AFFIRM Type Library
Susan L. Gerhart, Editor

Where's the Stack?
AFFIRM

Type Library

Susan L. Gerhart, Editor

Version 2.0 - February 19, 1981
Corresponds to AFFIRM Version 1.21

USC Information Sciences Institute
4676 Admiralty Way
Marina Del Rey, California 90291
(213) 822-1511  ARPANET: AFFIRM@ISIF

Copyright © 1981, USC/Information Sciences Institute
The AFFIRM Reference Library

AFFIRM is an experimental interactive system for the specification and verification of abstract data types and programs. It was developed by the Program Verification Project at the USC Information Sciences Institute (ISI) for the Defense Advanced Research Projects Agency. The Reference Library is composed of five documents:

Reference Manual
A detailed discussion of the major concepts behind AFFIRM presented in terms of the abstract machines forming the system's structure as seen by the user.

Users Guide
A question-and-answer dialogue detailing the whys and wherefores of specifying and proving using AFFIRM.

Type Library
A listing of several abstract data types developed and used by the ISI Program Verification Project. The data type specifications are maintained in machine-readable form as an integral part of the system.

Annotated Transcripts
A series of annotated transcripts displaying AFFIRM in action, to be used as a sort of workbook along with the Users Guide and Reference Manual.

Collected Papers
A collection of articles authored by members of the ISI Program Verification Project (past and present), as well as an annotated bibliography of recent papers relevant to our work.

Program Verification Project Members


Cover designs by Nelson Lucas.

Special dedication to Affirmed, the only race horse named after a verification system.

This research was supported by the Defense Advanced Research Projects Agency and the Rome Air Defense Command. Views and conclusions are the authors'.
# Table of Contents

1. Introduction
   1.1. Data Types
      1.1.1. File Naming Conventions
      1.1.2. Parameterization: The Instantiation Model
      1.1.3. Exercises and Tutorials
   1.2. Programs

2. DATA TYPES
   2.1. ELEM TYPE
      2.1.1. Discussion
      2.1.2. Specification
   2.2. SEQUENCE
      2.2.1. Discussion
      2.2.2. Specification
         2.2.2.1. Basic Axioms
         2.2.2.2. Additional Axioms
         2.2.2.3. Additional Definitions
      2.2.3. Representation Functions
         2.2.3.1. From Mapping
         2.2.3.2. From Queue
         2.2.3.3. To Circle
      2.2.4. Lemmas
   2.3. CIRCLE
      2.3.1. Discussion
      2.3.2. Specification
   2.4. QUEUE
      2.4.1. Discussion
      2.4.2. Specification
   2.5. SET
      2.5.1. Discussion
      2.5.2. Specification
      2.5.3. Lemmas
   2.6. MAPPING
      2.6.1. Discussion
      2.6.2. Specification
   2.7. GRAPH
      2.7.1. Discussion
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.7.2.</td>
<td>Specification</td>
<td>27</td>
</tr>
<tr>
<td>2.7.3.</td>
<td>Lemmas</td>
<td>31</td>
</tr>
<tr>
<td>2.8.</td>
<td>BINARYTREE</td>
<td>34</td>
</tr>
<tr>
<td>2.8.1.</td>
<td>Discussion</td>
<td>34</td>
</tr>
<tr>
<td>2.8.2.</td>
<td>Axioms</td>
<td>34</td>
</tr>
<tr>
<td>2.8.2.1.</td>
<td>Basic Axioms</td>
<td>34</td>
</tr>
<tr>
<td>2.8.2.2.</td>
<td>Additional Axioms</td>
<td>36</td>
</tr>
<tr>
<td>2.9.</td>
<td>INTEGER</td>
<td>37</td>
</tr>
<tr>
<td>2.9.1.</td>
<td>Discussion</td>
<td>37</td>
</tr>
<tr>
<td>2.9.2.</td>
<td>Specification</td>
<td>37</td>
</tr>
<tr>
<td>2.9.3.</td>
<td>Lemmas</td>
<td>38</td>
</tr>
<tr>
<td>3.</td>
<td>LESSON</td>
<td>39</td>
</tr>
<tr>
<td>3.1.</td>
<td>Discussion</td>
<td>39</td>
</tr>
<tr>
<td>3.2.</td>
<td>Notation</td>
<td>39</td>
</tr>
<tr>
<td>3.3.</td>
<td>&quot;Theorems&quot;</td>
<td>41</td>
</tr>
<tr>
<td>Appendix I.</td>
<td>PROGRAMS</td>
<td>42</td>
</tr>
<tr>
<td>I.1.</td>
<td>Remove Blanks</td>
<td>42</td>
</tr>
<tr>
<td>I.2.</td>
<td>Remove Duplicates</td>
<td>48</td>
</tr>
<tr>
<td>I.3.</td>
<td>SimpleSend</td>
<td>51</td>
</tr>
<tr>
<td>I.4.</td>
<td>A Sorting Algorithm</td>
<td>53</td>
</tr>
<tr>
<td>Appendix II.</td>
<td>PVLIBRARY</td>
<td>61</td>
</tr>
</tbody>
</table>
1. Introduction

The AFFIRM Type Library is a collection of types developed and used extensively for some time by members of the ISI PV project. The Type Library is maintained as an integral part of the AFFIRM system. Its purpose is to

1. provide users, whether new or old, with ready-made well-constructed types to use for their own purposes, e.g., as the data types of a program to be verified;

2. provide models for development of additional types or further development of the existing types;

3. collect a good set of test cases for AFFIRM system maintenance and enhancement;

4. serve as examples of the range of concepts related to abstract data types, axiomatic specifications, rewrite rules, induction proofs, and data type interactions with programs; and

5. stimulate further research into the individual theories associated with each type, the structure of these theories, and the interaction between types and theories.

The material of the AFFIRM Type Library is referred to elsewhere in the Reference Manual and User’s Guide, so the present manual will primarily archive and document the separate files stored in the library.

1.1. Data Types

The Users' Guide discusses the validity of data types. As far as we know now, there is nothing in the types provided here that would destroy their validity. We hope to formally establish this validity in a forthcoming AFFIRM memo.

The present volume is not necessarily comprehensive. The Type Library will undoubtedly be extended with more types or with more notation and lemmas. We'll try to record any changes in a separate file.
1.1.1. File Naming Conventions

A set of file name conventions has been established that allows the system to automatically access files containing type specifications via the needs command.

The Type Library is stored as a set of Tenex or Tops-20 files in a directory normally named <PVLIBRARY>. File names in the Tenex and Tops-20 operating systems consist of three parts: a name, an extension, and a version number. The file name conventions imposed in the type library are as follows:

1. The name field is always a type name.

2. The extension field is used to further qualify the contents of a file associated with a particular type. The extensions we have defined so far include: empty, COM, LISP, and AXIOMS. The preceding extensions are meaningful to the needs command, while the other names are purely conventions for organizing <PVLIBRARY>.

3. The version field is not usually explicitly mentioned.

The file organization and naming scheme will be keyed to the separate types as follows:

typeName..
   (Here the extension field is empty.) The saved version of the type, i.e., the uncompiled version of the functions and environment created for a type.

typeName.COM
   The compiled version of the typeName.

typeName.AXIOMS
   The text version of axioms, in a form readable by AFFIRM. These are produced by print type typeName and then edited out of a transcript. Library Invariants (hopefully) are

   \[ \text{Save(Read(typeName.AXIOMS))} = \text{typeName..} \]

   \[ \text{print type typeName(read(typeName.AXIOMS))} = \text{typeName.AXIOMS} \]

   The second invariant is not in general true, due to difficulties in precedence (See the discussion of Idempotent I/O in the AFFIRM Users' Guide.)

typeName.SomethingNOTATION
   Notation and operations which extend typeName, but are neither common to the base type (if parameterized) nor properly considered part of the data type.

typeName.LEMMAS
A source (text) version of lemmas related to typeName. Maintained in the form "assume
nm, expr; annotate nm, something". These can be read in and put on the proof structure as
assumed and annotated. If not all lemmas are desired, then edit this file to include only
those wanted immediately in the proof structure. Annotations were automatically added
using the auto profile setting but were edited to refer to appropriate files.

typeName.PROOFS
Final proof structure of lemma proofs and the table of lemma usage.

typeName.DOCUMENTATION
Setup for the corresponding section in this document, including comments on the type,
printable versions of the axioms, etc.

typeName.CHANGES
Summary of changes of the type after publication of the AFFIRM Reference Library.

1.1.2. Parameterization: The Instantiation Model
Parameterization of types is not automatically handled in AFFIRM. Therefore the structured types
have been "relativized" to ElemType, a type with a minimally defined equality relation (there is one
axiom that states the reflexive property). This type may be "instantiated" by editing references to
ElemType to be the desired type. See the Users' Guide for a description of this process.

1.1.3. Exercises and Tutorials
One "lesson" has been prepared so far. See Chapter 3, but don't peek at the proofs until you have
done them yourself.

1.2. Programs
The Type Library is also the initial repository for program proofs, which are organized as follows:

ProgramName.PROGRAM
The main program or procedure, ready to be read.

ProgramNameCONTEXT
A source file of all the declarations, interfaces, axioms and definitions required for
the proof of ProgramName.PROGRAM.

ProgramName.LEMMAS
The verification conditions and associated lemmas.

ProgramName.PROOFS
Printable file of the program, context, lemmas and VCs, and their proofs.

Since the programs are not considered standard, we have placed them in the Appendix. They may be of interest for the style of proof and syntax that we have found workable and advisable. Other examples of programs appear in *AFFIRM* memos [Thompson 80, Gerhart 80a, Wing 80, Lee 80, Gerhart 80b, Gerhart 80c] and in [Thompson 81].
2. DATA TYPES

2.1. ELEMTYPE

2.1.1. Discussion
This is what you get as a "minimally specified" type, namely just a dummy variable and the reflexivity of equality axiom. Domain and Range are used later as minimally specified types for the mapping structure. The automatically declared variable, here dummy, may be set by a profile entry.

2.1.2. Specification

\begin{verbatim}

\texttt{type ElemType;}

\texttt{declare dummy: ElemType;}

\texttt{axiom dummy = dummy = = TRUE;}

\texttt{end \{ElemType\};}

\end{verbatim}
2.2. SEQUENCE

2.2.1. Discussion
The Sequence type is by far the best developed and most extensively used of all the types in the Type Library. The theory structure of Sequence is illustrated by the various lemmas (usually properties relating pairs of operations) and their pattern of usage.

It may be preferable to use Sequence rather than Queue for just this very reason. Remember that Stacks and Queues are just disciplines on adding and removing elements from Sequences, that is, just a subset of the possible operations on Sequences. Thus the theory of Stacks and Queues is a sub-theory (in some sense) of that of Sequences.

This data type also illustrates significant use of the Knuth-Bendix algorithm in generating appropriate rewrite rules for "normalizing" extender operations $apl$ and $join$ to $apr$. The current ordering of axioms requires no interaction with Knuth-Bendix but considerable experimentation was required to find this order.

Sequence also shows the different ways of specifying operations, define and axiom.

An often-used variation, SequenceOfInteger, is also maintained. See the SelectSort Program in Appendix I for examples of its use and for further notation.

There is one more Induction schema that usual, one which inducts on the front rather than the end of a sequence. We once thought this schema was useful and justified it via the standard (constructor) induction rule, but we have not found many uses for it recently. See the Users' Guide for a discussion of schemas.

