttt{display}, we can find out there is an induction rule for its sort. So I tried the command \texttt{resume by induction}. Other commands might cause a free variable to be converted into a constant (e.g., \texttt{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:

\[
(stc = \text{bottom}) = (\text{bind}(stc, o, v, s2) = \text{bottom})
\]

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 \texttt{resume by case stc =bottom}.

LP proved the case \( stc = \text{bottom} \) automatically, and generated the other case \( \neg (stc = \text{bottom}) \). At this step, the current subgoal becomes:

\[
\neg (\text{bind}(stc, o, v, s2) = \text{bottom}).
\]

It can be simply proved by trying the command \texttt{resume by contradiction} and the command \texttt{critical-pairs *Hyp with *}. These two steps are usually used together. Because when we use \texttt{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:

\[
(\text{bind}(\text{bind}(st, obj, v, typs), obj, v1, typs1)) = (\text{bind}(st, obj, v1, typs1))
\]

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 \text{State}, so what we are proving is that one \text{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 \texttt{State_Basics2}. When using the deduction rule, we need to use the \texttt{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

\[
\forall \text{obj} \ (\text{allocated(obj, st1)} <=> \text{allocated(obj, st2)})
\]

and

\[
\forall \text{obj} \ (\text{eval(obj, st1)} = \text{eval(obj, st2)})
\]

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:

\[
\text{prove}
\]
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, types), obj, v1, types1))
  = eval(obj1, bind(st, obj, v1, types1))
  ...

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
  \( \text{eval}(\text{obj1}, \text{bind}(\text{st}, \text{obj}, \text{v}, \text{types})) \)
is equal to
  \( \text{eval}(\text{obj1}, \text{st}) \).
So, I rewrote it to the form I want by using the command `instantiate st` by `bind(st, obj, v2, types2)` in State_Basics.6. The rule State_Basics.6 has been transformed to the following:

\[
(\text{st} = \text{bottom}) \Rightarrow \text{eval}(\text{obj1}, \text{bind}(\text{st}, \text{obj}, \text{v2}, \text{types2}), \text{obj}, \text{v}, \text{types})
  = (\text{if } \text{obj1} = \text{obj} \text{ then } \text{v} \text{ else } \text{eval}(\text{obj1}, \text{bind}(\text{st}, \text{obj}, \text{v2}, \text{types2})))
\]

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 \( st = \text{bottom} \) has been successfully proved.

For the case \( st = \text{bottom} \), the current subgoal is:

Current subgoal:
  \( \text{eval}(\text{obj1}, \text{bind}(\text{bottom}, \text{obj}, \text{v}, \text{types}), \text{obj}, \text{v1}, \text{types1}))
  = \text{eval}(\text{obj1}, \text{bind}(\text{bottom}, \text{obj}, \text{v1}, \text{types1})) \)

Obviously, if I had a lemma: `bind(bottom, obj, v, type)= 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.

\( \text{bind}(\text{bottom}, \text{obj}, \text{v}, \text{types})= \text{bottom} \)

Then I turned immunization off next. At this time, the case \( st = \text{bottom} \) is also proved.

Then, I executed the command:

prove
  \( \text{allocated}(\text{obj1}, \text{bind}(\text{st}, \text{obj}, \text{v}, \text{types}), \text{obj}, \text{v1}, \text{types1}))
  = \text{allocated}(\text{obj1}, \text{bind}(\text{st}, \text{obj}, \text{v1}, \text{types1})) \)
  ...

I separated it to two cases: \( st = \text{bottom} \) and \( st~\text{=bottom} \). 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 who does the proving work. LP can only finish proving a conjecture 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 cannot 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 $xc$ cannot. 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: \( f(\text{true}) = g(\text{true}) \), and \( f(\text{false}) = g(\text{false}) \). These two subgoals 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 \( \text{true} = \text{false} \). 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 ∈ 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 \texttt{ZPowerSet}, there is a constant \textit{universe}. The \textit{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 \( \circ \) is defined as following:

\[
\begin{array}{c}
X, Y, Z \\
\forall Q : X \leftrightarrow Y; R : Y \leftrightarrow Z. \\
\end{array}
\]

\[
\begin{array}{c}
\bigcup \{ x : X; y : Y; z : Z | x Q y \land y R z \land x \leftrightarrow z \} \\
\end{array}
\]

Following this, in Spivey’s book, there is a law id \( X \Downarrow P = P \), and 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 \( f : A \to C \), 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, \( \min : \mathcal{P}X \to X \) 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 induces Graph(\( g \)) in his traits and graph is the set of all pairs of the form \( (x,g(x)) \), where \( x \) is an 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 \( \emptyset \). 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 operation’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:

```c
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 mathematical 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

```
% 0(#)@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, DresRes(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 \rrres \{x\})) ;
```

A.2 BagMemSub.lsl

