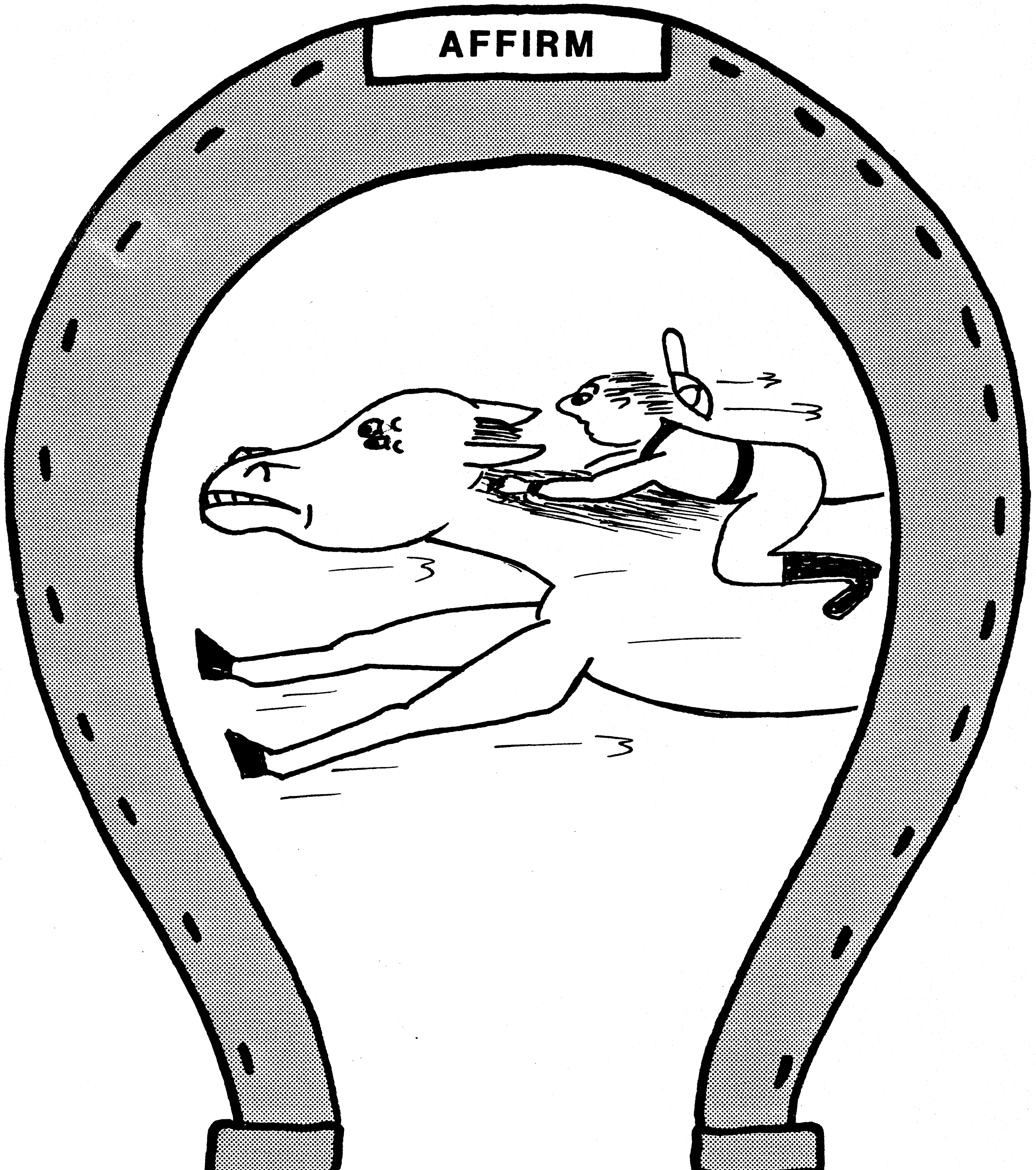

AFFIRM Annotated Transcripts

Raymond L. Bates and Susan L. Gerhart, Editors

AFFIRM

Annotated Transcripts

Raymond L. Bates and Susan L. Gerhart, Editors

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

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

The USC/Information Sciences Institute Program Verification Project is headed by Susan L. Gerhart, with members Roddy W. Erickson, Stanley Lee, Lisa Moses, and David H. Thompson. Past project members include Raymond L. Bates, Ralph L. London, David R. Musser, David G. Taylor, and David S. Wile.

