

# Appendix A

## Predefined Mathematical Constructs in CARE

The CARE toolset has some built-in predefined theory, based on the Z Mathematical toolkit [87]. A number of extensions have been made to this toolkit; the extensions used in this thesis are defined below in CARE theory notation.

**Appending a element to a list:** the function *append* creates a new list by appending an element to the front of an existing list.

Theory definition of *append*.  
 $append : X \times seq X \rightarrow seq X$ ;  
 $\forall h : X; t : seq X \bullet append(h, t) = \langle h \rangle \hat{\ } t$

**Non-repeating lists:** the relation *IsNonRep* describes those lists which don't contain any repetitions.

Theory definition of *IsNonRep*.  
 $IsNonRep : seq X$ ;  
 $IsNonRep(\langle \rangle)$ ,  
 $\forall h : X; t : seq X \bullet IsNonRep(append(h, t)) \Leftrightarrow \neg h \in ran t \wedge IsNonRep(t)$

**Ordered lists:** the relation *IsOrdered* is a relation which describes those lists which are ordered with respect to some ordering.

Theory definition of *IsOrdered*.

$IsOrdered : seq X;$

$IsOrdered(\langle \rangle),$

$\forall h : X; t : seq X \bullet IsOrdered(append(h, t)) \Leftrightarrow$   
 $(t = \langle \rangle \vee h \leq head t \wedge IsOrdered(t))$

**Singleton set:** the function *mkSet* creates a singleton set from an element.

Theory definition of *mkSet*.

$mkSet : X \rightarrow \mathbb{F} X;$

$\forall x : X \bullet mkSet(x) = \{x\}$

The CARE tools use an ASCII representation of the language. Table. A.1 gives a conversion from the pretty printed mathematical symbols used in this thesis to the ASCII representation used by the tools. The conversion of other operators (e.g. individual numbers, arithmetic operators, singleton set, etc.) are obvious.

Symbol	Description	ASCII
true	truth	True
$\wedge$	logical and	&
$\vee$	logical or	
$\neg$	logical not	~
$\Rightarrow$	logical implication	=>
$\Leftrightarrow$	logical equivalence	<=>
$\forall$	universal quantifier	All
$\exists$	existential quantifier	Exists
$\neq$	not equal	~=
seq	sequence	ListsOf
$\langle \rangle$	empty sequence	nil
$\wedge$	sequence concatenation	concat
rev	sequence reverse	rev
$\mathbb{F}$	finite sets	SetsOf
$\in$	set membership	In
$\cup$	set union	union
$\mathbb{N}$	natural numbers	Nat

Table A.1: ASCII representations for mathematical symbols

# Appendix B

## CARE tool syntax

Note that the CARE prototype tools use an ASCII representation for units, which is slightly different to the pretty printed language used in this thesis. The differences were elided in Chapter 2 in the interest of simplicity. They are largely superficial and an artifact of the CARE methodology’s development history. More details of the concrete syntax for the CARE language are given in [42, 32].

### B.1 Simple fragments

A *simple fragment* has the following form:

```
name(inputvars)<<precond>> spec
  ::= body.
```

where

`name` is the name of the fragment.

`inputvars` is a list consisting of the fragment’s input variables, with their corresponding types.

`precond` is the fragment’s “precondition” (or “applicability condition”), describing the circumstances under which the fragment’s correctness will be ensured. The

precondition is a Z-term representing a predicate of the input variables (see the examples below).

*spec* is the fragment's specification (see Section B.3 below).

*body* is a fragment language term representing the implementation of the fragment (see Section B.4 below).

Intuitively, a simple fragment represents the class of all functions whose domain is given by *precond* and whose values satisfy the constraints given in the specification. Fragment evaluation can also have side-effects (which get described in the specification part using a hidden state), but because of space limitations this paper will only consider pure side-effect-free fragments.