2.2.2. Specification
To save space when several types are loaded, the specification has been broken into the basic operations on sequences, axioms for additional functions, and definitions for additional functions.
2.2.2.1. Basic Axioms

type SequenceOfElemType;

needs type ElemType;

declare dummy, ss, s, s1, s2: SequenceOfElemType;
declare k, ii, i, i1, i2, j: ElemType;

interfaces {Constructors} NewSequenceOfElemType, s apr i,
{Extenders} i apl s, seq(i), s1 join s2, LessFirst(s),
LessLast(s): SequenceOfElemType;

infIX join, apl, apr;

interfaces isNew(s), FirstInduction(s), Induction(s), NormalForm(s), i in s: Boolean;

infIX in;

interfaces First(s), Last(s): ElemType;

interface Length(s): Integer;

axioms dummy = dummy = = TRUE,
NewSequenceOfElemType = s apr i = = FALSE,
s apr i = NewSequenceOfElemType = = FALSE,
s apr i = s1 apr i1 = = ((s = s1) and (i = i1));

axioms i apl NewSequenceOfElemType = = NewSequenceOfElemType apr i,
i apl (s apr i1) = = (i apl s) apr i1;

axiom seq(i) = = NewSequenceOfElemType apr i;

axioms NewSequenceOfElemType join s = = s,
(s apr i) join s1 = = s join (i apl s1);

axiom LessFirst(s apr i) = = if s = NewSequenceOfElemType
then NewSequenceOfElemType
else LessFirst(s) apr i;

axiom LessLast(s apr i) = = s;

axiom isNew(s) = = (s = NewSequenceOfElemType);
axioms i in NewSequenceOfElemType = FALSE,
   i in (s apl i1) = (i in s or (i = i1));

axiom First(s apl i) = if s = NewSequenceOfElemType
    then i
    else First(s);

axiom Last(s apl i) = i;

axioms Length(NewSequenceOfElemType) = 0,
   Length(s apl i) = Length(s) + 1;

rulelemmas NewSequenceOfElemType = i apl s = FALSE,
   i apl s = NewSequenceOfElemType = FALSE;

rulelemmas s join (s1 apl i) = (s join s1) apl i,
   s join NewSequenceOfElemType = s,
   i apl s1) join s2 = i apl (s1 join s2),
   (s join (i apl s1)) join s2 = s join (i apl (s1 join s2)),
   s join (s1 join s2) = (s join s1) join s2;

rulelemma LessFirst(i apl s) = s;

rulelemma LessLast(i apl s) = if s = NewSequenceOfElemType
    then NewSequenceOfElemType
    else i apl LessLast(s);

rulelemma i in (i1 apl s) = (i in s or (i = i1));

rulelemma First(i apl s) = i;

rulelemma Last(i apl s) = if s = NewSequenceOfElemType
    then i
    else Last(s);

schemas FirstInduction(s)
   = cases(Prop(NewSequenceOfElemType), all ss, ii (IH(ss)
      imp Prop(ii apl ss))),

Induction(s)
   = cases(Prop(NewSequenceOfElemType), all ss, ii (IH(ss)
      imp Prop(ss apr ii))),

NormalForm(s)
   = cases(Prop(NewSequenceOfElemType), all ss, ii (Prop(ss apr ii)));

end {SequenceOfElemType} ;
2.2.2.2. Additional Axioms
See file SEQUENCEOFELMTYPE.ADDLAXIOMS

needs types SequenceOfElemType, ElemType;

declare s, s1, s2: SequenceOfElemType;
declare i: ElemType;

interfaces dedup(s), reverse(s): SequenceOfElemType;

interfaces nodups(s), s1 subseq s2, s1 disjoint s2: Boolean;

infix subseq, disjoint;

axioms dedup(NewSequenceOfElemType) = = NewSequenceOfElemType,
dedup(s apr i) = = if i in s
then dedup(s)
else dedup(s) apr i;

axioms reverse(NewSequenceOfElemType) = = NewSequenceOfElemType,
reverse(s apr i) = = i apl reverse(s);

axioms nodups(s apr i) = = (nodups(s) and ~(i in s)),
nodups(NewSequenceOfElemType) = = TRUE;

axioms s subseq NewSequenceOfElemType = = (s = NewSequenceOfElemType),
s1 subseq (s apr i)
= = ( (s1 = NewSequenceOfElemType) or s1 subseq s
or LessLast(s1) subseq s and (Last(s1) = i));

axioms NewSequenceOfElemType disjoint s = = TRUE,
(s apr i) disjoint s1 = = (s disjoint s1 and ~(i in s1));
2.2.2.3. Additional Definitions
See file SEQUENCEOELEMETYPE.ADDLDEFNS

needs types SequenceOfElemType, ElemType;

declare s: SequenceOfElemType;
declare k: Integer;

interfaces Rotate(s, k), Initial(s, k), LessInitial(s, k), deletepth(s, k)
     : SequenceOfElemType;

interface pth(s, k): ElemType;

define Rotate(s, k)
   = if (s = NewSequenceOfElemType) or (k = 0)
      then s
      else if 1 <= k
         then Rotate(LessFirst(s) apr First(s), k-1)
         else Rotate(Last(s) apl LessLast(s), k + 1),

Initial(s, k)
   = if (s = NewSequenceOfElemType) or (k <= 0)
      then NewSequenceOfElemType
      else First(s) apl Initial(LessFirst(s), k-1),

LessInitial(s, k)
   = if (s = NewSequenceOfElemType) or (k <= 0)
      then s
      else LessInitial(LessFirst(s), k-1),

deletepth(s, k)
   = if k <= 0
      then s
      else if k = 1
         then LessFirst(s)
         else First(s) apl deletepth(LessFirst(s), k-1),

pth(s, k)
   = if k = 1
      then First(s)
      else pth(LessFirst(s), k-1);
2.2.3. Representation Functions

Many examples of representation function use appear in [Gerhart 81] and in the programs in Appendix II.

2.2.3.1. From Mapping

```plaintext
declare a:MappingFromIntegerToElementType;
declare lb, ub: Integer;
declare x: ElementType;

interface rep(a, lb, ub): SequenceOfElementType;
define rep(a, lb, ub) = = if lb>ub then NewSequenceOfElementType
else rep(a, lb, ub-1) apr (a sub ub);
```

2.2.3.2. From Queue

```plaintext
declare q: QueueOfElementType;
declare i: ElementType;

interface rep(q): SequenceOfElementType;
axioms rep{NewQueueOfElementType} = = NewSequenceOfElementType,
rep(q Add i) = = rep(q) apr i;
```

2.2.3.3. To Circle

```plaintext
declare s: SequenceOfElementType;
declare i: ElementType;
declare k: Integer;

interfaces rep1(s), rep2(s, k): CircleOfElementType;

Two different ways of mapping sequences to circles are defined - with the pointer assumed to be
at the front of the sequence and with it explicit as a parameter.

axioms rep1(NewSequenceOfElementType) = = NewCircleOfElementType,
rep1(s apr i) = = InsertLast(rep1(s), i);
```

```plaintext
define rep2(s, k) = = Rotate(s, k-1);
```
2.2.4. Lemmas

```plaintext
declare s, s1, s2, s3, s4: SequenceOfElemType;
declare \kappa_1, \kappa_2: Integer;
declare i: ElemType;

First and Last Lemmas

assume LastSplit, isNew(s) or (LessLast(s) and Last(s) = s);

assume FirstSplit, isNew(s) or (s = First(s) and LessFirst(s));

Length Lemmas

assume LengthLessLast, isNew(s) or (Length(LessLast(s)) = Length(s) - 1);
assume LengthLessFirst, isNew(s) or (Length(LessFirst(s)) = Length(s) - 1);
assume LengthNonNeg, Length(s) \geq 0;
assume LengthNew, Length(s) = 0 eqv isNew(s);
assume LengthJoin, Length(s1 join s2) = Length(s1) + Length(s2);

Join Lemmas

assume LastJoin, if isNew(s2)
    then Last(s1 join s2) = Last(s1)
    and LessLast(s1 join s2) = LessLast(s1)
    else Last(s1 join s2) = Last(s2)
    and LessLast(s1 join s2) = s1 join LessLast(s2);

assume FirstJoin, if isNew(s1)
    then First(s1 join s2) = First(s2)
    and LessFirst(s1 join s2) = LessFirst(s2)
    else First(s1 join s2) = First(s1)
    and LessFirst(s1 join s2) = LessFirst(s1) join s2;

assume SpecialJoin, s1 join s2 = s3
    imp s join s3 = (s join s1) join s2;

assume apleq, i apl s1 = i apl s2 eqv s1 = s2;
assume join1steq, s join s1 = s join s2 imp s1 = s2;
```
\textbf{12}

\begin{verbatim}
assume join2ndeq, s1 join s = s2 join s imp s1 = s2;
assume join1stNew, s1 join s2 = s2 imp isNew(s1);
assume join2ndNew, s1 join s2 = s1 imp isNew(s2);
assume join2New, isNew(s1 join s2) eqv isNew(s1) and isNew(s2);
assume SplitAt1, ~isNew(s2)
    and ~isNew(s3)
    and (s1 = LessLast(s3)) and (s1 join s2 = s3 join s4)
    imp s4 = LessFirst(s2);

Initial and LessInitial Lemmas

assume InitialFirstLast, (1 \leq p) and (p \leq \text{Length}(s))
    imp Initial(s, p-1)
    = LessLast(Initial(s, p))
    and LessInitial(s, p)
    = LessFirst(LessInitial(s, p-1));
assume InitialLessLast, (1 \leq p) and (p \leq \text{Length}(s))
    imp Initial(s, p-1) = LessLast(Initial(s, p));
assume LessInitialJoin, (0 \leq p) and (p \leq \text{Length}(s1) + \text{Length}(s2))
    imp LessInitial(s1 join s2, p)
    = if p \leq \text{Length}(s1)
        then LessInitial(s1, p) join s2
        else LessInitial(s2, p - \text{Length}(s1));
assume InitialJoin, (0 \leq p) and (p \leq \text{Length}(s1) + \text{Length}(s2))
    imp Initial(s1 join s2, p)
    = if p \leq \text{Length}(s1)
        then Initial(s1, p)
        else s1 join Initial(s2, p - \text{Length}(s1));
assume InitialLength, (0 \leq p) and (p \leq \text{Length}(s))
    imp \text{Length}(Initial(s, p)) = p
    and \text{Length}(LessInitial(s, p)) = \text{Length}(s) \cdot p;
assume LengthInitial, 0 \leq p
    imp \text{Length}(Initial(s, p)) = \min(p, \text{Length}(s));
assume InitialOutOfBounds, p < 0
\end{verbatim}
\begin{align*}
\text{imp} & \quad \text{isNew}(\text{Initial}(s, p)) \\
\text{and} & \quad \text{LessInitial}(s, p) = s \\
\text{and} & \quad p \geq \text{Length}(s) \quad \text{imp} \quad \text{isNew}(\text{LessInitial}(s, p)) \\
\text{and} & \quad \text{Initial}(s, p) = s; \\
\end{align*}

**assume** \ InitialSplit, Initial(s, p) join LessInitial(s, p) = s;

\textit{Rotate Lemmas}

**assume** \ RotateTwice, Rotate(Rotate(s, k1), k2) = Rotate(s, k1 + k2);

**assume** \ RotateNew, isNew(Rotate(s, p)) eqv isNew(s);

**assume** \ LengthRotate, Length(Rotate(s, p)) = Length(s);

**assume** \ RotateOverLength, p \geq \text{Length}(s) \quad \text{imp} \quad \text{Rotate}(s, p) = \text{Rotate}(s, p - \text{Length}(s));

**assume** \ RotateInitial, (0 \leq p) \text{ and } (p \leq \text{Length}(s)) \\
\text{imp} \quad \text{Rotate}(s, p) \\
= \text{LessInitial}(s, p) \text{ join } \text{Initial}(s, p);