```
% 0(#)@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]] \rightarrow Bool \quad \% Bag Membership
-- \bagsubs__ : P[pair[X,Z]], P[pair[X,Z]] \rightarrow Bool \quad \% Sub-bag relation

asserts

\forall x : X, b, c : P[pair[X,Z]]
\quad x \inbag b \iff x \in \text{dom} (\text{dom} b);
\quad \text{\bagsubs} c \iff \forall x ((b \not\in c) \iff (c \not\in 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] \rightarrow P[P[pair[X, Z]]] \quad \% Bags
\\bagcount__ : P[pair[X,Z]]\rightarrow P[pair[X,Z]] \quad \% Multiplicity
-- \#__ : P[pair[X,Z]], X \rightarrow Z
-- \otimes__ : Z,P[pair[X,Z]]\rightarrow P[pair[X,Z]] \quad \% bag scaling

asserts

\forall x : X, p : P[X], n : Z, b : P[pair[X,Z]]
\quad \\bag p \ equalTo p \\pfun \text{natone};
\quad \\bagcount b \equalTo (\text{universe \prod \{0\}}) \\forall r b;
\quad b \# x \equalTo \text{aply}(\\\bagcount b, x);
\quad (n \\otimes b) \# x \equalTo 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]] \rightarrow P[pair[X,Z]] \quad \% bag union
-- \bdiff__ : P[pair[X,Z]], P[pair[X,Z]] \rightarrow P[pair[X,Z]] \quad \% bag difference

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

### 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
  _ _ \ \text{cmp} _ _ : P[\text{pair}[X,Y]] , P[\text{pair}[Y,Z]] -> P[\text{pair}[X,Z]]
% Backward relational composition
  _ _ \ \text{cmp} _ _ : P[\text{pair}[Y,Z]] , P[\text{pair}[X,Y]] -> P[\text{pair}[X,Z]]

asserts

\[ \forall x : X, \ y : Y, \ z : Z, \ p : P[X], \]
\[ r : P[\text{pair}[X,Y]], \ q : P[\text{pair}[Y,Z]] \]
\[ [x,z] \ \text{mem} (r \ \text{cmp} q) == \ \exists \ y ([x,y] \ \text{mem} r \ \text{and} \ [y,z] \ \text{mem} q); \]
\[ [x,z] \ \text{mem} (q \ \text{cmp} r) == \ \exists \ y ([x,y] \ \text{mem} r \ \text{and} \ [y,z] \ \text{mem} q); \]

implies

\[ \forall r : P[\text{pair}[X,Y]], \ q : P[\text{pair}[Y,Z]] \]
\[ r \ \text{fcmp} q == q \ \text{cmp} r; \]

### 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

\[ \forall x : X, \ y : Y, \ z : Z, \ p,p1,p2 : P[X],\ t : P[Y], \]
\[ r : P[\text{pair}[X,Y]], \ q : P[\text{pair}[Y,Z]],s : P[\text{pair}[Z,W]] \]
\[ (r \ \text{fcmp} q) \ \text{fcmp} s == r \ \text{fcmp} (q \ \text{fcmp} s); \]
\[ (\ \text{id} \ p) \ \text{fcmp} r == r; \]
\[ r \ \text{fcmp} (\ \text{id} \ t) == r; \]
A.7 ConcatRev.lsl

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) \setminus ((n \mem (\dom t)) \setminus ([n-(# s), x] \mem t));
  [n,x] \mem (\rev s) == [(#s) -n+1, x] \mem s;

A.8 DConcat.lsl

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
  __ __\otimes __: 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\}) \forall b b;
  b # x = apply(\bagcount b, x);
  (n \otimes b) # x == n *(b # x);

A.9 DisjointPartitions.lsl

DisjointPartitions(Z): trait
  includes
  ZNumber, FinSet(Z)
  introduces
  __ __\bag __: P[Z] -> P[P[pair[Z,Z]]] % Bags
  __ __\bagcount __: P[pair[Z,Z]] -> P[pair[Z,Z]] % Multiplicity
  __ __\otimes __: P[pair[Z,Z]], Z -> Z
  __ __\otimes __: Z, P[pair[Z,Z]] -> P[pair[Z,Z]] % bag scaling

  asserts
  \forall x: X, p: P[Z], n: Z, b: P[pair[Z,Z]]
  \bag p == p \pfun natone;
  \bagcount b == (universe \prod \{0\}) \forall b b;
  b # x = apply(\bagcount b, x);
  (n \otimes b) # x == n *(b # x);
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
  \in s \implies \left(\in s \implies \exists i,j ((i \neq j) \implies \left(\map(s,i) \equiv \map(s,j) = \emptyset \right)\right)\right) = \left(\disjoint s\right) = \left(\map(s,i) \equiv \map(s,j) = \emptyset \right) \land \left(\disjoint s\right) = \left(\setmap(s) \equiv \set(s) = s\right)\right)