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

Preface

The ***AFFIRM*** Annotated Transcripts volume illustrates a number of features of ***AFFIRM***. Each transcript is prefaced with a short description of what the transcript deals with and other highlights. All of these transcripts are from the current system as of the writing of this volume. Some of these proofs are highly polished and many people contributed to them.

1. Proof of subseq transitivity

The main point of this proof, aside from its 'historical' significance and that it took us a week to find, is the reasoning involved in disjunctions. The axiomatic definition of subseq is

```
s subseq NewSequenceOfInteger == (s=NewSequenceOfInteger)

s subseq (s1 apr i) == (s=NewSequenceOfInteger
         or s subseq s1
         or Last(s)=i and LessLast(s) subseq s1)
```

The creation of the axioms for subseq was stimulated by John Ulrich's posing this problem to us during his visit to ISI. The first axioms we came up with were like

```
NewSequenceOfInteger subseq s == TRUE

(s apr i) subseq NewSequenceOfInteger == FALSE

(s apr i) subseq (s1 apr i1) ==
        (i=i1 and s subseq s1 or s apr i subseq s1)
```

This formulation follows the way one would program subseq, chopping off end elements successively and comparing them. The proof of subseq transitivity using these axioms was never accomplished because we could not find any way to use our usual Induction. (Moore accomplished this same proof at SRI using a rather complicated ordering and their generalized induction mechanism.) With the change in axioms the following proof was easily found. Gerrard Terrine of IRIA pointed out to us that our second axioms have the seed of transitivity built into them.

There are some lessons here. First, you have to decide whether to have many induction methods or just a few schemas. We have chosen the latter course, whereas Boyer and Moore extensively pursued the former. Second, these axioms are not programs. The variability of expressiveness in axioms beyond the usual recursion seems worth using.

See Appendix I on page 56 for a listing of the type SequenceOfInteger.

Transcript file <RBATES>AFFIRMTRANSCRIPT.14-NOV-80.7
is open in the AFFIRM system <AFFIRM>AFFIRM.EXE.120

1 U:

We are increasing the linewidth so the transcript fills a printed page (the default value is 79).

2 U: **profile TerminalLineWidth = 88;**

TerminalLineWidth: 88

3 U: **needs type SequenceOfInteger;**

compiled for AFFIRM on 7-Aug-80 12:38:59
file created for AFFIRM on 7-Aug-80 11:10:22
SEQUENCEOFINTEGERCOMS

(File created under Affirm 108)
<PVLIBRARY>SEQUENCEOFINTEGER.COM.3

4 U:

The <u>needs</u> command gives us the current SequenceofInteger. The message about "(File created ...)" states that the loaded version of SequenceOfInteger wasn't created in the current version of Affirm, but the version difference is irrelevant. The library version of SequenceOfInteger will be different if you repeat this transcript.

5 U: **declare s1,s2,s3:SequenceOfInteger;**

s1, s2, s3: SequenceOfInteger

6 U: **try subseqTrans, s1 subseq s2 and s2 subseq s3 imp s1 subseq s3;**

Making node subseqTrans a theorem.
subseqTrans is untried.

all s1, s2, s3 (s1 subseq s2 and s2 subseq s3 imp s1 subseq s3)

7 U:

It should be clear after studying the axioms that s3 is the most natural variable to induct on, since the axioms permute the constructors through the second parameter position of subseq.

8 U: **employ Induction(s3);**

Case NewSequenceOfInteger: Prop(NewSequenceOfInteger) remains to be shown.
Case apr: all ss, ii (IH(ss) imp Prop(ss apr ii)) remains to be shown.
(NewSequenceOfInteger:)
all s1, s2 (s1 subseq s2 and (s2 = NewSequenceOfInteger) imp s1 = NewSequenceOfInteger)

9 U: **replace s2;**

TRUE

10 U:

We have proved the basis case (NewSequenceOfInteger) so off we go to the induction step. We could have turned on AutoNext, a profile entry, so we wouldn't have to explicitly do the next command.

11 U: **next;**

Going to leaf apr:.

```
all ss', ii', s1, s2
 (      IH(ss', 1 {subseqTrans}) and s1 subseq s2
     and    (s2 = NewSequenceOfInteger) or s2 subseq ss'
        or LessLast(s2) subseq ss' and (Last(s2) = ii')
  imp   (s1 = NewSequenceOfInteger) or s1 subseq ss'
     or LessLast(s1) subseq ss' and (Last(s1) = ii'))
```

12 U: **suppose s2 = NewSequenceOfInteger ;**

```
(yes:)
all ss', ii', s1, s2
 (      (s2 = NewSequenceOfInteger) and IH(ss', 1 {subseqTrans})
     and s1 subseq s2
  imp   (s1 = NewSequenceOfInteger) or s1 subseq ss'
     or LessLast(s1) subseq ss' and (Last(s1) = ii'))
```

13 U:

We could have done a split instead, but sometimes it is harder to figure out what a split will do.