\textit{Lemmas for pth}

**assume** \ pthOverJoin, (1 \leq p) \text{ and } (p \leq \text{Length}(s1 \text{ join } s2)) \\
\text{imp} \quad \text{pth}(s1 \text{ join } s2, p) \\
= (\text{if } p \leq \text{Length}(s1) \\
\quad \text{then } \text{pth}(s1, p) \\
\quad \text{else } \text{pth}(s2, p - \text{Length}(s1))) \\
\text{and } \quad \text{deletpth}(s1 \text{ join } s2, p) \\
= (\text{if } p \leq \text{Length}(s1) \\
\quad \text{then } \text{deletpth}(s1, p) \text{ join } s2 \\
\quad \text{else } s1 \text{ join } \text{deletpth}(s2, p - \text{Length}(s1)));

**assume** \ pthSplits, (1 \leq p) \text{ and } (p \leq \text{Length}(s)) \\
\text{imp} \quad \text{Initial}(s, p-1) \text{ apr } \text{pth}(s, p) \\
\text{join } \text{LessInitial}(s, p) \\
= s \\
\text{and } \quad \text{pth}(s, p) = \text{First}(\text{LessInitial}(s, p-1)) \\
\text{and } \quad \text{pth}(s, p) = \text{Last}(\text{Initial}(s, p)) \\
\text{and } \quad \text{deletpth}(s, p) \\
= \text{Initial}(s, p-1) \text{ join } \text{LessInitial}(s, p);
2.3. CIRCLE

2.3.1. Discussion
Very little use has been found for the circle type, except in the Josephus Circle example [Gerhart 81]. Like queues, its axioms are similar to sequences and its theory is therefore isomorphic to a sub-theory of sequences.

2.3.2. Specification

```plaintext
type CircleOfElemType;

needs type ElemType;

declare c, c', c1, c2, cc, dummy: CircleOfElemType;
declare i, i', i1, i2, j: ElemType;
declare k: Integer;

interfaces {Constructors} NewCircleOfElemType, InsertLast(c, i),
{Extenders} InsertFirst(c, i), DeleteLast(c),
DeleteFirst(c), RotateLeft(c), RotateRight(c), InsertCircle(c1, c2),
Rotate(c, k): CircleOfElemType;

interfaces
  First(c), Last(c): ElemType;

interface Size(c): Integer;

interfaces
  l in c, isNew(c), Induction(c), NormalForm(c): Boolean;

infix in;

axioms
  dummy = dummy = TRUE,
  NewCircleOfElemType = InsertLast(c, i) = FALSE,
  InsertLast(c, i) = NewCircleOfElemType = FALSE,
  InsertLast(c, i) = InsertLast(c1, i1) = ((c = c1) and (i = i1));

axioms
  InsertFirst(NewCircleOfElemType, i) = InsertLast(NewCircleOfElemType, i),
```

InsertLast(InsertLast(c, i), j) = = InsertLast(InsertFirst(c, j), i);

axioms
DeleteLast(NewCircleOfElemType) = = NewCircleOfElemType,
DeleteLast(InsertLast(c, i)) = = c;

axioms
DeleteFirst(NewCircleOfElemType) = = NewCircleOfElemType,
DeleteFirst(InsertLast(c, i))
    = = if c = NewCircleOfElemType
        then c
        else InsertLast(DeleteFirst(c), i);

axioms
RotateLeft(NewCircleOfElemType) = = NewCircleOfElemType,
RotateLeft(InsertLast(c, i)) = = InsertFirst(c, i);

axioms
RotateRight(NewCircleOfElemType) = = NewCircleOfElemType,
RotateRight(InsertLast(c, i))
    = = InsertLast(if c = NewCircleOfElemType
        then c
        else InsertLast(DeleteFirst(c), i),
            First(InsertLast(c, i)));

axioms
InsertCircle(c, NewCircleOfElemType) = = c,
InsertCircle(c, InsertLast(c1, i)) = = InsertLast(InsertCircle(c, c1), i);

axiom First(InsertLast(c, i)) = = if c = NewCircleOfElemType
    then i
    else First(c);

axiom Last(InsertLast(c, i)) = = i;

axioms
Size(NewCircleOfElemType) = = 0,
Size(InsertLast(c, i)) = = Size(c) + 1;

axioms
i in NewCircleOfElemType = = FALSE,
i in InsertLast(c, i1) = = ((i = i1) or i in c);

axiom isNew(c) = = (c = NewCircleOfElemType);

rulelemmas
InsertFirst(c, i) = NewCircleOfElemType = FALSE,
NewCircleOfElemType = InsertFirst(c, i) = FALSE;

**rulelemma** DeleteFirst(InsertFirst(c, i)) = = c;

*Note that Rotate here differs from the bi-directional Rotate in SequenceOfElemtype. The definition can easily be changed to match.*

**define** Rotate(c, k)
    = if c = NewCircleOfElemType
        then NewCircleOfElemType
        else if k = 0
            then c
            else Rotate(RotateRight(c), k-1);

**schema**

Induction(c)
    = cases(Prop(NewCircleOfElemType),
        all cc, ii (IH(cc) imp Prop(InsertLast(cc, ii))));

NormalForm(c) = cases(Prop(NewCircleOfElemType), all cc, ii (Prop(InsertLast(cc, ii))));

**end {CircleOfElemType} ;**
2.4. QUEUE

2.4.1. Discussion
We omit lemmas for Queues which are, in fact, almost the same as sequences. See the earlier discussion of Sequences. Examples appear in [Thompson 81].

2.4.2. Specification

\begin{verbatim}
type QueueOfElemType;

needs type ElemType;

declare dummy, q, q1, q2, qq: QueueOfElemType;
declare i, i1, i2, ii: ElemType;

interfaces {Constructors NewQueueOfElemType, q Add i,
{Extenders} Remove(q), Append(q1, q2), que(i)
: QueueOfElemType;

infix Add;

interfaces
  Front(q), Back(q): ElemType;

interfaces
  NormalForm(q), Induction(q), i in q, isNew(q): Boolean;

infix in;

axioms
dummy = dummy = = TRUE,
q Add i = NewQueueOfElemType = = FALSE,
NewQueueOfElemType = q Add i = = FALSE,
q1 Add i1 = q2 Add i2 = = ((q1 = q2) and (i1 = i2));

axioms
Remove(NewQueueOfElemType) = = NewQueueOfElemType,
Remove(q Add i)
  = = if q = NewQueueOfElemType
    then q
    else Remove(q) Add i;
\end{verbatim}
axioms
  Append(q, NewQueueOfElemType) = q,
  Append(q_0' Add i_1) = Append(q, q_1) Add i_1;

axiom que() = NewQueueOfElemType Add i;

axiom Front(q Add i) = if q = NewQueueOfElemType
  then i
  else Front(q);

axiom Back(q Add i) = i;

axioms
  i in NewQueueOfElemType = FALSE,
  i in (q Add i_1) = (i in q or (i = i_1));

axiom isNew(q) = (q = NewQueueOfElemType);

rule lemma Append(NewQueueOfElemType, q) = q;

schema
  NormalForm(q)
  = cases(Prop(NewQueueOfElemType), all qq, ii (Prop(qq Add ii))),

  Induction(q)
  = cases(Prop(NewQueueOfElemType),
           all qq, ii (IH(qq) imp Prop(qq Add ii)));

end (QueueOfElemType);
2.5. SET

2.5.1. Discussion
We have found it most convenient to define 'subset' and 'equal' rather than give them as rewriting rules. Sometimes two equivalent defines are wanted, here in terms of 'subset' and of eqv for qEqual.

Since equality and 'subset' are definitions, we have added rulelemmas for special cases where rewriting appears to be always desirable. Similar reasoning applies to special cases of commutativity.

2.5.2. Specification

**type** SetOfElemType;

**needs** type ElemType;

declare dummy, s, s1, s2, ss: SetOfElemType;
declare i, i1, i2, ii, x: ElemType;

**interfaces** {Constructors} NewSetOfElemType, s add x,
{Extenders} s rem i, s diff s1, s int s1, s union s1, setof(x) :
SetOfElemType;

**infix** add, diff, int, rem, union;

**interfaces**

i in s, isNew(s), s subset s1, Induction(s), NormalForm(s), s1 disjoint s2,
qEqual(s1, s2): Boolean;

**infix** in, subset;

**interface** Size(s): Integer;

**axiom** dummy = dummy = = TRUE;

**axioms**

NewSetOfElemType rem i = = NewSetOfElemType,
(s add x) rem i
= = if x = i
then s rem i
else (s rem i) add x;
axioms
NewSetOfElemType s = NewSetOfElemType,
(s add x) s1 = = if x in s1
   then s1 add x
   else (s s1) add x;

axioms
NewSetOfElemType s1 = NewSetOfElemType,
(s add x) int s1 = = if x in s1
   then (s int s1) add x
   else s1 s1;

axioms
NewSetOfElemType union s1 = = s1,
(s add x) union s1 = = union (s union s1) add x;

axioms
x in NewSetOfElemType = = FALSE,
i in (s add x) = = (i = x) or i in s;

axiom isNew s = = (s = NewSetOfElemType);

axiom setof x = = NewSetOfElemType add x;

axioms
Size(NewSetOfElemType) = = 0,
Size(s add x) = = if x in s
   then Size(s)
   else Size(s) + 1;

rulelemmas
NewSetOfElemType = s add i = = FALSE,
s add i = NewSetOfElemType = = FALSE;

rulelemma s diff NewSetOfElemType = = s;

rulelemma s int NewSetOfElemType = = NewSetOfElemType;

rulelemma s union NewSetOfElemType = = s;

rulelemmas
NewSetOfElemType subset s = = TRUE,
s subset NewSetOfElemType = = (s = NewSetOfElemType);
rulelemma NewSetOfElemType disjoint s = = TRUE;

define

s1 = s2 = (s1 subset s2 and s2 subset s1),

s subset s1
 = = all x (x in s imp x in s1),

s1 disjoint s2
 = = all x (x in s1 imp ~ (x in s2)),

qEqual(s1, s2)
 = = all x (x in s1 eqv x in s2);

schema

Induction(s)
 = = cases(Prop(NewSetOfElemType), all ss, ii (IH(ss) imp Prop(ss add ii))),

NormalForm(s) = = cases(Prop(NewSetOfElemType), all ss, ii (Prop(ss add ii)));

end {SetOfElemType} ;
2.5.3. Lemmas

in Lemmas

**Assume** inRemoval, x in (A rem y) eqv x in A and (x ≠ y);

**Assume** inDifference, x in (A diff B) eqv x in A and ¬(x in B);

**Assume** inIntersection, x in (A int B) eqv x in A and x in B;

**Assume** inUnion, x in (A union B) eqv x in A or x in B;

Size Lemmas

**Assume** SizeNonNeg, Size(A) ≥ 0;

**Assume** SizeDifference, Size(A diff B) = Size(A) - Size(A int B);

**Assume** SizeUnion, Size(A union B) = Size(A) + Size(B) - Size(A int B);

Intersection Lemmas

**Assume** IntersectionAssoc, A int (B int C) = (A int B) int C;

**Assume** IntCommutes, A int B = B int A;

**Assume** UnionOverInt, A union (B int C) = (A union B) int (A union C);

Union Lemmas

**Assume** UnionAssoc, A union (B union C) = (A union B) union C;

**Assume** IntOverUnion, A int (B union C) = (A int B) union (A int C);

**Assume** unionadd, (A union B) add x = A union (B add x);

**Assume** unionremadd, x in A

    imp (B union (A rem x)) add x = B union A;