DomRan.lsl

\forall x: X, y: Y, r: P[pair[X,Y]]
  x \in \dom r \equiv \exists y\ (\pair[x,y] \in r)
  y \in \ran r \equiv \exists x\ (\pair[x,y] \in r)
  \implies
  \forall x: X, y: Y, q,r: P[pair[X,Y]]
  \pair[x,y] \in r \implies (\pair[x,y] \in \dom q) \land \pair[x,y] \in q
  \pair[x,y] \in r \implies (\pair[x,y] \in \ran q) \land \pair[x,y] \in q
  \q \uni \ran r = (\q \uni \dom r)
\( \text{ran}(q \ \text{uni} \ r) = (\text{ran} \ q) \ \text{uni} \ (\text{ran} \ r); \)
\( (\text{dom}(q \ \text{int} \ r)) \ \text{subs} \ ((\text{dom} \ q) \ \text{int} \ (\text{dom} \ r)); \)
\( (\text{ran}(q \ \text{int} \ r)) \ \text{subs} \ ((\text{ran} \ q) \ \text{uni} \ (\text{ran} \ r)); \)
\( \text{dom emptyset} = \text{emptyset}; \)
\( \text{ran emptyset} = \text{emptyset}; \)
\( \text{converts} \)
\( \text{dom}_{..}: P[\text{pair}[X,Y]] \rightarrow P[X], \)
\( \text{ran}_{..}: P[\text{pair}[X,Y]] \rightarrow P[Y] \)

A.11 DresRres.lsl

\%
% 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
  \text{dres} \quad \forall x \in X, y \in Y, s \in P[X], t \in P[Y], r \in P[\text{pair}[X,Y]]
  \[ x, y \in \text{mem} (s \ \text{dres} \ r) \Rightarrow x \in \text{mem} s \ \cap \ [x, y] \in \text{mem} r; \]
  \[ x, y \in \text{mem} (r \ \text{rres} \ t) \Rightarrow y \in \text{mem} t \ \cap \ [x, y] \in \text{mem} r; \]

implies
  \forall x \in X, y \in Y, s \in P[X], t \in P[Y], r \in P[\text{pair}[X,Y]]
  \[ s \ \text{dres} \ r = (\text{Id} \ s) \ \text{fcmp} \ r; \]
  \[ s \ \text{dres} \ r = (s \ \text{prod universe}) \ \text{int} \ r; \]
  \[ r \ \text{rres} \ t = r \ \text{fcmp} (\text{Id} \ t); \]
  \[ r \ \text{rres} \ t = r \ \text{int} (\text{universe} \ \text{prod} \ t); \]

\( \text{dom} (s \ \text{dres} \ r) = s \ \text{int} ((\text{dom} \ r)); \)
\( \text{ran} (r \ \text{rres} \ t) = (\text{ran} \ r) \ \text{int} \ t; \)
\( (s \ \text{dres} \ r) \ \text{subs} \ r; \)
\( (r \ \text{rres} \ t) \ \text{subs} \ r; \)
\( (s \ \text{dres} \ r) \ \text{rres} \ t = s \ \text{dres} \ (r \ \text{rres} \ t); \)
\( s \ \text{dres} \ (v \ \text{dres} \ r) = (s \ \text{int} \ v) \ \text{dres} \ r; \)
\( (r \ \text{rres} \ t) \ \text{rres} \ w = r \ \text{rres} \ (t \ \text{int} \ w); \)

43
A.12 DsubRsub.lsl

% @(#)$Id: DsubRsub.lsl,v 1.5 1997/10/03 03: 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, y : X, y : Y, s : P[X], t : P[Y], r : P[pair[X,Y]]
  [x,y] \mem (s \dsub r) \iff x \mem s \&\& [x,y] \mem r;
  [x,y] \mem (r \rsub t) \iff y \mem 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(\success, Z, Z, SuccSet for graph),
  Graph2(<, Z, Z)
introduces
  \_\_ \ires \_ : P[Z], P[pair[Z,X]] -> P[pair[Z,X]] % Extraction from a sequence
  \_\_ \ires \_ : 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
  \forall v \ires s \iff \squash (v \dres s);
  s \ires u \iff \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 $
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) \equiv pass(x) \land (x \mem p)

implies
\forall x,y: X, p: P[X]
  filter(\emptyset) = \emptyset;
  (\forall x ( (x \mem p) \Rightarrow pass(x) = \false))
  \Rightarrow (filter(p) = \emptyset);
  x \mem filter(p) \equiv pass(x) \land (x \mem p);
  (\forall x ( (x \mem p) \Rightarrow (pass(x) = \true)))
  \Rightarrow (filter(p) = p);
converts filter

A.15 FinitePartial.lsl

% $Id: FinitePartial.lsl,v 1.4 1997/10/03 03:59:42 hzhong Exp$
% Written by Hua Zhong with the help of Dr. Gary T. Leavens
% page 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,Y)
introduces
  \_\_ \ffun \_\_ : P[X], P[Y] \rightarrow P[P[pair[X,Y]]]  % Finite Partial functions
  \_\_ \finj \_\_ : P[X], P[Y] \rightarrow 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) \equiv
  f \mem (s \pinj t) \land ((\dom f) \mem (\fset s));
  s \finj t \equiv (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 112 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)
   \quad == (s \sub x) \land (\E n \land (f \mem ((1 \upto n) \\tfun s))
   \quad \land (\ran f = s));
   s \mem (\fsetone x) == (s \mem (\fset x)) \land (s \neq emptyset);
   s \mem (\fset x) =>
   (n = (# s)) = \E f ((f \mem ((1 \upto n) \\tinj s))
   \quad \land (\ran f = s));

A.17 Func.lsl

\% 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

\% 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 \L \{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)) \L (\dom f = universe);
  f \mem (s \pinj t) ==
    (f \mem (s \pfun t)) \L (\{x1, x2\} \mem (\dom f) \L (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)) \L (\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