## B.2 Branching fragments

A *branching* fragment with two branches has the following form:

```
name(inputvars)<<precond>>
  |1> <<guard>>spec1
  |e> spec2
::=body.
```

where *name*, *inputvars*, *precond* and *body* are as for simple fragments and *guard* is the fragment's "branching condition". As for the precondition, the test is a Z-term representing a predicate of the input variables.

*spec1*, *spec2* are the "specification parts" of the branching fragment. *spec1* specifies the "success branch" of the fragment (explained below) and *spec2* specifies the "failure branch".

## B.3 Fragment specifications

The *specification* of a fragment is that part up to, but not including, the body. This corresponds roughly to a function declaration in a C or Pascal program, but is much more mathematically expressive.

### B.3.1 The specification part of a simple fragment

Specification parts are divided into two groups: direct and indirect.

#### B.3.1.1 Direct specifications

A *direct specification* is simply a list of Z-terms (separated by commas) denoting the values returned by the fragment, together with the corresponding types.

#### Examples:

```
hd(s:List)<< s ~= nil>>head(s):Elem.
```

`hd` returns the initial element of a list. The precondition is that `s` is a non-empty sequence ( $s \neq \langle \rangle$ ). The value returned by `hd(s)` is *head s*.

```
nil()<<true>> nil:List.
```

`nil` returns the empty sequence.

```
concat(s,t:List) concat(s,t):List .
```

`concat` returns the concatenation of the two lists `s` and `t`. Note that a trivial precondition may be omitted.

#### B.3.1.2 Indirect specifications

An *indirect specification part* has the form

```
preval::outvars;postval <<postcond>>
```

where

`preval` is a Z-term (called the “prevalue”) representing a function of the input variables.

`outvars` is a list consisting of the output variables.

`postval` is a Z-term (called the “postvalue”) representing a function of the input and output variables.

`postcond` is a Z-term (called the “postcondition”) representing a predicate of the input and output variables.

Intuitively, the family of functions represented by the simple fragment

```
name(inputvars)<<precond>>preval::outvars;postval<<postcond>> .
```

consists of those functions with domain given by *precond* and co-domain given by *postcond* such that the values of the function are constrained by the following relationship: the value of *preval* before the execution of the fragment should equal the value of *postval* after execution.

Note that an direct specification can always be converted into an equivalent indirect specification by adding output variables: e.g.

```
hd(s:List)<<s ~= nil>>head(s)::y:Elem;y <<true>>.
```

### Examples:

Given positive integers  $a$  and  $b$ , the following fragment finds the integer quotient and remainder when  $a$  is divided by  $b$ :

```
int_divide(a,b:Int)<<a>0 & b>0>> a::q,r:Int;q*b+r<<0 <= r & r < b>>.
```

The specification part of `int_divide(a,b)` says that  $a = q * b + r$  and  $0 \leq r < b$  (or in other words, that  $r = a \bmod b$ ). Thus for example, `int_divide(17,3)` produces the pair (5,2).

The next fragment finds an index at which an element  $x$  occurs in a sequence  $s$ , under the assumption that  $x$  is an element of  $s$  ( $x \in \text{ran } s$ ):

```
findIndex(x:Elem,s>List)<<x In ran(s)>>x::i:Nat;s(i)<<1 <= i & i <= #s>>.
```

### B.3.2 The specification parts of a branching fragment

Recall that a branching fragment specification with two branches has the form

```
name(inputvars)<<precond>>
  |1> <<guard>> spec1
  |e> spec2 .
```

Intuitively, this can be read as “if *guard* holds, then return values which satisfy *spec1*, otherwise return values that satisfy *spec2*”. The specification parts *spec1* and *spec2* can be direct or indirect, or they can be missing altogether (i.e., they are optional). The specification parts of the two branches are independent, and the two branches can produce different numbers, and even types, of results. When the specification part is missing, control simply branches and no result is returned by the branch.