14 U: **print proof ;**

theorem subseqTrans, s1 subseq s2 and s2 subseq s3 imp s1 subseq s3;

```
proof tree:
8:|   subseqTrans
         employ Induction(s3)
9:    NewSequenceOfInteger:
        2   replace s2
        (proven!)
12:   apr:
        3   suppose s2 = NewSequenceOfInteger
?->      yes:
```

4

? no:5

15 U: **replace s2;**

TRUE

16 U: **next;**

Going to leaf no:.

```
all ss', ii', s1, s2
 (      (s2 ~= NewSequenceOfInteger) and IH(ss', 1 {subseqTrans})
      and s1 subseq s2
      and    s2 subseq ss'
         or LessLast(s2) subseq ss' and (Last(s2) = ii')
   imp    (s1 = NewSequenceOfInteger) or s1 subseq ss'
      or LessLast(s1) subseq ss' and (Last(s1) = ii'))
```

17 U:

The other case of event 12: not(s2 =NewSequenceOfInteger).

18 U: **suppose s2 subseq ss' ;**

```
(yes:)
all ss', ii', s1, s2
 (      s2 subseq ss' and (s2 ~= NewSequenceOfInteger)
      and IH(ss', 1 {subseqTrans})
      and s1 subseq s2
   imp    (s1 = NewSequenceOfInteger) or s1 subseq ss'
      or LessLast(s1) subseq ss' and (Last(s1) = ii'))
```

19 U: **invoke IH ;**

Now we need the Induction Hypothesis to link s1, ss', and s2 transitively.

```
all ss', ii', s1, s2 (some s1', s2'
 (      s2 subseq ss' and (s2 ~= NewSequenceOfInteger)
      and s1' subseq s2' and s2' subseq ss' imp s1' subseq ss'
      and s1 subseq s2
   imp    (s1 = NewSequenceOfInteger) or s1 subseq ss'
      or LessLast(s1) subseq ss' and (Last(s1) = ii')))
```

20 U:

After invoking the induction hypothesis we have to instantiate s1' and s2', the free variables of the prop we are inducting on. Let's see if <u>*search*</u> *can find it.*

21 U: **search ;**

1/13: (s2' = ss') and (s1' = s2)
2/13: s2' = s2
   1/3: s1' = s2
   2/3: s1' = s1
Proved by chaining and narrowing
using the substitution

(s2' = s2) and (s1' = s1)

TRUE

22 U: **next;**

Going to leaf no:.

```
all ss', ii', s1, s2
 (      ~(s2 subseq ss') and (s2 ~= NewSequenceOfInteger)
     and IH(ss', 1 {subseqTrans})
     and s1 subseq s2
     and LessLast(s2) subseq ss'
     and Last(s2) = ii'
  imp   (s1 = NewSequenceOfInteger) or s1 subseq ss'
     or LessLast(s1) subseq ss' and (Last(s1) = ii'))
```

23 U:

The other case from event 18: not(s2 subseq ss'). The system doesn't realize that if s2 isn't NewSequenceOfInteger then s2=LessLast(s2) apr Last(s2), but by employing the NormalForm schema we will enumerate the cases that s2 can take on (NewSequenceOfInteger or apr) thus firing the axioms for subseq with respect to s2.

24 U: **employ NormalForm(s2) ;**

Case NewSequenceOfInteger: Prop(NewSequenceOfInteger) proven.
Case apr: all ss, ii (Prop(ss apr ii)) remains to be shown.

```
(apr:)
all ss', ii', ss, ii, s1
 (      ~((ss' apr ii') subseq ss) and IH(ss, 1 {subseqTrans})
     and s1 ~= NewSequenceOfInteger
     and    s1 subseq ss'
        or LessLast(s1) subseq ss' and (Last(s1) = ii')
     and ss' subseq ss
```

```
    and ii' = ii
  imp   s1 subseq ss
    or LessLast(s1) subseq ss and (Last(s1) = ii))
```

25 U: **replace ii ;**

```
all ss', ii', ss, ii, s1
 (   ~((ss' apr ii') subseq ss) and IH(ss, 1 {subseqTrans})
  imp   s1 = NewSequenceOfInteger
    or if s1 subseq ss'
       then    ss' subseq ss and (ii' = ii)
           imp   s1 subseq ss
              or   LessLast(s1) subseq ss
                 and Last(s1) = ii'
       else        LessLast(s1) subseq ss'
              and Last(s1) = ii'
              and ss' subseq ss
              and ii' = ii
           imp s1 subseq ss or LessLast(s1) subseq ss)
```