% 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 all a: P[P[X]], s: P[X], x: X
  x \mem (\duni a) == \E s (x \mem s \L s \mem a);
  x \mem (\dint a) == \A s (s \mem a => x \mem s);
implies
\[ \forall a,b : P[X], s,t : P[X], x: P[X] \]
\[ \text{\texttt{\textbackslash duni}}( a \text{ \textbackslash uni} b) = ( \text{\texttt{\textbackslash duni}} a) \text{ \textbackslash uni} ( \text{\texttt{\textbackslash duni}} b); \]
\[ \text{\texttt{\textbackslash dint}} ( a \text{ \textbackslash uni} b) = ( \text{\texttt{\textbackslash duni}} a) \text{ \textbackslash int} ( \text{\texttt{\textbackslash duni}} b); \]
\[ \text{\texttt{\textbackslash duni}} \text{\texttt{\textbackslash emptyset}} : P[P[X]] = \text{\texttt{\textbackslash emptyset}}; \]
\[ (\text{\texttt{\textbackslash dint}} \text{\texttt{\textbackslash emptyset}} : P[P[X]]) = \text{\texttt{universe}}; \]
\[ \forall x (\text{\texttt{\func}}(x) = (x \text{ \textbackslash int} s) ) \Rightarrow (s \text{ \textbackslash int} (\\text{\texttt{\textbackslash duni}} a)) = (\text{\texttt{\textbackslash duni}} (\text{\texttt{\map}}(a : P[P[X]]))); \]
\[ \forall x (\text{\texttt{\func}}(x) = (x \text{ \textbackslash uni} s) ) \Rightarrow s \text{ \textbackslash uni} (\text{\texttt{\textbackslash dint}} a) = (\text{\texttt{\textbackslash dint}} \text{\texttt{\map}}(a : P[P[X]])); \]
\[ \forall x (\text{\texttt{\func}}(x) = (s \text{ \textbackslash setminus} x)) \Rightarrow \text{\texttt{\textbackslash duni}} a \text{\texttt{\textbackslash setminus}} s = (\text{\texttt{\textbackslash duni}} \text{\texttt{\map}}(a : P[P[X]])); \]
\[ \forall x (\text{\texttt{\func}}(x) = (x \text{ \textbackslash setminus} s)) \Rightarrow \text{\texttt{\textbackslash setminus}} s \text{ \texttt{\setminus}} (\text{\texttt{\textbackslash duni}} a) = (\text{\texttt{\textbackslash duni}} \text{\texttt{\map}}(a : P[P[X]])); \]
\[ (a \text{ \texttt{psubs}} b) \Rightarrow ((\text{\texttt{\textbackslash duni}} a) \text{ \texttt{psubs}} (\text{\texttt{\textbackslash duni}} b)); \]
\[ (a \text{ \texttt{psubs}} b) \Rightarrow ((\text{\texttt{\textbackslash duni}} b) \text{ \texttt{psubs}} (\text{\texttt{\textbackslash duni}} a)); \]

converts
\[ \text{\texttt{\textbackslash duni}} \texttt{\texttt{\_\_}} : P[P[X]] \rightarrow P[X]; \]
\[ \text{\texttt{\textbackslash dint}} \texttt{\texttt{\_\_}} : P[P[X]] \rightarrow P[X] \]

A.21 Graph.lsl

\texttt{Graph}(f, X, Y):\texttt{trait}

\texttt{includes}
\texttt{Function(X, Y)}

\texttt{assumes}
\texttt{Func(f, X, Y)}

\texttt{introduces}
\texttt{graph: \rightarrow P[ pair[ X , Y ]]}

\texttt{asserts}
\texttt{\forall x:X, y:Y}
\texttt{(((x, y)) \texttt{\mem} \texttt{graph}) = f(x) = y;}

\texttt{implies}
\texttt{\forall x:X}
\texttt{\texttt{\apply}(graph, x) = f(x);}

A.22 Graph2.lsl

\texttt{Graph2}(f, X, Y):\texttt{trait}

\texttt{includes}
\texttt{Function(X, Y)}

\texttt{assumes}
\[\text{Func2}(f, X, Y)\]

introduces
\[
\text{graph} : \to P[ \text{pair}[X, Y]]
\]

asserts
\[
\forall x : X, y : Y
\quad
((x, y) \in \text{graph}) \Rightarrow f(x, y);
\]

implies
\[
\forall x : X, y : Y
\quad
\text{apply}(\text{graph}, x) = y \Rightarrow f(x, y);
\]