#### Examples:

In the following example, both specification parts are missing. The fragment simply causes branching: neither branch produces a result.

```
empty(s>List)
  |1> <<s=nil>> |e> .
```

The next fragment is in some sense the inverse of `cons`. Given a sequence *s*, it branches according to whether or not *s* is empty. In the success case, it produces no results. (The success specification part is missing.) In the fail case — when *s* is non-empty — the fragment returns two results, namely the head and tail of *s*. (The fail specification part is an indirect specification.)

```
snoc(s) |1> <<s=nil>>
  |e> s::h:Elem,t>List; append(h,t).
```

## B.4 Fragment implementations

The implementation (*body*) part of a fragment is written in a high level language which composes fragments by assigning the results of fragment calls to local variables. Statements in the fragment language have a tree-like structure, with assignments at the nodes. (Branching fragment calls give rise to branches in the tree.) When written out in linear form, branches of the tree corresponding to simple fragment calls are separated by semicolons (;). Branches corresponding to branching fragment calls are indicated by `|1>` and `|e>`. The leaves of the tree are (lists of) simple fragment calls and/or variables. Recursive calls are allowed.

### B.4.1 Implementing simple fragments

Some of the examples of fragment headers given in Section B.3 would be implemented directly by target language code constructs. This section illustrates the kind of fragments that the software engineer might write in the fragment language. Let us assume that fragments `hd`, `tl`, `nil`, `cons`, `snoc` and `empty` have already been implemented and satisfy their specifications as given in Section B.3.

A fragment for constructing singleton sequences is

```
mkseq(x:Elem) mkSeq(x):List
 ::= cons(x,nil).
```

The body consists of a call to the `nil` fragment nested inside a call to the `cons` fragment.

An alternative (but equivalent) implementation would be to introduce a new local variable, `s` say, and to bind `s` to the result returned by `nil` (written `nil::s`), and then to call `cons(x,s)`:

```
mkseq(x:Elem) mkSeq(x):List
 ::= nil::s;
    cons(x,s).
```

The `concat` fragment defined below concatenates two sequences together and returns the result:

```
concat(s1,s2>List) concat(s1,s2):List
 ::=snoc(s1) |1> s2
      |e>: :h,t;cons(h,concat(t,s2)).
```

The call to `snoc` corresponds to a test. The fragment header for `snoc(s1)` indicates that if `s1` is the empty sequence then `snoc` returns no values; in this case the success branch (on the line following the ‘|1>’) is used, and `concat` returns the single value `s2`. If, on the other hand, `s1` is non-empty, then the failure branch (following the ‘|e>’) is taken and `snoc` returns a pair of values — namely, the head and tail of `s1`. The results are bound to `h` and `t` and the `cons` fragment gets applied to `h` and the result of the recursive call `concat(t,s2)`.

The following example finds the last element in a sequence:

```
end(s>List)<<s~= nil>> last(s):Elem
 ::= tl(s)::t;
      empty(t) |1> hd(s)
      |e> end(t).
```

### B.4.2 Implementing branching fragments

Branching fragments are implemented via trees of fragment calls as for simple fragment implementations, *except that* each path ends in a “chooser” (<1| or <e|) or a branching fragment call. The chooser indicates whether the path through the body to this point corresponds to the success (<1|) or failure (<e|) case of the fragment header. In order to illustrate the implementation of branching fragments, let us assume we have fragments with headers as follows:

Testing whether two elements are equal:

```
equals(x,y:Elem) |1> <<x=y>> |e> .
```

Testing whether an integer is positive:

```
is_positive(n:Int) |1> << 0 <= n>> |e> .
```

The software engineer could use these to build up fragments as follows:

Finding the absolute value of an integer:

```
abs(n:Int) |1> <<0\leq n>> n:Int
          |e> -n:Int
 ::= is_positive(n) |1> n <1|
          |e> neg(n) <e|.
```

The `abs` fragment is implemented as follows: `is_positive(n)` is used to test if  $n$  is positive; if so, the fragment returns  $n$  and reports success; otherwise, it calls `neg(n)` and reports failure. The examples which follow are self-explanatory.