26 U: **suppose s1 subseq ss' ;**

```
(yes:)
all ss', ii', ss, ii, s1
 (      s1 subseq ss' and ~((ss' apr ii') subseq ss)
     and IH(ss, 1 {subseqTrans})
     and s1 ~= NewSequenceOfInteger
     and ss' subseq ss
     and ii' = ii
  imp   s1 subseq ss
     or LessLast(s1) subseq ss and (Last(s1) = ii'))
```

27 U: **invoke IH ;**

```
all ss', ii', ss, ii, s1 (some s1', s2
 (      s1 subseq ss' and ~((ss' apr ii') subseq ss)
     and s1' subseq s2 and s2 subseq ss imp s1' subseq ss
     and s1 ~= NewSequenceOfInteger
     and ss' subseq ss
     and ii' = ii
  imp   s1 subseq ss
     or LessLast(s1) subseq ss and (Last(s1) = ii')))
```

28 U: **search ;**

```
1/16: (s2 = ss') and (s1' = s1)
```

Proved by chaining and narrowing
using the substitution

(s2 = ss') and (s1' = s1)

TRUE

29 U: **next;**

Going to leaf no:.

```
all ss', ii', ss, ii, s1
 (       ~(s1 subseq ss')
     and ~((ss' apr ii') subseq ss)
     and IH(ss, 1 {subseqTrans})
     and s1 ~= NewSequenceOfInteger
     and LessLast(s1) subseq ss'
     and Last(s1) = ii'
     and ss' subseq ss
     and ii' = ii
  imp s1 subseq ss or LessLast(s1) subseq ss)
```

30 U:

The other case of step 26 : not(s1 subseq ss').

31 U: **invoke IH ;**

```
all ss', ii', ss, ii, s1 (some s1', s2
 (       ~(s1 subseq ss')
     and ~((ss' apr ii') subseq ss)
     and s1' subseq s2 and s2 subseq ss imp s1' subseq ss
     and s1 ~= NewSequenceOfInteger
     and LessLast(s1) subseq ss'
     and Last(s1) = ii'
     and ss' subseq ss
     and ii' = ii
  imp s1 subseq ss or LessLast(s1) subseq ss))
```

32 U: **search ;**

```
1/17: (s2 = ss') and (s1' = s1)
2/17: (s2 = ss) and (s1' = ss' apr ii')
3/17: s2 = ss' apr ii'
4/17: s1' = ss' apr ii'
   1/5: s2 = ss
   2/5: s2 = ss' apr ii'
```

```
   3/5:  s2 = ss'
   4/5:  s2 = s1
   5/5:  s2 = LessLast(s1)
5/17:  (s2 = ss') and (s1' = LessLast(s1))
Proved by chaining and narrowing
using the substitution

(s2 = ss') and (s1' = LessLast(s1))

TRUE
subseqTrans proved.
```

33 U:

We are all done with subseqTrans, let's review the proof.

34 U: **print proof ;**

```
theorem subseqTrans, s1 subseq s2 and s2 subseq s3 imp s1 subseq s3;

proof tree:
8:!   subseqTrans
        employ Induction(s3)
9:     NewSequenceOfInteger:
        2   replace s2
        (proven!)
12:    apr:
        3   suppose s2 = NewSequenceOfInteger
15:      yes:
          4   replace s2
          (proven!)
18:      no:5   suppose s2 subseq ss'
19:        yes:
            6   invoke IH
21:           8   put (s2' = s2) and (s1' = s1) {search}
21:           (proven!)
24:        no:7   employ NormalForm(s2)
            NewSequenceOfInteger:
             Immediate
25:          apr:
             10   replace ii
26:            11   suppose s1 subseq ss'
27:            yes:
               12   invoke IH
28:              14   put (s2 = ss') and (s1' = s1) {search}
28:              (proven!)
31:            no:13   invoke IH
32:              16   put    s2 = ss'
                        and s1' = LessLast(s1) {search}
```

32:-> (proven!)

35 U: **quit;**

Type CONTINUE to return to AFFIRM.