Basic Equality Lemma

**Assume** qEqual, A = B eqv qEqual(A, B);

Removal Lemmas

**Assume** remadd, x in A imp (A rem x) add x = A;
\textit{subset Lemmas}

\texttt{assume} subset\textunderscore intersection, \(A \subset \text{subset} (B \text{int} \ C) \subset (B \text{int} \ C)\);

\texttt{assume} subset\textunderscore union, \(A \subset \text{subset} (A \text{union} \ C) \subset (B \text{union} \ C)\);

\texttt{assume} subset\textunderscore difference, \(A \subset \text{subset} (A \text{diff} \ C) \subset (B \text{diff} \ C)\);

\texttt{assume} subset\textunderscore Transitivity, \(A \subset \text{subset} \text{and} \text{B subset} \text{subset} \text{C imp} A \subset \text{subset} \text{C}\);

\texttt{assume} subset\textunderscore union, \(A \subset \text{subset} (A \text{union} \ C) \subset (B \text{union} \ C)\);

\texttt{assume} subset\textunderscore difference, \(A \subset \text{subset} (A \text{diff} \ C) \subset (B \text{diff} \ C)\);

\texttt{assume} subset\textunderscore Transitivity, \(A \subset \text{subset} \text{B and} \text{B subset} \text{subset} \text{C imp} A \subset \text{subset} \text{C}\);

\texttt{assume} subset\textunderscore Union\textunderscore Equal, \(A \subset \text{subset} (A \text{union} \ C) = B\);

\texttt{assume} subset\textunderscore union, \(A \subset \text{subset} (A \text{union} \ C)\);

\texttt{assume} subset\textunderscore Remove, \(A \subset \text{subset} (B \text{rem} \ x) \equiv A \subset \text{subset} \text{and} \sim(x \text{in} \ A)\);

\texttt{assume} subset\textunderscore Add, \(A \subset \text{subset} (B \text{add} \ x)\);

\textit{Difference Lemmas}

\texttt{assume} difference\textunderscore intersection, \(A \text{diff} B = A \text{diff} (A \text{int} B)\);

\texttt{assume} Difference\textunderscore Remove, \(A \text{rem} x = A \text{diff} \text{setof}(x)\);

\texttt{assume} sub\textunderscore Diff, \(A \subset \text{subset} (B \text{diff} \ C) \equiv A \subset \text{subset} \text{B and} \text{A disjoint} \text{C}\);

\textit{disjoint Lemmas}

\texttt{assume} disjoint\textunderscore Difference, \(\text{disjoint} \text{B imp} A \text{diff} B = A\);

\texttt{assume} disjoint\textunderscore Intersection, \(\text{disjoint} \text{B imp} \text{isNew} (A \text{int} B)\);
2.6. MAPPING

2.6.1. Discussion
This data type is used to mimic a vector. Domain and Range are minimally specified types. Variations are maintained where Domain is Integer and Range. See the programs' uses of arrays as implementations of sequences. Also see type Sequence for the representation functions.

2.6.2. Specification

type MappingFromDomainToRange;

needs types Domain, Range;

declare a, b, dummy: MappingFromDomainToRange;
declare i, j, k: Domain;
declare x, y, z: Range;

interfaces {Constructors}
   NewMappingFromDomainToRange, assn(a, i, x): MappingFromDomainToRange;

interface Size(a): Integer;

interface a sub i: Range;

infix sub;

interfaces
   isNew(a), isdefinedfor(a, i), Eq Undefined(a, b), EqualForDefined(a,b),
   Induction(a), NormalForm(a): Boolean;

axiom dummy = dummy = = TRUE;

axioms
   Size(NewMappingFromDomainToRange) = = 0,
   Size(assn(a, i, x))
   = = if isdefinedfor(a, i)
       then Size(a)
       else Size(a) + 1;

axiom assn(a, i, x) sub j
   = = if i = j
then x
else a sub j;

**axiom** isNew(a) = (a = NewMappingFromDomainToRange);

**axioms**

isNew(NewMappingFromDomainToRange, i) = FALSE,
isNew(assn(a, i, x), i)
= ((i = j) orisNew(a, i));

**define**

a = b = (EqualDefined(a, b) and EqualForDefined(a, b)),

EqualDefined(a, b) *
= all i (isNew(a, i) eqv isNew(b, i)),

EqualForDefined(a, b)
= all i (isNew(a, i)
   imp a sub i = b sub i);

**schema**

Induction(b)
= cases(Prop(NewMappingFromDomainToRange),
   all a, i, x (IH(a) imp Prop(assn(a, i, x)))),

NormalForm(b)
= cases(Prop(NewMappingFromDomainToRange),
   all a, i, x (Prop(assn(a, i, x))));

end {MappingFromDomainToRange} ;
2.7. GRAPH

2.7.1. Discussion

The Graph type is one of the larger and more complex types in the Type Library, in part because it needs both the Set and Sequence types for various functions: it is not quite as "primitive" a type as Set or Sequence. Graph is not a completely general version of directed graphs: such notions as deleting edges from a graph or even equality of two graphs are not included, making this type (without extensions) not very suitable for general graph theory proofs. Graph is designed for "modelling" and verifying algorithms, such as scheduling or network-related algorithms, that use directed graphs explicitly or implicitly: algorithms that build up, over time, graphs recording dependency relationships, say, or the history of a multi-process operation.¹

The base type is GraphOfElemType, in which nodes of the graph are of type ElemType and thus have no internal structure. As usual, most uses of Graph will require instantiating the base type to GraphOfYourNodeType.

Reachability is an important concept in reasoning about (directed) graphs; there are two kinds of reachability predicates in Graph. The path predicate simply indicates whether or not a path exists from one node to another.² Frequently it is necessary, in proving a graph theorem, to explicitly reason about a particular path between two nodes -- in Graph this is done with "path sequences" (of type Sequence), sequences of nodes linked together in the graph. The predicates included are:

- \texttt{pathSeq}(g, ps) \quad \text{Sequence } ps \text{ is a path in graph } g \\
- \texttt{onPathSeq}(g, ps, n) \quad \text{node } n \text{ in contained in path } ps \\
- \texttt{pathSeqFrom}(g, ps, a, d) \quad \text{path } ps \text{ runs from nodes } a \text{ (ancestor) to } d \text{ (descendant)} \\
- \texttt{somePSF}(g, a, d) \quad \text{some path runs from } a \text{ to } d \text{ -- this is equivalent to } \texttt{path}(g,a,d)^3

¹The Graph type was originally developed in trying to verify a network operating system file-consistency algorithm.
²Note that in Graph, all paths are of length 1 or more; thus \texttt{path}(g,n,n) is false unless the edge \texttt{<n,n>} has been explicitly added to graph g, or some cycle of length \textgreater{} 2 from n to n is present in g.
³See lemma \texttt{pathEqvSomePSF}.
2.7.2. Specification

type GraphOfElemType;

needs types ElemType, SetOfElemType, SequenceOfElemType, Integer;

declare dummy, g, g', gg: GraphOfElemType;
declare a, c, c', d, n, n1, n2, p, p': ElemType;
declare nodeset: SetOfElemType;
declare ps, ps1, ps2: SequenceOfElemType;

interfaces

{Constructors} emptyG, addedge(g, p, c),
{Extenders} addEdgesToNode(g, nodeset, c),
            addEdgesFromNode(g, nodeset, p): GraphOfElemType;

interfaces

n in g, edgen(g, p, c), leaf(g, n), noSons(g, n),
path(g, a, d), dpath(g, a, d), pathSeq(g, ps),
pathSeqFrom(g, ps, a, d), onPathSeq(g, ps, n),
somePSF(g, a, d), g' extensionOf g, g' disjExtOf g, isEmptyG(g),
Induction(g), NormalForm(g): Boolean;

infix disjExtOf, extensionOf, in;

interfaces nodes(g), leavesOf(g), sonsOf(g, n), parentsOf(g, n): SetOfElemType;

interfaces

nodeCount(g), addedgeCount(g): Integer;

axioms

dummy = dummy = = TRUE,
emptyG = addedge(g, p, c) = = FALSE,
addedge(g, p, c) = emptyG = = FALSE;

axioms

addEdgesToNode(g, NewSetOfElemType, c) = = g,
addEdgesToNode(g, nodeset add p, c)
= = addedge(addEdgesToNode(g, nodeset, c), p, c);

axioms

addEdgesFromNode(g, NewSetOfElemType, p) = = g,
addEdgesFromNode(g, nodeset add c, p)
= = addedge(addEdgesFromNode(g, nodeset, p), p, c);
axioms
n in emptyG = = FALSE,
n in addedge(g, p, c)
= = ((n = p) or (n = c) or n in g);

axioms
edgeIn(emptyG, p, c) = = FALSE,
edgeIn(addedge(g, p, c), p', c')
= = if p = p'
then (c = c') or edgeIn(g, p', c')
else edgeIn(g, p', c');

axioms
leaf(emptyG, n) = = FALSE,
leaf(addedge(g, p, c), n)
= = ((n = c) or n in g) and noSons(g, n)
and n ~ = p);

axioms
noSons(emptyG, n) = = TRUE,
noSons(addedge(g, p, c), n)
= = (noSons(g, n) and (n ~ = p));

axioms
pathSeq(g, NewSequenceOfElemType) = = FALSE,
pathSeq(g, ps app n)
= = if ps = NewSequenceOfElemType
then n in g
else pathSeq(g, ps)
and edgeIn(g, Last(ps), n);

axioms
emptyG extensionOf g = = FALSE,
addedge(g', n1, n2) extensionOf g
= = ((g' = g) or g' extensionOf g);

axioms
emptyG disjExtOf g = = FALSE,
addedge(g', p, c) disjExtOf g
= = (~c in g) and ((g' = g) or g' disjExtOf g));

axioms
nodes(emptyG) = = NewSetOfElemType,
nodes(addedge(g, p, c)) = = (nodes(g) add p) add c;
axioms
leavesOf(emptyG) = NewSetOfElemType,
leavesOf(addedge(g, p, c))
  = if leaf(g, p)
    then if noSons(g, c) and (c ≠ p)
    then (leavesOf(g) rem p) add c
    else leavesOf(g) rem p
    else if noSons(g, c) and (c ≠ p)
    then leavesOf(g) add c
    else leavesOf(g);

axioms
sonsOf(emptyG, n) = NewSetOfElemType,
sonsOf(addedge(g, p, e), a)
  = if a = p
    then sonsOf(g, a) add c
    else sonsOf(g, a);

axioms
parentsOf(emptyG, n) = NewSetOfElemType,
parentsOf(addedge(g, p, e), n)
  = if e = n
    then parentsOf(g, n) add p
    else parentsOf(g, n);

axioms
nodeCount(emptyG) = 0,
nodeCount(addedge(g, p, c))
  = if p in g
    then if c in g
      then nodeCount(g)
      else nodeCount(g) + 1
    else if c in g
      then nodeCount(g) + 1
      else nodeCount(g) + 2;

axioms
addedgeCount(emptyG) = 0,
addedgeCount(addedge(g, p, c)) = addedgeCount(g) + 1;

rulelemma pathSeq(emptyG, ps) = FALSE;

rulelemma isEmptyG(g) = (g = emptyG);

define
  path(emptyG, a, d) = FALSE,
  path(addedge(g, p, c), a, d)
= (path(g, a, d)  
  or  dnpath(g, a, p)  
  and dnpath(g, c, d)).  

dnpath(g, a, d) = = ((a = d) or path(g, a, d)).  

pathSeqFrom(g, ps, a, d)  
= = (pathSeq(g, ps) and (2 ≤ Length(ps))  
and First(ps) = a  
and Last(ps) = d),  

onPathSeq(g, ps, n) = = (pathSeq(g, ps) and n in ps),  

somePSF(g, a, d)  
= = some ps' (pathSeqFrom(g, ps', a, d));  

schema  
Induction(g)  
= = cases(Prop(emptyG),  
  all g', p, c (IH(g')  
  imp Prop(addedge(g', p, c)))).  