De-constructing a sequence:

```
snoc(s>List) |1> <<s=nil>>
          |e> s::h:Elem,t>List;append(h,t)
 ::=empty(s) |1> <1|
          |e> hd(s),tl(s) <e|.
```

Checking whether  $x$  is an element of sequence  $s$ :

```
member(x:Elem,s>List) |1> <<x In ran(s)>> |e>
 ::= snoc(s) |1> <e|
          |e>::h:Elem,t>List;
          equals(h,x) |1> <1|
          |e> member(x,t).
```

## B.5 Types

The notation for type specifications is fairly simple, with the symbol “==” replacing the keywords “has specification”.

# Appendix C

## A Partial Library of CARE

### Templates

In this appendix, we give the collection of templates used in this thesis, which makes up part of the CARE library. The templates are described in more detail earlier in the thesis, in particular in Section 2.7. The library contains a number of templates not listed here, but is currently still relatively small. We would expect the library to be considerably larger in a commercial release of such a tool.

Two templates for list accumulation are used in this thesis. The first of these, the so-called standard `accumulator` template is given in Fig C.1. The algorithm is implemented by successively applying a so-called “step function” to each element in the list step-by-step, progressing from the first through to the last element, and accumulating an intermediate value at each stage. The accumulator is started with a given base value. A special case of the standard accumulator is the *associative commutative accumulator* template, given in Fig. C.2. This template is much the same as the standard accumulator, except that the step function is assumed to be AC.

The `if then intro` template given in Fig. C.3 on page 273 implements the main fragment `ifthenintro` by introducing an exception case via a call to the branching fragment `bfrag`. For this exception, the second argument of the fragment is returned. Otherwise the result returned is governed by an auxiliary fragment

frag.

Three templates containing refinements of sets in terms of lists are used in this thesis. The first of these, sets in terms of (possibly repeating) *lists* is given in Fig. C.6. For this particular refinement, the set is represented by the range of the list. The second of the data refinements is sets as *non-repeating lists* (see Fig. C.7). The set is represented by the range of the list and an invariant is given stating that the list representing the set cannot contain any repetitions. The third data refinement is sets in terms of *ordered lists* (see Fig. C.8). The same refinement relation as for the previous two templates is used, but in this case the underlying list must be ordered. Each template contains the fragment `abOp1` used for implementing binary operations on sets, and the fragment `abOp2` for performing operations on a set and an element. These set manipulating fragments are all parameterised, so they can be adapted to solve a number of problems.

The natural numbers as unsigned integers template given in Fig C.5, contains primitives for representing natural numbers as unsigned integers in C. A number of fragments which perform basic operations on these data structures are included in the template: included are addition, subtraction, multiplication, division, equality and inequality tests, together with fragments for implementing the constants “0” and “1”.

A template for representing sequences as (singly) linked lists in C is given in Fig. C.4. The template is parameterised over the type of the underlying list elements, and can be instantiated to give lists of different types. Primitive components are given for: appending elements to the list, calculating the head and tail of a list, representing the empty list, and testing whether a list is empty.

<u>Template: Accumulator</u>
<u>Formal parameters:</u> $E, Acc, f : \text{seq } E \rightarrow Acc, hd : Acc \times E \rightarrow Acc, dh : E \times Acc \rightarrow Acc, base : E.$
<u>Applicability conditions:</u> $f(\langle \rangle) = base,$ $\forall h : E, t : \text{seq } E \bullet f(\text{append}(h, t)) = dh(h, f(t)),$ $\forall x : E \bullet dh(x, base) = hd(base, x),$ $\forall x, y : E; a : Acc \bullet dh(x, hd(a, y)) = hd(dh(x, a), y).$
<u>Theories:</u> $fold : \text{seq } E \times Acc \rightarrow Acc;$ $\forall y : Acc \bullet fold(\langle \rangle, y) = y,$ $\forall h : E, t : \text{seq } E, y : Acc \bullet fold(\text{append}(h, t), y) = fold(t, hd(y, h))$
<u>Types:</u> Type List has specification: $\text{seq } E$ Type Element has specification: $E$ Type Acc has specification: $Acc$
<u>Fragments:</u> Fragment <code>processList(x:List)</code> has specification: output $y:Acc$ such that $y = f(x)$ implementation: return accumulator(x, base).  Fragment <code>accumulator(x:List, y:Acc)</code> has specification: output $z:Acc$ such that $z = fold(x, y)$ implementation: cases decomposeList(x) of: empty: return y nonempty: assign output to $v:Element, w:List;$ assign processElem(v, y) to $y:Acc;$ return accumulator(w, y)  Fragment <code>processElem(e:Element, a:Acc)</code> has specification: output $r:Acc$ such that $r = hd(e, a).$  Fragment <code>base()</code> has specification: output $b:Acc$ such that $b = base.$  Branching fragment <code>decomposeList(x:List)</code> has specification: if $x = \langle \rangle$ then report empty else report nonempty with output $h:Element, t:List$ such that $x = \text{append}(h, t).$