A.23 \ HeadLastTailFront.lsl

\%
\% 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
\[\text{ZNumber}, \text{Sequences}(N,X), \text{DresRres}(N,X)\]

introduces
\[
\begin{align*}
\text{head} & : \ P[\text{pair}[N,X]] \to X \\
\text{last} & : \ P[\text{pair}[N,X]] \to X \\
\text{tail} & : \ P[\text{pair}[N,X]] \to P[\text{pair}[N,X]] \\
\text{front} & : \ P[\text{pair}[N,X]] \to P[\text{pair}[N,X]]
\end{align*}
\]

asserts
\[
\forall s : P[\text{pair}[N,X]], x : P[\text{pair}[N,X]], n : N
\quad
\begin{align*}
(s \in \text{sequence universe}) & \Rightarrow (\text{head } s) = \text{apply}(s, 1); \\
(s \in \text{sequence universe}) & \Rightarrow (\text{last } s) = \text{apply}(s, \#s); \\
(s \in \text{sequence universe}) & \Rightarrow \\
(\forall n ((n \in \text{sequence universe}) \Rightarrow (\text{tail } s) = \text{apply}(s, (\#s) - 1))) & \Rightarrow (\text{front } s) = (1 \text{upto } (\#s - 1)) \text{dres } s;
\end{align*}
\]

A.24 \ Id.lsl

\%
\% 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 \[\text{Rel}(X,X)\]

introduces
\[
\text{id} : \ P[X] \to P[\text{pair}[X,X]] \% \text{Identify Relation}
\]

49
asserts
\forall x_1, x_2: X, p: P[X]
[x_1, x_2] \in mem(\{id p\}) \equiv x_1 = x_2 \land (x_1 \not\in p);

implies
\forall x_1, x_2: X, p: P[X]
(x_1 \mapsto x_2) \in mem(\{id p\}) \equiv (x_1 = x_2) \land (x_1 \not\in 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), 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) \equiv \{id universe\};
iter(k+1, r) \equiv r \text{ if cmp iter}(k, r);
iter(-k, r) \equiv iter(k, (r \text{ inv}));
r \text{ iter } k \equiv 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
\forall p: pair[X,Y]
[first(p),second(p)] == p;

converts
\forall p: pair[X,Y]
    first: \pair{Z}{X} \rightarrow X,
    second: \pair{Z}{Y} \rightarrow Y,
    -- \prod -- : P[X],P[Y] \rightarrow P[\pair{Z}{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 \rightarrow \text{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), BresRres(Z, X), ConcatRev

introduces
    -- \prefix --: \pair{Z}{X}, \pair{Z}{X} \rightarrow \text{Bool} % Prefix relation
    -- \suffix --: \pair{Z}{X}, \pair{Z}{X} \rightarrow \text{Bool} % Suffix relation
    -- \inseq --: \pair{Z}{X}, \pair{Z}{X} \rightarrow \text{Bool} % Segment relation

asserts
\forall s,t,u,v: \pair{Z}{X}, x: \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 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] \rightarrow P[P[pair[X,Y]]] % relation
  \__ \map _-_: X, Y \rightarrow 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) \Rightarrow ((p \mem (s \rel t)) \Rightarrow x \mem s \setminus y \mem t);

  converts
  \__ \rel _-_: P[X], P[Y] \rightarrow P[P[pair[X,Y]]],
  \__ \map _-_: X, Y \rightarrow 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] \rightarrow 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) \setminus (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

52
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

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

  introduces
  __ \fvr __: P[pair[X,Y]], P[pair[X,Y]] -> P[pair[X,Y]] \% function override

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

A.33 Sequences.lsl

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

  introduces
  \seq __: P[X] -> P[Pair[Z,X]] \% Finite sequences
  \sequone __: P[X] -> P[Pair[Z,X]] \% Non-empty finite sequences
  \isef __: P[X] -> P[Pair[Z,X]] \% Injective sequences
  \rangle \rangle: -> P[Pair[Z,X]]
\[ \forall f : P[pair[Z,X]], x : X, y_1, y_2, y_3 : X \]
\[ f \in mem (\{seq x\}) = (\{ f \in mem (nat \{fun x\}) \} \uplus (\{ dom f \} = (1 \upto (#f)))); \]
\[ f \in mem (\{seq one x\}) = f \in mem (\{seq x\}) \uplus ( (# f) > 0); \]
\[ \{seq x\} = (\{seq x\} \int (nat \{pinj x\}); \]
\[ \forall x \in range == emptyset : P[pair[Z,X]]; \]
\[ \forall y \in range == \{[1, y]\}; \]
\[ \forall y_1, y_2 \in range == \{[1, y_1], [2, y_2]\}; \]
\[ \forall y_1, y_2, y_3 \in range == \{[1, y_1], [2, y_2], [3, y_3]\}; \]

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] \rightarrow P[Y] \% use func to map from one set to another

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

implies
\[ \forall x: X, p: P[X] \]
\[ x \in mem p \rightarrow func(x) \in mem map(p); \]
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
\texttt{minAsSet: \rightarrow P[pair[P[Z],Z]] \hspace{1cm} \% Minimum of a set of numbers}
\texttt{maxAsSet: \rightarrow P[pair[P[Z],Z]] \hspace{1cm} \% Maximum of a set of numbers}
\\texttt{\textbackslash min \_\_ : P[Z] \rightarrow Z \hspace{1cm} \% Minimum of a set of numbers}
\texttt{\textbackslash max \_\_ : P[Z] \rightarrow Z \hspace{1cm} \% Maximum of a set of numbers}