NormalForm(g)  
= = cases(Prop(emptyG), all g', p, c (Prop(addedge(g',  
  p, c))));  

end (GraphOfElementType) ;
2.7.3. Lemmas

**path Lemmas**

**Assume** transpath, \( \text{path}(g, a, d) \)
\( \text{and } \text{path}(g, d, n) \)
\( \text{imp } \text{path}(g, a, n); \)

**Assume** pathInExtension, \( g' \text{ extensionOf } g \)
\( \text{and } \text{path}(g, n1, n2) \)
\( \text{imp } \text{path}(g', n1, n2); \)

**Assume** pathInDisjExt, \( n1 \text{ in } g \)
\( \text{and } n2 \text{ in } g \)
\( \text{and } g' \text{ disjExtOf } g \)
\( \text{and } \text{path}(g', n1, n2) \)
\( \text{imp } \text{path}(g, n1, n2); \)

**Assume** pathEndptsInGraph, \( \text{path}(g, a, d) \)
\( \text{imp } d \text{ in } g \text{ and } a \text{ in } g; \)

**Assume** sourceDEpathInSub, \( d \text{ in } g' \)
\( \text{and } g' \text{ disjExtOf } g' \)
\( \text{and } \text{path}(g, a, d) \)
\( \text{imp } a \text{ in } g'; \)

**edgeIn Lemmas**

**Assume** edgelnImpPath, \( \text{edgeln}(g, a, d) \)
\( \text{imp } \text{path}(g, a, d); \)

**Assume** pathEdgeln, \( \text{path}(g, a, d) \)
\( \text{and } \text{edgeln}(g, d, n1) \)
\( \text{imp } \text{path}(g, a, n1); \)

**extensionOf Lemmas**

**Assume** extensionOfIrreflexive, \( \neg (g \text{ extensionOf } g) \);

**Assume** extensionOfAntiSymmetric, \( \neg (g' \text{ extensionOf } g \text{ and } g \text{ extensionOf } g') \);

**Assume** addedgeCountExt, \( g \text{ extensionOf } g' \text{ imp } \text{addedgeCount}(g') < \text{addedgeCount}(g) \);
leaf Lemmas

**assume leafEqv**, leaf(g, n)
  eqv n in g and noSons(g, n);

**assume sonsOfLeaf**, leaf(g, p)
  eqv sonsOf(g, p) = NewSetOfElemType
  and p in g;

**assume noSonsEqv**, noSons(g, n) eqv sonsOf(g, n) = NewSetOfElemType;

parentsOf, sonsOf Lemmas

**assume parentsOfDisconnNode**, ~(n in g)
  imp parentsOf(g, n) = NewSetOfElemType;

**assume parentsOfAddToNode**, parentsOf(addEdgesToNode(g, nodeset, c), c)
  = parentsOf(g, c) union nodeset;

**assume sonsOfAddFromNode**, sonsOf(addEdgesFromNode(g, nodeset, p), p)
  = sonsOf(g, p) union nodeset;

pathSeq Lemmas

**assume pathSeqInExtension**, pathSeq(g, ps) and g' extensionOf g
  imp pathSeq(g', ps);

**assume pathSeqLLast**, pathSeq(g, ps) and (2 < Length(ps))
  imp pathSeq(g, LessLast(ps));

**assume pathSeqJoin**, pathSeq(g, ps)
  and pathSeq(g, ps2)
  and First(ps2) = Last(ps)
  imp pathSeq(g, LessLast(ps) join ps2);

somePSF (some pathSeqFrom) Lemmas

**assume pathEqvSomePSF**, path(g, a, d)
  eqv somePSF(g, a, d);

**assume transSomePSF**, somePSF(g, a, d)
and somePSF(g, d, n1)
imp somePSF(g, a, n1);

\textbf{assume} somePSF\textit{basis}, somePSF(addedge(g, a, d), a, d);

\textbf{assume} somePSF\textit{inExtension}, somePSF(g, a, d)
and g' extensionOf g
imp somePSF(g', a, d);
2.8. BINARYTREE

2.8.1. Discussion
A variation of this data type was used in the Josephus Circle problem [Gerhart 81] and in the Delta Experiment [Gerhart 79], but its theory has not been well explored with **AFFIRM**

2.8.2. Axioms

2.8.2.1. Basic Axioms

```plaintext
declare bt, bt1, bt11, bt12, bt2, bt21, bt22, dummy: BinaryTreeOfElemType;
declare d, d1, d2: ElemType;

interfaces {Constructors} NewBinaryTreeOfElemType, Tree(bt1, bt2, d), Leaf(d),
{Extenders} left(bt), right(bt): BinaryTreeOfElemType;

interface datum(bt): ElemType;

interfaces
  isleaf(bt), NormalForm(bt), isNew(bt), Induction(bt): Boolean;

interfaces
  Depth(bt), Size(bt): Integer;

axioms
  dummy = dummy = = TRUE,
  NewBinaryTreeOfElemType = Tree(bt1, bt2, d) = = FALSE,
  Tree(bt1, bt2, d) = NewBinaryTreeOfElemType = = FALSE,
  Tree(bt11, bt12, d1) = Tree(bt21, bt22, d2)
    = = ((d1 = d2) and (bt11 = bt21) and (bt12 = bt22));

axiom left(Tree(bt1, bt2, d)) = = bt1;

axiom right(Tree(bt1, bt2, d)) = = bt2;

axioms
  datum(Leaf(d)) = = d,
  datum(Tree(bt1, bt2, d)) = = d;

axioms
  isleaf(NewBinaryTreeOfElemType) = = FALSE,
  isleaf(Leaf(d)) = = TRUE,
```
isleaf(Tree(bt1, bt2, d)) = = FALSE;

axiom isNew(bt) = = (bt = NewBinaryTreeOfElemType);

axioms
Depth(NewBinaryTreeOfElemType) = = 0,
Depth(Leaf(d)) = = 1,
Depth(Tree(bt1, bt2, d))
  = = (if Depth(bt1) < Depth(bt2)
        then Depth(bt2)
        else Depth(bt1)) + 1;

axioms
Size(NewBinaryTreeOfElemType) = = 0,
Size(Leaf(d)) = = 1,
Size(Tree(bt1, bt2, d)) = = Size(bt1) + Size(bt2) + 1;

schema
NormalForm(bt)
  = = cases(Prop(NewBinaryTreeOfElemType),
            all d (Prop(Leaf(d))),
            all bt1, bt2, d (Prop(Tree(bt1, bt2, d)))).

Induction(bt)
  = = cases(Prop(NewBinaryTreeOfElemType),
            all d (Prop(Leaf(d))),
            all d, bt1, bt2
            ( IH(bt1) and IH(bt2)
              imp Prop(Tree(bt1, bt2, d))));

end {BinaryTreeOfElemType} ;
2.8.2.2. Additional Axioms

type Basis;

needs types BinaryTreeOfElemType, SequenceOfElemType, ElemType;

declare bt, bt1, bt2: BinaryTreeOfElemType;
declare d: ElemType;

interfaces
PreOrder(bt), PostOrder(bt), InOrder(bt): SequenceOfElemType;

axioms
PreOrder(NewBinaryTreeOfElemType) = = NewSequenceOfElemType,
PreOrder(Leaf(d)) = = NewSequenceOfElemType apr d,
PreOrder(Tree(bt1, bt2, d)) = = d aPl (PreOrder(bt1) join PreOrder(bt2));

axioms
PostOrder(NewBinaryTreeOfElemType) = = NewSequenceOfElemType,
PostOrder(Leaf(d)) = = NewSequenceOfElemType apr d,
PostOrder(Tree(bt1, bt2, d))
= = (PostOrder(bt1) join PostOrder(bt2)) apr d;

axioms
InOrder(NewBinaryTreeOfElemType) = = NewSequenceOfElemType,
InOrder(Leaf(d)) = = NewSequenceOfElemType apr d,
InOrder(Tree(bt1, bt2, d)) = = InOrder(bt1) join (d aPl InOrder(bt2));
2.9. INTEGER

2.9.1. Discussion
This is far from the complete specification of Integer; numerous simplifications are automatically applied and the Normint algorithm may be user-invoked. See the Reference Manual and Users' Guide for more extensive discussion of the Integer type. The lemmas in the following section are occasionally required to make the rest of the Integer machinery work. AddSwitch rearranges integer expressions. LEAdd expresses summation over inequalities.

2.9.2. Specification

type Integer;

declare i1, i2, i3, ii: Integer;

interfaces
   i1 + i2, i1-i2, i1*i2, max(i1, i2), min(i1, i2), i1/i2,
   EXPT(i1, i2), i1 mod i2, 1/i1, i1 div i2, -i1: Integer;

interfaces
   i1>i2, i1<i2, i1 <= i2, i1 >= i2, induction(i1): Boolean;

axiom i1 = i1 = TRUE;

axiom max(i1, i2) = = if i1 <= i2
   then i2
   else i1;

axiom min(i1, i2) = = if i1 <= i2
   then i1
   else i2;

schema induction(i1)
   = = cases(Prop(0),
      all ii ( (ii <= 0) and IH(ii)
            imp Prop(ii-1)),
      all ii ( (0 <= ii) and IH(ii)
            imp Prop(ii + 1)));

end {Integer} ;
2.9.3. Lemmas

```plaintext
declare k1,k2,k3,k4:Integer;

assume AddSwitch, k1 + k2 = k3 eqv k1 = k3·k2 and k2 = k3·k1;

assume LEAdd, k1 < k2 and k3 < k4 imp k1 + k3 < k2 + k4;
```
3. LESSON

3.1. Discussion
The lesson is accessed by the command "read <pvlibrary>lesson.setup", which loads the needed types and notations and then reads the "theorems" which the learner is to prove. These exercises are somewhat repetitive, but cover the basic set of commands and provide a good feeling for AFFIRM's data type induction capability. The user is reminded that not all the "theorems" may actually be such.