Figure C.1: The Accumulator template

Template: Associative Commutative Accumulator
<u>Formal parameters:</u> $E, f : \text{seq } E \rightarrow E, hd : E \times E \rightarrow E, base : E.$
<u>Applicability conditions:</u> $f(\langle \rangle) = base,$ $\forall h : E, t : \text{seq } E \bullet f(\text{append}(h, t)) = hd(f(t), h),$ $\forall x, y : E \bullet hd(x, y) = hd(y, x),$ $\forall x, y, z : E \bullet hd(x, hd(y, z)) = hd(hd(x, y), z).$
<u>Theories:</u> $fold : \text{seq } E \times Acc \rightarrow Acc;$ $\forall y : Acc \bullet fold(\langle \rangle, y) = y,$ $\forall h, y : E, t : \text{seq } E \bullet fold(\text{append}(h, t), y) = fold(t, hd(h, y))$
<u>Types:</u> Type List has specification: $\text{seq } E$ Type Element has specification: $E$
<u>Fragments:</u> Fragment <code>processList(x:List)</code> has specification: output <code>y:Element</code> such that $y = f(x)$ implementation: return <code>accumulator(x, base)</code> .  Fragment <code>accumulator(x:List, y:Element)</code> has specification: output <code>z:Element</code> such that $z = fold(x, y)$ implementation: cases <code>decomposeList(x)</code> of: empty: return <code>y</code> nonempty: assign output to <code>v:Element, w:List</code> ; assign <code>processElem(v, y)</code> to <code>y:Element</code> ; return <code>accumulator(w, y)</code>  Fragment <code>processElem(e:Element, a:Element)</code> has specification: output <code>r:Element</code> such that $r = hd(e, a)$ .  Fragment <code>base()</code> has specification: output <code>b:Element</code> such that $b = base$ .  Branching fragment <code>decomposeList(x:List)</code> has specification: if $x = \langle \rangle$ then report empty else report nonempty with output <code>h:Element, t:List</code> such that $x = \text{append}(h, t)$ .

Figure C.2: The Associative Commutative Accumulator template

Template: If then intro
<u>Formal parameters:</u> $X, Y, f : X \times Y \rightarrow Y, R : X \times Y,$ $P : X \times Y, Q : X \times Y \times Y$
<u>Applicability conditions:</u> $\forall x : X, y : Y \bullet R(x, y) \wedge P(x, y) \Rightarrow Q(x, y, y)$ $\forall x : X, y : Y, z : Y \bullet \neg R(x, y) \wedge P(x, y) \wedge z = f(x, y) \Rightarrow Q(x, y, z)$
<u>Types:</u> Type U has specification: $X$ . Type V has specification: $Y$ .
<u>Fragments:</u> Fragment <code>ifthenintro(x:U,y:V)</code> has specification: precondition $P(x, y)$ output $z:V$ such that $Q(x, y, z)$ implementation: cases <code>bfrag(x,y)</code> of: <code>ifcase</code> : assign $y$ to $z:V$ ; return $z$ <code>thencase</code> : assign <code>frag(x,y)</code> to $z:V$ ; return $z$  Branching fragment <code>bfrag(x:U,y:V)</code> has specification: result defined by cases if $R(x, y)$ then report <code>ifcase</code> else report <code>thencase</code>  Fragment <code>frag(x:U,y:V)</code> has specification: output $z:V$ such that $z = f(x, y)$