\textbf{asserts}
\\texttt{\textasciitilde \forall s: P[Z], m,n: Z}
\texttt{\hspace{1cm} minAsSet \mem ((\set{\text{psetone \ integer}} \ \ \ \text{pfun \ integer}) \ \ (\{s\} \ \ \text{mem \ minAsSet} =\ (m \ \ \text{mem \ s} \ \ \text{\textbackslash A \ n \ ((n \ \ \text{mem \ s} \ \Rightarrow \ (m <\ n)))})};
\texttt{\hspace{1cm} \textbackslash min \ s \ \Rightarrow \ \text{apply}(\text{minAsSet}, \ s);}
\texttt{\hspace{1cm} \{s\} \ \ \text{mem \ maxAsSet} =\ (m \ \ \text{mem \ s} \ \ \text{\textbackslash A \ n \ ((n \ \ \text{mem \ s} \ \Rightarrow \ (m \ \ \text{\leq} \ n)))});
\texttt{\hspace{1cm} \textbackslash max \ s \ \Rightarrow \ \text{apply}(\text{maxAsSet}, \ s);}

\textbf{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
\texttt{includes \texttt{ZSet(X)}}
\texttt{\texttt{introduces \texttt{\_\_ un\_\_ : P[X],P[X] \rightarrow P[X]} \hspace{1cm} \% set union}
\texttt{\texttt{\_\_ \texttt{int\_\_ : P[X],P[X] \rightarrow P[X]} \hspace{1cm} \% set intersection}
\texttt{\texttt{\_\_ \texttt{setminus \_\_ : P[X],P[X] \rightarrow P[X]} \hspace{1cm} \% set difference}}

\textbf{asserts}
\texttt{\texttt{\textasciitilde \forall s,t: P[X], x: X}}
\texttt{\hspace{1cm} x \text{ mem \ (s \ \text{\textbackslash uni \ t})} =\ (x \ \text{mem \ s} \ \backslash / \ x \ \text{mem \ t});}
\texttt{\hspace{1cm} x \text{ mem \ (s \ \text{int \ t})} =\ (x \ \text{mem \ s} \ \backslash / \ x \ \text{mem \ t});}
\texttt{\hspace{1cm} x \text{ mem \ (s \ \text{setminus \ t})} =\ (x \ \text{mem \ s} \ \backslash / \ (x \ \text{mem \ t}))};

\textbf{implies}
\texttt{\texttt{\texttt{Distributive(\text{\textbackslash uni},\text{\textbackslash int},P[X]),}}}
\texttt{\% \text{\textbackslash int for \text{\textbackslash glb}, \text{\textbackslash uni for \text{\textbackslash lub},}}}
\texttt{\texttt{\texttt{Semi\_\_lattice(\text{emptyset for \text{\textbackslash bot}, \text{\textbackslash int for \text{\textbackslash glb}, \text{\textbackslash uni for \text{\textbackslash lub},}}}}}
\texttt{\texttt{\texttt{\texttt{\texttt{\subs for \text{\leq}, \psubs for \text{\leq}, \sup for \text{\geq}, \psups for \text{\geq},}}}}}
\texttt{\texttt{\texttt{P[X] for \text{T}}}})

\% \texttt{\textasciitilde \forall s,t,v: P[X]}
\texttt{\hspace{1cm} s \ \text{\textbackslash uni \ s} =\ s;}
\texttt{\hspace{1cm} s \ \text{\textbackslash int \ s} =\ s;}
\texttt{\hspace{1cm} s \ \text{emptyset \ \text{\textbackslash setminus \ emptyset} =\ emptyset;}}
\texttt{\hspace{1cm} s \ \text{\textbackslash setminus \ emptyset} =\ emptyset;}
\texttt{\hspace{1cm} emptyset \ \text{\textbackslash setminus \ emptyset} =\ emptyset;}
\texttt{\hspace{1cm} (s \ \text{\textbackslash setminus \ t}) \ \text{\textbackslash int \ t} =\ emptyset;али

55
\[
s \setminus (t \setminus v) == (s \setminus t) \setminus v;
\]
\[
s \setminus (t \setminus v) == (s \setminus t) \setminus (s \setminus v);
\]
\[
(s \setminus t) \setminus v == (s \setminus v) \setminus (t \setminus v);
\]
\[
(s \setminus v) \setminus (t \setminus v) == (s \setminus t) \setminus (s \setminus v);
\]

A.37 TclRtcl.lsl

% @(#)Id: TclRtcl.lsl,v 1.4 1997/10/03 03: 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
   \texttt{\textbackslash tcl} : P[pair[X,X]] -> P[pair[X,X]] \texttt{ % Transitive closure}
   \texttt{\textbackslash rtcl} : P[pair[X,X]] -> P[pair[X,X]] \texttt{ % Reflexive-transitive closure}

asserts
\forall x,y,z : X, s : P[X], r,s : P[pair[X,X]]
   \begin{align*}
   [x,y] \text{\em mem} (r \\texttt{\textbackslash tcl}) &==
      ([x,y] \text{\em mem} r \land ([x,y] \text{\em mem} (r \\texttt{\textbackslash tcl}))); \\
   [x,y] \text{\em mem} (r \\texttt{\textbackslash rtcl}) &==
      (x=y) \land
      ([x,y] \text{\em mem} r \land ([x,y] \text{\em mem} (r \\texttt{\textbackslash tcl})));
   \end{align*}