3.2. Notation

```plaintext
type LessonNotation;

needs types SequenceOfElemType, ElemType;

declare dummy: LessonNotation;
declare s, s1, s2: SequenceOfElemType;
declare i, j, k: ElemType;

interfaces
  deleteNonp(s), dedup(s), reverse(s): SequenceOfElemType;

interfaces
  nodups(s), s1 subseq s2, p(i), allp(s): Boolean;

infix subseq;

axiom dummy = dummy = TRUE;

axioms
  deleteNonp(NewSequenceOfElemType) = NewSequenceOfElemType,
  deleteNonp(s apr i)
    = = if p(i)
      then deleteNonp(s) apr i
    else deleteNonp(s);

axioms
  dedup(NewSequenceOfElemType) = NewSequenceOfElemType,
  dedup(s apr i)
    = = if i in s
      then dedup(s)
```

else dedup(s apr i);

axioms
reverse(NewSequenceOfElemType) = NewSequenceOfElemType,
reverse(s apr i) = i apl reverse(s);

axioms
nodups(s apr i) = (nodups(s) and ~(i in s)),
nodups(NewSequenceOfElemType) = TRUE;

axioms
s subseq NewSequenceOfElemType = (s = NewSequenceOfElemType),
s1 subseq (s apr i)
= (s1 = NewSequenceOfElemType or s1 subseq s
or LessLast(s1) subseq s and (Last(s1) = i));

axioms
allp(NewSequenceOfElemType) = TRUE,
alip(s apr i) = (p(i) and allp(s));

end {LessonNotation} ;
3.3. "Theorems"

The following propositions may not all be theorems; that's part of what the lesson is teaching.

```latex
theorem AllpDeNonp, allp(deleteNonp(s));
theorem NodupsDedup, nodups(dedup(s));
theorem DeNonpSubseq, deleteNonp(s) subseq s;
theorem DedupSubseq, dedup(s) subseq s;
theorem AllpDedup, allp(s) imp allp(dedup(s));
theorem NodupsDeNonp, nodups(s) imp nodups(deleteNonp(s));
theorem DeNonpJoin, deleteNonp(s1 join s2) = deleteNonp(s1) join deleteNonp(s2);
theorem AllpJoin, allp(s1 join s2) eqv allp(s1) and allp(s2);
note Notice the difference between the two theorems
  AllpJoin and AllpJoinBad;
theorem AllpJoinBad, allp(s1 join s2) = allp(s1) and allp(s2);
theorem DedupDeNonp, dedup(deleteNonp(s)) subseq s and
  allp(dedup(deleteNonp(s))) and
  nodups(dedup(deleteNonp(s)));
theorem AllpReverse, allp(s) eqv allp(reverse(s));
theorem ReverseDedup, reverse(dedup(s)) = dedup(reverse(s));
theorem NodupsReverse, nodups(s) eqv nodups(reverse(s));
```
Appendix I
PROGRAMS

1.1. Remove Blanks
This program removes all extra blanks, reducing a string of blanks to a single one, as in minimizing the space between a stream of words in a text. It is first proved that the program computes a function, Rembl, defined by axioms. Later various properties of Rembl are proved to establish that it does do something like removing blanks.

PROGRAM

program RemoveBlanks;
procedure RB(input:SequenceOfElemType; var output:SequenceOfElemType);
pre TRUE;
post output = Rembl(input');
var LastChar,ThisChar ElemType;
begin
if isNew(input) then output = NewSequenceOfElemType
else
begin LastChar,input,output,input.output:
= First(input),LessFirst(input),seq(First(input));
maintain RemblInvariant(input',input,output,LastChar)
while ~isNew(input) do
begin
ThisChar,input:: First(input), LessFirst(input);
if ~(isBlank(LastChar) and isBlank(ThisChar)) then
output,LastChar: = output apr ThisChar,ThisChar
end;
end;
end;

CONTEXT

declare input, input', input1, output, output', output1, output2, Rs, s, s',
   s1, s1', s1'', s2, s2', s2'', s3, s3', s3'', ss, ss', ss'', w1, w1', w2,'w2': SequenceOfElemType;
declare Blank, LastChar, LastChar1, ThisChar, ThisChar1, i, i', ii, ii', j, j': ElemType;
declare k: Integer;

interface Rembl(input): SequenceOfElemType;

interfaces
   BL, pth(s, k): ElemType;
interfaces

RemblInvariant(input', input, output, LastChar), isBlank(i),

s1 subseq s2, NoAdjacentBlanks(s), WordsIn(w1, w2, s), NoBlanks(s),

Nab(s), MatchBlanks(s1, s2), AllBlanks(s), MatchEndBlank(s1, s2),

MatchEndNonBlanks(s1, s2): Boolean;

 infix subseq;

interfaces

RemoveBlanks, RB(input, output): ProcedureCall;

Rembl is the function, defined axiomatically, for removing blanks. It is proved that the
program computes Rembl for its input.

axioms

Rembl(NewSequenceOfElemType) = = NewSequenceOfElemType,

Rembl(s apr i) = = if s = = NewSequenceOfElemType

and isBlank(Last(s))

and isBlank(i)

then Rembl(s)

else Rembl(s) apr i;

The loop invariant for the program, RemblInvariant describes how the loop is computing
Rembl. LessLast(output) is the initial part of Rembl(input') and then Rembl remains to be
computed for input relative to Last(output) being a blank or not. The program variable
LastChar is maintained to be Last(output).

axiom RemblInvariant(input', input, output, LastChar)

= = ( output = = NewSequenceOfElemType

and Rembl(input')

= LessLast(output) join Rembl(Last(output) apr input)

and LastChar = Last(output));

subseq is a standard function for sequences.

axioms

s subseq NewSequenceOfElemType = = (s = NewSequenceOfElemType),

s1 subseq (s apr i)

= = ( s1 = NewSequenceOfElemType) or s1 subseq s

or LessLast(s1) subseq s and (Last(s1) = i));

axioms

NoBlanks(NewSequenceOfElemType) = = TRUE,

NoBlanks(s apr i) = = (NoBlanks(s) and ~isBlank(i));
Nab is an axiomatic version of NoAdjacent Blanks, used to make the proof easier.

**axioms**

\[
\begin{align*}
\text{Nab} & : \text{NewSequenceOfElemType} = \text{TRUE}, \\
\text{Nab}(s \text{ apr } i) & = (s = \text{NewSequenceOfElemType} \\
& \quad \text{or } \text{Nab}(s) \\
& \quad \text{and isBlank}(i) \implies \neg \text{isBlank(Last}(s))) \\
\end{align*}
\]

**axioms**

\[
\begin{align*}
\text{AllBlanks} & : \text{NewSequenceOfElemType} = \text{TRUE}, \\
\text{AllBlanks}(s \text{ apr } i) & = (\text{isBlank}(i) \text{ and } \text{AllBlanks}(s)); \\
\end{align*}
\]

**define**

\[
\begin{align*}
\text{NoAdjacentBlanks}(s) & : = \text{all } s_1, s_2, i, j \\
& \quad (s_1 \text{ join } i \text{ apr } (j \text{ apr } s_2)) \\
& \quad = s \\
& \quad \text{and isBlank}(i) \\
& \quad \text{imp } \neg \text{isBlank}(j), \\
\end{align*}
\]

*MatchBlanks is used to express one of the important properties of Rembl, that except for other blanks where Rembl(s) has a blank, Rembl(s) and s are the same.*

\[
\begin{align*}
\text{MatchBlanks}(s_1, s_2) & = \text{if } s_1 = \text{NewSequenceOfElemType} \\
& \quad \text{then } s_2 = \text{NewSequenceOfElemType} \\
& \quad \text{else if isBlank(Last}(s_1)) \\
& \quad \quad \text{then MatchEndBlank}(s_1, s_2) \\
& \quad \quad \text{else MatchEndNonBlanks}(s_1, s_2), \\
\end{align*}
\]

\[
\begin{align*}
\text{MatchEndBlank}(s_1, s_2) & = \text{some } w_1, w_2, s_1', s_2' \\
& \quad (\text{Last}(s_1) = s_1' \text{ join } w_1 \\
& \quad \quad \text{and } s_2 = (s_2' \text{ join } w_1) \text{ join } w_2 \\
& \quad \quad \text{and NoBlanks}(w_1) \\
& \quad \quad \text{and AllBlanks}(w_2) \\
& \quad \quad \text{and MatchBlanks}(s_1', s_2')). \\
\end{align*}
\]

\[
\begin{align*}
\text{MatchEndNonBlanks}(s_1, s_2) & = \text{some } w_1, s_1', s_2' \\
& \quad (s_1 = s_1' \text{ join } w_1 \\
& \quad \quad \text{and } s_2 = s_2' \text{ join } w_1 \\
\end{align*}
\]
and NoBlanks(w1)
and MatchBlanks(s1', s2');

LEMMAS

Verification Conditions

VC for empty input, bypassing the loop.

**assume** RB # 1, isNew(input) imp NewSequenceOfElemType = Rembl(input);

VC for exiting the loop.

**assume** RB # 2, ~isNew(input)
and RemblInvariant(input,
   input1, output2, LastChar1)
and isNew(input1)
imp output2 = Rembl(input);

VC for entering the loop.

**assume** RB # 3, ~isNew(input)
imp RemblInvariant(input,
   LessFirst(input),
   seq(First(input)), First(input));

VC for traversing the loop after just reading a non-blank or with a blank not preceded by a blank.

**assume** RB # 4, RemblInvariant(input',
   input, output, LastChar)
and ~isNew(input)
and ~isBlank(LastChar) and isBlank(First(input)))
imp RemblInvariant(input',
   LessFirst(input),
   output apr First(input), First(input));

VC for traversing the loop with two successive blanks.

**assume** RB # 5, RemblInvariant(input',
   input, output, LastChar)
and ~isNew(input)
and isBlank(LastChar)
and isBlank(First(input))
imp RemblInvariant(input',
   LessFirst(input), output, LastChar);
Computes Lemma to link the procedure RB with any calls on it.

**assume** computesRB, computes(RB(input, output), result(output1))
  imp output1 = Rembl(input);

**assume** RB, verification(RB);

Properties of Rembl used as lemmas for VC's and as evidence of Rembl’s correctness.

The following property of Rembl says that a string ending with a blank leaves a blank when processed by Rembl. A corresponding property could be proved for the beginning of a string.

**assume** EndsWithBlank, ~isNew(s) and isBlank(Last(s))
  imp isBlank(Last(Rembl(s)));

This property says that Rembl(s) and s match characters except for where Rembl(s) has a blank in which case s may have extra blanks.

**assume** MatchBlanks, MatchBlanks(Rembl(s), s);

Another property of Rembl states that there are no adjacent blanks in Rembl(s).

**assume** NoAdjacentBlanks, NoAdjacentBlanks(Rembl(s));

A little property of sequences needed for the above lemmas.

**assume** BlankNew, isNew(s1 join s2)
  eqv isNew(s1) and isNew(s2);

A lemma proved to make NoAdjacentBlanks easier to prove. See the proofs for further explanation.

**assume** NabEqv, Nab(s) eqv NoAdjacentBlanks(s);

A weaker property than MatchBlanks, this says that at least Rembl(s) didn't add any characters because it is a subsequence of s.

**assume** NoWordsAdded, Rembl(s) subseq s;

A little property used to cover the starting case.

**assume** FirstNonBlank, ~isBlank(i)
  imp Rembl(i apl s)
  = i apl Rembl(s);

Here's a nice property that no non-blank character gets removed by Rembl.

**assume** NonBlank, ~isBlank(i)
  imp Rembl((s1 apr i) join s2)
  = (Rembl(s1) apr i) join Rembl(s2);

The Split lemmas are assumed, having been proved (trivially) in the type library.

**assume** FirstSplit, isNew(s)
  or First(s) apl LessFirst(s) = s;
assume LastSplit, isNew(s)
or LessLast(s) apr Last(s) = s;

A big property about any two adjacent characters in s and how they come out in Rembl(s).

assume Rembl2adjacent, Rembl(((s1 apr i) apr j) join s2)
= if isBlank(j)
  then if isBlank(i)
    then Rembl((s1 apr i) join s2)
    else Rembl(s1 apr i)
        join Rembl(j apr s2)
  else Rembl(s1 apr i) apr j
      join Rembl(s2);
I.2. Remove Duplicates

This program removes duplicate elements from a sequence by iterating left to right through the elements, omitting any element which has occurred previously in the sequence. The concrete data structure is an array with base index 1.