Figure C.3: The if then intro template

Template: Linked lists
Formal parameters: $E$ .
Types: Type Element has specification: $E$ . Type List has specification: $\text{seq}(E)$ implementation: value $v$ 1: "< List > 1" Assign: 1 associated code: "struct linked_list {< Element > val; struct linked_list * next;};", "typedef struct linked_list POINTER;", "typedef POINTER * < List >;".
Fragments: Fragment $\text{apndl}(e:\text{Element}, v:\text{List})$ has specification: output $l0:\text{List}$ such that $l0 = \text{append}(e, v)$ implementation: <<target code for allocation of cell and linking into list elided>>. Fragment $\text{nil}()$ has specification: output $l1:\text{List}$ such that $l1 = \langle \rangle$ implementation: <<target code elided>>. Branching fragment $\text{apndlI}(v:\text{List})$ has specification: result defined by cases if $v \neq \langle \rangle$ then report non-empty with output $h:\text{Element}, t:\text{List}$ such that $\text{append}(h, t) = v$ else report empty implementation: cases $\text{isEmpty}(v)$ of: yes: report empty and return no: assign $\text{hdl}(v)$ to $h:\text{Element}$ ; assign $\text{tll}(v)$ to $t:\text{List}$ ; report non-empty and return $h, t$ Branching fragment $\text{isEmpty}(v:\text{List})$ has specification: result defined by cases if $v = \langle \rangle$ then report yes else report no implementation: <<target code elided>>. Fragment $\text{hdl}(v:\text{List})$ has specification: precondition $v \neq \langle \rangle$ output $e0:\text{Element}$ such that $e0 = \text{head } v$ implementation: <<target code elided>>. Fragment $\text{tll}(v:\text{List})$ has specification: precondition $v \neq \langle \rangle$ output $l2:\text{List}$ such that $l2 = \text{tail } v$ implementation: <<target code elided>>.

Figure C.4: Linked lists

Template: Natural numbers
<p><u>Types:</u></p> <p>Type Num has specification: <math>\mathbb{N}</math>  implementation:  value <math>v</math> with constraint <math>v \leq \text{uintmax}</math> <math>i: \langle   \text{Num}   \rangle i</math> Assign: <math>i</math>  associated code:  <code>"typedef unsigned int &lt; Num &gt;;"</code>.</p>
<p><u>Fragments:</u></p> <p>Branching fragment <code>lessthan(x:Num,y:Num)</code> has specification:  result defined by cases  if <math>x &lt; y</math> then report <code>yes</code> else report <code>no</code>  implementation:  <code>&lt;&lt;target code elided&gt;&gt;</code>.</p> <p>Branching fragment <code>lessthaneq(x:Num,y:Num)</code> has specification:  result defined by cases  if <math>x \leq y</math> then report <code>yes</code> else report <code>no</code>  implementation:  <code>&lt;&lt;target code elided&gt;&gt;</code>.</p> <p>Branching fragment <code>equal(x:Num,y:Num)</code> has specification:  result defined by cases  if <math>x = y</math> then report <code>yes</code> else report <code>no</code>  implementation:  <code>&lt;&lt;target code elided&gt;&gt;</code>.</p> <p>Fragment <code>add(x:Num,y:Num)</code> has specification:  output <math>z: \text{Num}</math> such that <math>z = x + y</math>  implementation:  <code>&lt;&lt;target code elided&gt;&gt;</code>.</p> <p>Fragment <code>mult(x:Num,y:Num)</code> has specification:  output <math>z: \text{Num}</math> such that <math>z = x * y</math>  implementation:  <code>&lt;&lt;target code elided&gt;&gt;</code>.</p> <p>Fragment <code>zero()</code> has specification:  output <math>n: \text{Num}</math> such that <math>n = 0</math>  implementation:  <code>&lt;&lt;target code elided&gt;&gt;</code>.</p> <p>Fragment <code>one()</code> has specification:  output <math>n: \text{Num}</math> such that <math>n = 1</math>  implementation:  <code>&lt;&lt;target code elided&gt;&gt;</code>.</p> <p>Fragment <code>increment(m:Num)</code> has specification:  output <math>n: \text{Num}</math> such that <math>n = m + 1</math>  implementation:  <code>&lt;&lt;target code elided&gt;&gt;</code>.</p>