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

ZNumber(Z): trait

includes Integer(Z), ZPowerSet(Z), Pair(Z,Z)

introduces
   nat : -> P[Z] \texttt{ % Natural Numbers}
   integer : -> P[Z] \texttt{ % Integers}

56
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

% 0(#)$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 \mem 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 \mem
forall x: X, s,t: P[X]
x \mem emptyset;
s = t == \A x (x \mem s = x \mem t);
A.40  ZSet.lsl

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

\forall s,t: P[X], x: X
  s \subs t == (\A x (x \mem s => x \mem t));
  s \mem (\pses t) == s \subs t;
  t \mem (\psetes s) == t \mem (\pses s) /\ \neg(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 == \neg(s = \emptyset);
  (\psetes s) = emptyset == s = emptyset;
  \neg(s = emptyset) == s \mem (\psetes s);
converts
  -- \subs __ : P[X], P[X] -> Bool,
  -- \psubs __ : P[X], P[X] -> Bool,
  \pses __: P[X] -> P[P[X]],
  \psetes __ : P[X] -> P[P[X]]
Appendix B

Examples for the Larch Prover

In this appendix I present two illustrative 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

evaluate 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
\[
\square \Rightarrow \text{subgoal}
\]
\[
\square \text{conjecture}
\]
\[
\text{qed}
\]
\[
\text{prove}
\]
\[
(x \in \text{emptyset})
\]
\[
\square \text{conjecture}
\]
\[
\text{prove}
\]
\[
(s = t \Rightarrow \forall x (x \in s = (x \in t)))
\]
\[
\square \text{conjecture}
\]
\[
\text{prove}
\]
\[
(\forall x (x \in s = (x \in t)) \Rightarrow s = t)
\]
\[
\Rightarrow \text{by}
\]
\[
\Rightarrow \text{subgoal}
\]
\[
\text{apply ZPowerSet.1 to conj}
\]
\[
\Rightarrow \text{subgoal}
\]
\[
\square \text{conjecture}
\]
\[
\text{prove}
\]
\[
(x \in s /\ x \in s) = (false)
\]
\[
\square \text{conjecture}
\]
\[
\text{qed}
\]

%% Conversions

freeze ZPowerSet

%% converts -- \in\ --: X, P[X] → Bool

thaw ZPowerSet

declare operators

-- \in\' --: X, P[X] → Bool

..

% subtrait 0: ZPowerSet (-- \in\' --: X, P[X] → Bool for -- \in\ --: X, P[X] → Bool)

set name ZPowerSet

assert

(x \in\ p) = (= (\neg (x \in p))

..

declare variables
set name conversionChecks

prove \( (\_x_1\ \neg\text{em}\ \_x_2) = (\_x_1\ \neg\text{em'}\ \_x_2) \)
\[\square\] conjecture

qed

prove \( (\_x_1\ \neg\text{em}\ \_x_2) = (\_x_1\ \neg\text{em'}\ \_x_2) \)
\[\square\] 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 \neq bottom) by con
  ..
  \langle\rangle contradiction subgoal
  \[\square\] contradiction subgoal
  \[\square\] 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, types) "= bottom)
..
□ conjecture
qed

prove
(~ isBottom(st) => eval(obj1, bind(st, obj1, v, types)) = v)
..
i instantiate obj by obj1 in State_Basics.6
□ conjecture
qed

prove
(~ isBottom(st) => eval(obj1, bind(st, obj, v, types)) = (if obj1 = obj then
eval(obj1, bind(emptyState, obj1, v, types)) else eval(obj1, st)))
..
res by =>
<> => subgoal
res by case obj1 = obj
<> case obj1c = objc
p eval(objc, bind(stc, objc, v, types)) = v
res by con
<> contradiction subgoal
crit *Hyp with *
□ contradiction subgoal
□ conjecture
res by con
<> contradiction subgoal
crit *Hyp with *
□ contradiction subgoal
\(\text{case objc = objc}\)

\(\text{\textless\textgreater\ case } \neg(\text{objc = objc})\)

\(\text{res by con}\)

\(\text{\textless\textgreater\ contradiction subgoal}\)

\(\text{crit *Hyp with *}\)

\(\text{\textgreater\ contradiction subgoal}\)

\(\text{\textgreater\ case } \neg(\text{objc = objc})\)

\(\text{\textgreater\ => subgoal}\)

\(\text{\textgreater\ conjecture}\)

\textit{qed}

\textit{prove}

\((-\text{isBottom(st)} \Rightarrow \text{allocated(obj1, bind(st, obj, v, typs)) = (if obj1 = obj then allocated(obj1, bind(emptyState, obj, v, typs)) else allocated(obj1, st))})\)

\(\text{..}\)

\(\text{res by =>}\)

\(\text{\textless\textgreater\ => subgoal}\)

\(\text{\textgreater\ => subgoal}\)

\(\text{\textgreater\ conjecture}\)

\textit{qed}

\textit{prove}

\((\text{bind(bind(st, obj, v, typs), obj, v1, typs1)) = (\text{bind(st, obj, v1, typs1}))\)

\(\text{..}\)

\textit{prove}

\(\text{eval(obj1,bind(bind(st, obj, v, typs), obj, v1, typs1)) = (eval(obj1,bind(st, obj, v1, typs1))}\)