**PROGRAM**

```plaintext
{This program REMOVES DUPLICATE ELEMENTS from the array V[1..vn] producing the array W[1..wn]}
procedure remdup(V:MappingFromIntegerToElemType; vn:integer;
    var W:MappingFromIntegerToElemType;
    var wn:integer);
pre vn >= 0;
pren(W,wn) = dedup(rep(V,vn));
var vp,vc:integer;
begien
    vp = 1; wn = 0;
    {This loop has done dedup(rep(V[1..vp-1])}
    maintain dedupInvariant(V,vp,vn,W,wn)
    while vp <= vn do
        begin
            vc = 1;
            {This loop is doing V[vp] in V[1..vp-1]}
            maintain dedupInvariant(V,vp,vn,W,wn) and invariant(V,vc,vp,vn)
            while V sub vc -= V sub vp do
                vc = vc + 1;
            if vc = vp then
                begin
                    wn = wn + 1;
                    W = assn(W, wn, V sub vp)
                end;
            vp = vp + 1
        end;
end;
```

**CONTEXT**

```plaintext
type remdupContext;
needs types MappingFromIntegerToElemType, SequenceOfElemType;
\* declare dummy: remdupContext;\*
\* declare i, j, k, vc, vn, vp, wn: Integer;\*
\* declare a, V, W: MappingFromIntegerToElemType;\*
interface rep(a, k): SequenceOfElemType;
```
interfaces
  bounds(i, j, k), dedupInvariant(V, vp, vn, W, wn),
  invariant(V, ve, vp, vn): Boolean;

axiom dummy = dummy = TRUE;

define
  rep(a, k)
  = if k <= 0
    then NewSequenceOfElemType
    else rep(a, k-1) apr (a sub k),

  bounds(i, j, k) = = ((i <= j) and (j <= k)),

  dedupInvariant(V, vp, vn, W, wn)
  = = ( bounds(1, vp, vn + 1) and bounds(0, wn, vp) and rep(W, wn) = dedup(rep(V, vp-1))),

  invariant(V, vc, vp, vn)
  = = ( bounds(1, vc, vp) and bounds(vc, vp, vn) and ~((V sub vc-1) in rep(V, vc-1)));

end {remdupContext} ;

LEMMAS

assume remdup #6, dedupInvariant(V, vp, vn, W, wn)
  and vp <= vn
  and dedupInvariant(V, vp, vn, W, wn)
  and invariant(V, vc3, vp, vn)
  and V sub vc3 = V sub vp
  and vc3 ~ = vp
  imp dedupInvariant(V, vp + 1, vn, W, wn);

assume remdup #5, dedupInvariant(V, vp, vn, W, wn)
  and invariant(V, vc, vp, vn)
  and V sub vc ~ = V sub vp
  imp dedupInvariant(V, vp, vn, W, wn)
  and invariant(V, vc + 1, vp, vn);

assume remdup #4, dedupInvariant(V, vp, vn, W, wn) and (vp <= vn)
  imp dedupInvariant(V, vp, vn, W, wn)
  and invariant(V, 1, vp, vn);
\textbf{assume} \text{remdup # 3,} \quad \text{dedupInvariant}(V, vp, vn, W, wn) \\
\quad \text{and } vp \leq vn \\
\quad \text{and dedupInvariant}(V, vp, vn, W, wn) \\
\quad \text{and inInvariant}(V, vc2, vp, vn) \\
\quad \text{and } V_{sub \ vc2} = V_{sub \ vp} \\
\quad \text{and } vc2 = vp \\
\quad \text{imp dedupInvariant}(V, \\
\quad \quad vp + 1, \\
\quad \quad vn, \\
\quad \quad assn(W, wn + 1, V_{sub \ vp}), \\
\quad \quad wn + 1); \\

\textbf{assume} \text{remdup # 2, } vn \geq 0 \text{ imp dedupInvariant}(V, 1, vn, W, 0); \\

\textbf{assume} \text{remdup # 1, } vn \geq 0 \\
\quad \text{and dedupInvariant}(V, vp1, vn, W2, wn2) \\
\quad \text{and } vn < vp1 \\
\quad \text{imp (wn2 \geq 0) and (rep(W2, wn2) = dedup(rep(V, vn))));} \\

\textbf{assume} \text{computesremdup, } vn \geq 0 \\
\quad \text{and computes(remdup}(V, vn, W, wn), \\
\quad \quad \text{result}(W1, wn1)) \\
\quad \text{imp some vn(some V(\text{\quad wn1} \geq 0 \\
\quad \quad \text{and } \text{rep}(W1, wn1) \\
\quad \quad \quad = \text{dedup(rep}(V, vn)));} \\

\textbf{assume} \text{remdup, verification(remdup);} \\

\textbf{assume} \text{repin, bounds(1, i, j) imp (a_{sub i}) in rep(a, j);} \\

\textbf{assume} \text{repaasn, } \sim \text{bounds}(1, i, j) \\
\quad \text{imp rep(assn(a, i, x), j) = rep(a, j);}
I.3. SimpleSend

This program simulates a ridiculous message sending system. It is used in both the Annotated Transcripts and in the Users' Guide as an annotated example.

PROGRAM

program SendReceive;
{
 This set of three procedures simulates an overly simple message-passing system. In SimpleSend, messages are simply "picked" out of RemainingToBeSent, "sent" to ReceivedSoFar, then deleted from RemainingToBeSent, which decreases from TotalToBeSent down to NewSetOfElemType. After "send" the message is either received or lost. No checks or resends are made so the strongest property we can prove about this program is that ReceivedSoFar is a subset of TotalToBeSent.
}

{ This procedure won't be proved, just left pending. }
procedure pick(s:SetOfElemType var it:ElemType);
pre s- = NewSetOfElemType;
post it in s';
;

{ Nor will this procedure be proved, only assumed. Note that the use of 'or' gives us a kind of non-determinism. }
procedure send(it:ElemType; var rec:SetOfElemType);
pre TRUE;
post rec = rec' add it' or rec = rec';
;
{ Here's the little procedure which simulates sending and receiving messages. }
procedure SimpleSend(TotalToBeSent:SetOfElemType;
 var ReceivedSoFar:SetOfElemType);
pre TRUE;
post ReceivedSoFar subset TotalToBeSent';

var NextToSend:ElemType;
var RemainingToBeSent : SetOfElemType;
begin
 RemainingToBeSent := TotalToBeSent;
 ReceivedSoFar := NewSetOfElemType;

 maintain ReceivedSoFar subset TotalToBeSent
 and RemainingToBeSent subset TotalToBeSent
 while RemainingToBeSent-~ = NewSetOfElemType do
begin
    pick(RemainingToBeSent, NextToSend);
    send(NextToSend, ReceivedSoFar);
    RemainingToBeSent := RemainingToBeSent \ modulo NextToSend;
end;
end;
1.4. A Sorting Algorithm

The program implements the common Selection Sort, where the largest element is found and then moved to the top of the array along with the previously sorted elements. The assertions show that the program computes a recursively defined function, SelectSort, and other lemmas (SelectSortSorts) then show that the Ordering and Permutation properties hold. Considerable notation is developed about the representation function, aspects of ordering and permutation, and sequences.

**PROGRAM**

```plaintext
procedure Sort(var A: MappingFromIntegerToInteger; Ib, ub: Integer);
pre Ib <= ub;
post Ordered(rep(A, Ib, ub)) and Permutation(rep(A', Ib', ub'), rep(A', Ib', ub'));
var NextToSort, NextToCompare, MaxCompared: Integer;
begin
    NextToSort := ub;
    maintain SoFarSorted(A, A', Ib, ub, NextToSort)
    while NextToSort > Ib do
        begin
            NextToCompare, MaxCompared := NextToSort - 1, NextToSort;
            maintain SoFarSorted(A, A', Ib, ub, NextToSort) and
                SoFarCompared(A, Ib, NextToCompare, NextToSort, MaxCompared)
            while NextToCompare > Ib do
                NextToCompare, MaxCompared := NextToCompare - 1,
                    if A sub MaxCompared < (A sub NextToCompare) then
                        NextToCompare else MaxCompared;
                    A, NextToSort := Swap(A, NextToSort, MaxCompared), NextToSort - 1
            end;
        end;
end;
```

**CONTEXT**

```plaintext
type SortNotation;

needs types MappingFromIntegerToInteger, SequenceOfInteger;

declare dummy: SortNotation;
declare A, A', A'', A1, A2: MappingFromIntegerToInteger;
declare diff, i, i', ii, il, Index1, k, k1, k1', k2, k2', k3, lb, lb', MaxCompared,
    MaxCompared', MaxCompared1, MaxCompared2, NextToCompare, NextToCompare1,
    NextToCompare2, NextToSort, NextToSort1, ub, ub', x, x', x'', y, y': Integer;
declare s, s', s'', s1, s1', s1'', s2, s2', ss, ss', sss: SequenceOfInteger;

interfaces
    rep(A, Ib, ub), SelectSort(s), SS(s), SL(s, s1, x), SwapLargest(s),
    DeleteLastOcc(s, x): SequenceOfInteger;
```
interfaces
SoFarSorted(A, A', lb, ub, k1), Permutation(s, s1), Ordered(s), Dominates(s, x),
SoFarCompared(A, lb, NextToCompare, NextToSort, MaxCompared),
RightmostOcc(A, lb, ub, x, k1), bd(lb, k, ub), bd2(lb, k1, k2, ub), DominatesSplit(s),
SomeSplit(s): Boolean;

interface Swap(A, k1, k2): MappingFromIntegerToInteger;

interface Occs(s, x): Integer;

interface Sort(A, lb, ub): ProcedureCall;

axiom dummy = dummy = = TRUE;

axioms
Ordered(NewSequenceOfInteger) = = TRUE,
Ordered(s apr x)
  = = if Ordered(s)
    then (s = NewSequenceOfInteger) or (Last(s) <= x)
    else Last(s) <= x;

axioms
Dominates(s apr i, x) = (Dominates(s, x) and (i <= x)),
Dominates(NewSequenceOfInteger, x) = = TRUE;

axioms
Occs(NewSequenceOfInteger, x) = = 0,
Occs(s apr y, x)
  = = if x = y
    then Occs(s, x) + 1
    else Occs(s, x);

axioms
DeleteLastOcc(NewSequenceOfInteger, x) = NewSequenceOfInteger,
DeleteLastOcc(s apr i, x)
  = = if i = x
    then s
    else DeleteLastOcc(s, x) apr i;

define
  rep(A, lb, ub)
  = = if ub < lb
A Sorting Algorithm

```plaintext
then NewSequenceOfInteger
else rep(A, lb, ub-1) apr (A sub ub),

SelectSort(s) = = if s = NewSequenceOfInteger
then s
else SS(SwapLargest(s)),

SS(s) = = SelectSort(LessLast(s)) apr Last(s),

SL(s, s1, x)
= = if (x < Last(s)) and Dominates(s, Last(s))
then (LessLast(s) join (x apr s1)) apr Last(s)
else SL(LessLast(s), Last(s) apr s1, x),

SwapLargest(s) *
= = if Dominates(s, Last(s))
then s
else SL(LessLast(s), NewSequenceOfInteger, Last(s)),

SoFarSorted(A, A', lb, ub, NextToSort)
= = ( (lb < NextToSort) and (NextToSort < = ub)
and SelectSort(rep(A', lb, ub))
= = SelectSort(rep(A, lb, NextToSort))
join rep(A, NextToSort + 1, ub)),

Permutation(s, s1)
= = all x' (Occs(s, x') = Occs(s1, x')),

SoFarCompared(A, Ib, NextToCompare, NextToSort, MaxCompared)
= = ( (lb < = NextToCompare + 1) and (NextToCompare < = MaxCompared)
and RightmostOcc(A, lb, NextToSort, A sub MaxCompared, MaxCompared)
and MaxCompared < = NextToSort
and Dominates(rep(A, NextToCompare + 1, NextToSort), A sub MaxCompared)),

Swap(A, k1, k2) = = assn(assn(A, k1, A sub k2),
 k2, A sub k1),

RightmostOcc(A, lb, ub, x, k1)
= = ( (x = A sub k1) and (lb < = k1) and (k1 < = ub)
and ~ (x in rep(A, k1 + 1, ub))),

bd(lb, k, ub) = = ((lb < = k) and (k < = ub)),

bd2(lb, k1, k2, ub) = = ( bd(lb, k1, ub)
 or bd(lb, k2, ub)),

DominatesSplit(s)
= = some x, s1, s2
( Dominates(s, x) and (s1 join (x apr s2) = s)