Figure C.5: Natural numbers

Template: Sets as repeating lists
<p>Formal parameters:</p> $E, P1 : \mathbb{F} E \times \mathbb{F} E, Q1 : \mathbb{F} E \times \mathbb{F} E \times \mathbb{F} E, P2 : \mathbb{F} E \times E, Q2 : \mathbb{F} E \times \mathbb{F} E \times E.$
<p>Types:</p> <p>Type Elem has specification: <math>E</math>.</p> <p>Type Set has specification: <math>\mathbb{F} E</math> refinement: value <math>s</math> is refined by <math>l:List</math> with refinement relation <math>s = \text{ran } l</math>.</p> <p>Type List has specification: <math>\text{seq } E</math>.</p>
<p>Fragments:</p> <p>Fragment <math>\text{abOp1}(s:Set, t:Set)</math> has specification: precondition <math>P1(s, t)</math> output <math>r:Set</math> such that <math>Q1(r, s, t)</math> implementation: decompose <math>s</math> into <math>sc:List</math>; decompose <math>t</math> into <math>tc:List</math>; assign <math>\text{conOp1}(sc, tc)</math> to <math>rc:List</math>; compose <math>rc</math> into <math>r:Set</math>; report <math>e</math> and return <math>r</math></p> <p>Fragment <math>\text{conOp1}(s:List, t:List)</math> has specification: precondition <math>P1(\text{ran } s, \text{ran } t)</math> output <math>r:List</math> such that <math>Q1(\text{ran } r, \text{ran } s, \text{ran } t)</math></p> <p>Fragment <math>\text{abOp2}(s:Set, e:Elem)</math> has specification: precondition <math>P2(s, e)</math> output <math>r:Set</math> such that <math>Q2(r, s, e)</math> implementation: decompose <math>s</math> into <math>sc:List</math>; assign <math>\text{conOp2}(sc, e)</math> to <math>rc:List</math>; compose <math>rc</math> into <math>r:Set</math>; report <math>e</math> and return <math>r</math></p> <p>Fragment <math>\text{conOp2}(s:List, e:Elem)</math> has specification: precondition <math>P2(\text{ran } s, e)</math> output <math>r:List</math> such that <math>Q2(\text{ran } r, \text{ran } s, e)</math></p>