\(\text{..}\)

\textit{declare variables v2: Value typs2: Set[TYPE]}\)

\textit{ins st by bind(st, obj, v2, typs2) in State_Basics.6}\)

\(\text{res by case st =\textless\textgreater\ bottom}\)

\(\text{\textgreater\ case stc =\textless\textgreater\ bottom}\)

\(\text{res by case obj1 = obj}\)

\(\text{\textless\textgreater\ case objc = objc}\)

\(\text{crit * with *Hyp}\)

\(\text{\textgreater\ case objc = objc}\)

\(\text{\textless\textgreater\ case } \neg(\text{objc = objc})\)

\(\text{crit * with *Hyp}\)

\(\text{\textgreater\ case } \neg(\text{objc = objc})\)

\(\text{\textgreater\ case stc =\textless\textgreater\ bottom}\)

\(\text{\textless\textgreater\ case } \neg(\text{stc =\textless\textgreater\ bottom})\)

\textit{qed}

\(\text{set immu on}\)

\textit{prove}\n
\(\text{bind(bottom, obj, v, typs) = bottom}\)

\(\text{\textless\textgreater\ conjecture}\)

\(\text{\textgreater\ case } \neg(\text{stc =\textless\textgreater\ bottom})\)

\(\text{\textgreater\ conjecture}\)

\(\text{set immu off}\)

\textit{prove}

63
allocated(obj1, bind(bind(st, obj, v, types), obj, v1, types))
   = allocated(obj1, bind(st, obj, v1, types))
   ..
   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, types), obj1, v1, types)) = (if obj1 = obj then
      bind(st, obj1, v1, types) else bind(bind(st, obj1, v1, types1), obj, v, types))
   ..
   declare variables obj2: Object v2: Value types2: Set[TYPE]
   prove allocated(obj2, bind(bind(st, obj, v, types), obj1, v1, types))
   = allocated(obj2, if obj1 = obj then bind(st, obj1, v1, types1)
   else bind(bind(st, obj1, v1, types1), obj, v, types))
   ..
   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, types), obj1, v1, types))
   = eval(obj2, if obj1 = obj then bind(st, obj1, v1, types1)
   else bind(bind(st, obj1, v1, types1), obj, v, types))
   ..
   res by case st = bottom
   <> case stc = bottom
   set immu on
   prove bind(bottom, obj, v, types) = bottom
   [] conjecture
   [] case stc = bottom
   <> case ~(stc = bottom)
   set immu off

64
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 = objic
p eval(objic, 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 = objic
<> case ~(obj2c = objic)
res by case obj2c = objc
<> case obj2c = objc
p eval(objc, bind(bind(stc, objc, v, typs), objic, v1, typs1))=v1
res by con
<> contradiction subgoal
crit *Hyp with *
□ contradiction subgoal
□ conjecture
ins st by bind(stc, objc, v1, typs1) in State_Basics.6
□ case obj2c = objc
<> case ~(obj2c = objc)
p eval(obj2c, bind(bind(stc, objc, v, typs), objic, 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 = objic)
□ case ~(obj1c = objic)
□ 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
\textbf{Conversions}

\textbf{freeze State\_Basics}

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

\textbf{thaw State\_Basics}

\textbf{declare operators}

\begin{align*}
\text{allocated}' & : \text{Object, State} \rightarrow \text{Bool} \\
\text{isBottom}' & : \text{State} \rightarrow \text{Bool} \\
\text{eval}' & : \text{Object, State} \rightarrow \text{Value}
\end{align*}

\%\% subtrait 0: State\_Basics (allocated': Object, State \rightarrow Bool for allocated: Object, State \rightarrow Bool, eval': Object, State \rightarrow Value for eval: Object, State \rightarrow Value, isBottom': State \rightarrow Bool for isBottom: State \rightarrow Bool)

\textbf{set name State\_Basics}

\textbf{assert}

\textbf{sort State partitioned by allocated', eval', isBottom'}

\%\% \texttt{\neg allocated'(obj, emptyState)}
\%\% \texttt{\neg allocated'(obj, bottom)}
\%\% \texttt{(allocated'(obj, bind(st, obj1, v, tys))) = (\neg isBottom'(st) \surd (obj = obj1 \surd allocated'(obj, st)))}
\%\% \texttt{(\neg isBottom'(st) => eval'(obj1, bind(st, obj, v, tys))) = (if obj1 = obj then v else eval'(obj1, st))}
\%\% \texttt{(\neg isBottom'(emptyState))}
\%\% \texttt{(isBottom'(bottom))}
\%\% \texttt{(isBottom'(bind(st, obj, v, tys))) = (isBottom'(st))}

\textbf{declare variables}

\begin{align*}
\text{obj} & : \text{Object} \\
_x1_ : & \text{Object} \\
_x1_ : & \text{State} \\
_x2_ : & \text{State}
\end{align*}

\textbf{set name exemptions}

\textbf{assert}

\begin{align*}
\text{eval(obj, bottom)} & = \text{eval'(obj, bottom)} \\
\text{eval(obj, emptyState)} & = \text{eval'(obj, emptyState)}
\end{align*}
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 (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
<> conjunction
<> 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 _x1_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