```
and \( \neg(x \text{ in } s2) \);

\textbf{LEMMAS}

\textit{Sequence Lemmas}

\textbf{assume} \text{LastIn}, \text{isNew(s)} \text{ or Last(s) in } s;

\textbf{assume} \text{Length0}, \text{Length(s)} = 0 \text{ eqv isNew(s)};

\textbf{assume} \text{Length1}, \text{Length(s)} = 1 \text{ imp seq(Last(s)) = s};

\textbf{assume} \text{LastSplit}, \text{isNew(s)} \text{ or } (\text{LessLast(s)} \text{ and } \text{Last(s)} = s);

\textbf{assume} \text{LengthNonNeg}, \text{Length(s)} \geq 0;

\textbf{assume} \text{LengthLessLast}, \text{isNew(s)} \text{ or } (\text{Length(LessLast(s))} = \text{Length(s)} \cdot 1);

\textbf{Dominates Lemmas}

\textbf{assume} \text{DominatesIn}, x \text{ in } s \text{ and } \text{Dominates(s, y)} \text{ imp } x \leq y;

\textbf{assume} \text{DominatesSplit}, \text{isNew(s)} \text{ or } \text{DominatesSplit(s)};

\textbf{assume} \text{DominatesNotIn}, \text{Dominates(s, x)} \text{ and } (y > x) \text{ imp } \neg(y \text{ in } s);

\textbf{assume} \text{DominatesJoin}, \text{Dominates(s, x)} \text{ and } \text{Dominates(s1, x)}
\text{ eqv } \text{Dominates(s join s1, x)};

\textbf{assume} \text{DominatesExtend}, \text{Dominates(s, x)} \text{ and } (x < y) \text{ imp } \text{Dominates(s, y)};

\textbf{Permutation Lemmas}

\textbf{assume} \text{PermIn}, i \text{ in } s \text{ and } \text{Permutation(s1, s)} \text{ imp } i \text{ in } s1;

\textbf{assume} \text{PermLength}, \text{Permutation(s1, s)} \text{ imp } \text{Length(s1)} = \text{Length(s)};

\textbf{assume} \text{PermSame}, \text{Permutation(s, s)};
A Sorting Algorithm

**PermCommutes**

\[ \text{assume PermCommutes, Permutation}(s, s_1) \equiv \text{Permutation}(s_1, s); \]

**PermLessLast**

\[ \text{assume PermLessLast, } \neg \text{isNew}(s) \text{ and Permutation}(s_1, \text{LessLast}(s)) \]

\[ \text{imp Permutation}(s_1 \text{ apr Last}(s), s); \]

**PermTransitivity**

\[ \text{assume PermTransitivity, Permutation}(s, s_1) \text{ and Permutation}(s_1, s_2) \]

\[ \text{imp Permutation}(s, s_2); \]

**OrderedPermutation**

\[ \text{assume OrderedPermutation, Ordered}(s) \text{ and Permutation}(s, s_1) \text{ and Dominates}(s_1, x) \]

\[ \text{imp Ordered}(s \text{ apr } x); \]

**SwapLargest Lemmas**

**DominatesSwapLargest**

\[ \text{assume DominatesSwapLargest, isNew}(s) \text{ or Dominates}(\text{SwapLargest}(s), \text{Last}(\text{SwapLargest}(s))); \]

**SwapLargestDominates**

\[ \text{assume SwapLargestDominates, isNew}(s) \text{ or Dominates}(\text{LastLast}(\text{SwapLargest}(s)), \text{Last}(\text{SwapLargest}(s))); \]

**LengthSwapLargest**

\[ \text{assume LengthSwapLargest, isNew}(s) \text{ or } (\text{Length}(\text{SwapLargest}(s)) = \text{Length}(s)); \]

**PermSwapLargest**

\[ \text{assume PermSwapLargest, isNew}(s) \text{ or Permutation}(\text{SwapLargest}(s), s); \]

**SwapLargestSplit**

\[ \text{assume SwapLargestSplit, Dominates}((s_1 \text{ apr } x) \text{ join } s_2, x) \]

\[ \text{and } \neg (x \text{ in } s_2) \text{ and } \neg \text{isNew}(s_2) \]

\[ \text{imp SwapLargest}((s_1 \text{ apr } x) \text{ join } s_2) \]

\[ = (\text{((s_1 \text{ apr } \text{Last}(s_2)) \text{ join } \text{LastLast}(s_2)) \text{ apr } x); \]

**SLSplit**

\[ \text{assume SLSplit, Dominates}((s_1 \text{ apr } x) \text{ join } s_2, x) \]

\[ \text{and } \neg (x \text{ in } s_2) \text{ and } (x > y) \]

\[ \text{imp SL}((s_1 \text{ apr } x) \text{ join } s_2, s, y) \]

\[ = (\text{((s_1 \text{ apr } y) \text{ join } s_2) \text{ join } s) \text{ apr } x; \]

**Occs Lemmas**

**OccsJoin**

\[ \text{assume OccsJoin, Occs}(s_1 \text{ join } s_2, x) \]

\[ = \text{Occs}(s_1, x) + \text{Occs}(s_2, x); \]

**inOccs**

\[ \text{assume inOccs, x in } s \text{ eqv Occs}(s, x) = 0; \]

**OccsNonNeg**

\[ \text{assume OccsNonNeg, Occs}(s, x) \geq 0; \]
SelectSort Lemmas

\textbf{assume} SelectSort1, Length(s) = 1 \implies \text{SelectSort}(s) = s;

\textbf{assume} SelectSortSorts, Ordered(SelectSort(s)) and Permutation(SelectSort(s), s);

DeleteLastOcc Lemmas

\textbf{assume} LengthDeleteLastOcc, Length(DeleteLastOcc(s, i))
= \begin{cases} 
\text{Length}(s) - 1 & \text{if } i \in s \\
\text{Length}(s) & \text{else }
\end{cases}

\textbf{assume} OccsDeleteLastOcc, \text{Occs}(DeleteLastOcc(s, i), x)
= \begin{cases} 
\text{Occs}(s, x) & \text{if } i \neq x \\
\text{Occs}(s, i) - 1 & \text{else if } i \in s \\
0 & \text{else}
\end{cases}

\textbf{assume} PermutationLastOcc, \text{Permutation}(s1, s \setminus i)
\implies \text{Permutation}(\text{DeleteLastOcc}(s1, i), s);

Rep Lemmas

\textbf{assume} rep1, rep(A, lb, ub) = \text{seq}(A \setminus lb);

\textbf{assume} SwapCommutes, Swap(A, k1, k2) = Swap(A, k2, k1);

\textbf{assume} Swap1, Swap(A, k1, k1) \subset k = A \setminus k;

\textbf{assume} Lengthrep, \quad \text{lb} \leq \text{ub}
\implies \text{Length}(\text{rep}(A, \text{lb}, \text{ub})) = (\text{ub} - \text{lb}) + 1;

\textbf{assume} repSplit1, \quad \text{lb} \leq \text{ub}
\implies \text{rep}(A, \text{lb}, \text{ub})
A Sorting Algorithm

= (A sub lb) apl rep(A, lb + 1, ub);

\textbf{assume} repSplit, \quad (lb \leq k1) \text{ and } (k1 \leq ub)
imp \quad rep(A, lb, k1)
join \ rep(A, k1 + 1, ub)
= rep(A, lb, ub);

\textbf{assume} repNew, isNew(rep(A, lb, ub)) imp ub < lb;

\textbf{assume} lemma1OfrepSwap, \quad \sim bd2(lb, k1, k2, ub)
imp \ rep(Swap(A, k1, k2), lb, ub)
= rep(A, lb, ub);

\textbf{assume} repSwap, \quad (lb \leq k1) \text{ and } (k1 \leq ub)
or \quad (lb \leq k2) \text{ and } (k2 \leq ub)
imp \ rep(Swap(A, k1, k2), lb, ub)
= rep(A, lb, ub);

\textbf{assume} VCS

\textbf{assume} ExitSortLoop, \quad lb \leq ub
and \ SoFarSorted(A2, A, lb, ub, NextToSort1)
and \ NextToSort1 \leq lb
imp \ Ordered(rep(A2, lb, ub))
and \ Permutation(rep(A2, lb, ub),
rep(A, lb, ub));

\textbf{assume} TraverseCompareLoop, \quad SoFarSorted(A,
A', lb, ub, NextToSort)
and \ SoFarCompared(A,
lb,
NextToCompare, NextToSort, MaxCompared)
and \ NextToCompare \geq lb
imp \ SoFarSorted(A,
A', lb, ub, NextToSort)
and \ SoFarCompared(A,
lb,
NextToCompare-1,
NextToSort,
if \ A sub MaxCompared < A sub NextToCompare
then \ NextToCompare
else \ MaxCompared);

\textbf{assume} ExitCompareLoop, \quad SoFarSorted(A,
A', lb, ub, NextToSort)
and \ NextToSort \geq lb
and \ SoFarSorted(A,
\[ A', \text{ib, ub, NextToSort} \]
and SoFarCompared(A,
ib, NextToCompare2, NextToSort,
MaxCompared2)
and NextToCompare2<ib
imp SoFarSorted(Swap(A, NextToSort, MaxCompared2),
A', \text{ib, ub, NextToSort-1});

\textbf{assume } EnterCompareLoop, \quad \text{SoFarSorted}(A,
A', \text{ib, ub, NextToSort})
and NextToSort>ib
imp \quad \text{SoFarSorted}(A,
A', \text{ib, ub, NextToSort})
and SoFarCompared(A,
ib,
NextToSort-1, NextToSort,
NextToSort);\]

\textbf{assume } EnterSortLoop, \quad \text{ib} \leq \text{ub} \quad \text{imp SoFarSorted}(A,
A, \text{ib, ub, ub});
Appendix II
PVLIBRARY

These are the files that should be in PVLIBRARY on every machine at all times. Version numbers will probably differ. Please report missing files via a gripe.

PS:<PVLIBRARY>
AFFIRMTRANSCRIPT.PRS.1
ANNOTALIB.2
BINARYTREEOFELEMTYPE..1
.ADDLAXIOMS.1
.AXIOMS.1
.COM.1
.DOCUMENTATION.2
CIRCLEOFELEMTYPE..1
.AXIOMS.1
.COM.1
.DOCUMENTATION.2
DEMO-HANDBOOK.ELEMTYPE-NOTES.3
.INTRODUCTION.3
.MSS.12
.SEQUENCE-NOTES.8
.TITLE-PAGE.7
ELEMTYPE..3
.COM.4
.DOCUMENTATION.3
FOO..1,2
GRAPHOFELEMTYPE..1
.AXIOMS.1
.DOCUMENTATION.6
.LEMMAS.1
HEADER.MSS.5,6
INTEGER.DOCUMENTATION.2
LESSON.DOCUMENTATION.4
.GREETING.3
.NOTATION.6
.PROOFS.1
.SETUP.5
.THEOREMS.4
.TRANSCRIPT.1
MAPPINGFROMDOMAINTORANGE..1
.AXIOMS.7
.COM.1
References