Figure C.6: Sets refined by (possibly) repeating lists

Template: Sets as non-repeating lists	
<u>Formal parameters:</u>	
$E, P1 : \mathbb{F} E \times \mathbb{F} E, Q1 : \mathbb{F} E \times \mathbb{F} E \times \mathbb{F} E, P2 : \mathbb{F} E \times E, Q2 : \mathbb{F} E \times \mathbb{F} E \times E.$	
<u>Types:</u>	
Type <code>Elem</code> has specification: $E$ .	
Type <code>Set</code> has specification: $\mathbb{F} E$	
refinement:	
value <code>s</code> is refined by <code>l:List</code>	
with invariant $IsNonRep(l)$	
with refinement relation $s = \text{ran } l$ .	
Type <code>List</code> has specification: $\text{seq } E$ .	
<u>Fragments:</u>	
Fragment <code>abOp1(s:Set,t:Set)</code> has	
specification:	
precondition $P1(s, t)$	
output <code>r:Set</code> such that $Q1(r, s, t)$	
implementation:	
decompose <code>s</code> into <code>sc:List</code> ;	
decompose <code>t</code> into <code>tc:List</code> ;	
assign <code>conOp1(sc,tc)</code> to <code>rc:List</code> ;	
compose <code>rc</code> into <code>r:Set</code> ;	
report <code>e</code> and return <code>r</code>	
Fragment <code>conOp1(s:List,t:List)</code> has	
specification:	
precondition $P1(\text{ran } s, \text{ran } t) \wedge IsNonRep(s) \wedge IsNonRep(t)$	
output <code>r:List</code> such that $IsNonRep(r) \wedge Q1(\text{ran } r, \text{ran } s, \text{ran } t)$	
Fragment <code>abOp2(s:Set,e:Elem)</code> has	
specification:	
precondition $P2(s, e)$	
output <code>r:Set</code> such that $Q2(r, s, e)$	
implementation:	
decompose <code>s</code> into <code>sc:List</code> ;	
assign <code>conOp2(sc,e)</code> to <code>rc:List</code> ;	
compose <code>rc</code> into <code>r:Set</code> ;	
report <code>e</code> and return <code>r</code>	
Fragment <code>conOp2(s:List,e:Elem)</code> has	
specification:	
precondition $P2(\text{ran } s, e) \wedge IsNonRep(s)$	
output <code>r:List</code> such that $IsNonRep(r) \wedge Q2(\text{ran } r, \text{ran } s, e)$	

Figure C.7: Sets as non-repeating lists

Template: Sets as ordered lists
<p><u>Formal parameters:</u></p> $E, P1 : \mathbb{F} E \times \mathbb{F} E, Q1 : \mathbb{F} E \times \mathbb{F} E \times \mathbb{F} E, P2 : \mathbb{F} E \times E, Q2 : \mathbb{F} E \times \mathbb{F} E \times E.$
<p><u>Types:</u></p> <p>Type <code>Elem</code> has specification: <math>E</math>.</p> <p>Type <code>Set</code> has specification: <math>\mathbb{F} E</math>  refinement:  value <code>s</code> is refined by <code>l:List</code>  with invariant <math>IsOrdered(l)</math>  with refinement relation <math>s = \text{ran } l</math>.</p> <p>Type <code>List</code> has specification: <math>\text{seq } E</math>.</p>
<p><u>Fragments:</u></p> <p>Fragment <code>abOp1(s:Set,t:Set)</code> has  specification:  precondition <math>P1(s, t)</math>  output <code>r:Set</code> such that <math>Q1(r, s, t)</math>  implementation:  decompose <code>s</code> into <code>sc:List</code>;  decompose <code>t</code> into <code>tc:List</code>;  assign <code>conOp1(sc,tc)</code> to <code>rc:List</code>;  compose <code>rc</code> into <code>r:Set</code>;  report <code>e</code> and return <code>r</code></p> <p>Fragment <code>conOp1(s:List,t:List)</code> has  specification:  precondition <math>P1(\text{ran } s, \text{ran } t) \wedge IsOrdered(s) \wedge IsOrdered(t)</math>  output <code>r:List</code> such that <math>IsOrdered(r) \wedge Q1(\text{ran } r, \text{ran } s, \text{ran } t)</math></p> <p>Fragment <code>abOp2(s:Set,e:Elem)</code> has  specification:  precondition <math>P2(s, e)</math>  output <code>r:Set</code> such that <math>Q2(r, s, e)</math>  implementation:  decompose <code>s</code> into <code>sc:List</code>;  assign <code>conOp2(sc,e)</code> to <code>rc:List</code>;  compose <code>rc</code> into <code>r:Set</code>;  report <code>e</code> and return <code>r</code></p> <p>Fragment <code>conOp2(s:List,e:Elem)</code> has  specification:  precondition <math>P2(\text{ran } s, e) \wedge IsOrdered(s)</math>  output <code>r:List</code> such that <math>IsOrdered(r) \wedge Q2(\text{ran } r, \text{ran } s, e)</math></p>

Figure C.8: Sets as ordered lists