2. The Knuth-Bendix Algorithm on Group Theory Axioms

This transcript shows the Knuth-Bendix algorithm generating a long sequence of rules. You will note we start with 3 rules, the 3 axioms which define a group:

```
axiom e op x = x;
axiom inv(x) op x = e;
axiom (x op y) op z = x op (y op z)
```

and end up with the 3 rules above and 7 rule lemmas:

```
inv(e) = e
inv(inv(y)) = y
inv(y op y'') = inv(y'') op inv(y)
inv(y) op (y op z) = z
z op e = z
y op (inv(y) op z) = z
y op inv(y) = e
```

This process is not automatic. You will note in the middle of the transcript when the system is proposing a new rule we have to reverse its direction (see page 14).

The rule lemmas could have been proven as theorems by induction using the first three axioms. The "induction" accomplished by Knuth-Bendix is discussed in [Musser 80].

Transcript file <RBATES>AFFIRMTRANSCRIPT.7-NOV-80.3
is open in the AFFIRM system <AFFIRM>AFFIRM.EXE.120

1 U: **print file <affirm>grp1.axioms ;**

<AFFIRM>GRP1.AXIOMS.3:

```
type Grp1;
declare x,y,z:Grp1;
interface e:Grp1;
interface inv(x),op(x,y):Grp1;
infix op;
axiom (x = x) = TRUE;
axiom e op x = x;
axiom inv(x) op x = e;
axiom (x op y) op z = x op (y op z);
```

```
end;

2 U: read <affirm>grp1.axioms ;

(Reading AFFIRM commands from <AFFIRM>GRP1.AXIOMS.3)
type Grp1
reflexive: Grp1
New rule:
    reflexive = reflexive
  -> TRUE
1/1. Affirmed.
x, y, z: Grp1
Rule simplifies to TRUE.  Affirmed.
New rule:
    e op x
  -> x
1/1. Affirmed.
New rule:
    inv(x) op x
  -> e
1/1.. Affirmed.
New rule:
    (x op y) op z
  -> x op (y op z)
1/1.!!.!
From (x op y) op z = = x op (y op z)
and inv(x) op x = = e
we obtain a new rule:
    inv(y) op (y op z)
  -> z
.!
2/2.!
From inv(y) op (y op z) = = z
and  inv(y) op (y op z) = = z
we obtain a new rule:
    inv(inv(y)) op z
  -> y op z
!.!!
From inv(y) op (y op z) = = z
and  (x op y) op z = = x op (y op z)
we obtain a new rule:
    inv(x op y') op (x op (y' op z))
  -> z
.!
From inv(y) op (y op z) = = z
and  inv(x) op x = = e
we obtain a new rule:
    z op e
  -> z
```

.!
From inv(y) op (y op z) = = z
and e op x = = x
we obtain a new rule:
inv(e) op z
-> z
3/6..!.!!.!
From z op e = = z
and inv(x) op x = = e
we obtain a new rule:
inv(e)
-> e
.
4/7.!.!.
5/7 discarding rule
inv(e) op z = z

6/7.!.!
From inv(inv(y)) op z = = y op z
and z op e = = z
we obtain a new rule:
inv(inv(y))
-> y
. discarding rule
inv(inv(y)) op z = y op z

7/8.!
From inv(inv(y)) = = y
and inv(y) op (y op z) = = z
we obtain a new rule:
y op (inv(y) op z)
-> z
.!
From inv(inv(y)) = = y
and inv(x) op x = = e
we obtain a new rule:
y op inv(y)
-> e
..!
8/10.!.!...!.!
From y op inv(y) = = e
and (x op y) op z = = x op (y op z)
we obtain a new rule:
x op (y' op inv(x op y'))
-> e
!..!
. 9/11.!.!.!.!!.!..!
From y op (inv(y) op z) = = z
and (x op y) op z = = x op (y op z)

we obtain a new rule:
x op (y' op (inv(x op y') op z))
-> z
!.!.!
10/12...!!.!.!!
From x op (y' op inv(x op y')) = = e
and y op (inv(y) op z) = = z
we obtain a new rule:
y' op inv(inv(y) op y')
-> y
!.:.!!
From x op (y' op inv(x op y')) = = e
and inv(y) op (y op z) = = z
we obtain a new rule:
y' op inv(y op y')
-> inv(y)
!.!
From x op (y' op inv(x op y')) = = e
and (x op y) op z = = x op (y op z)
we obtain a new rule:
x' op (y op (y' op inv(x' op (y op y'))))
-> e
!!!.!.!!!
11/15...!
From y' op inv(y op y') = = inv(y)
and y' op inv(y op y') = = inv(y)
we obtain a new rule:
inv(y op y # 3) op y
-> inv(y # 3)
!. discarding rule
x op (y' op inv(x op y')) = e

.!.!
From y' op inv(y op y') = = inv(y)
and y op (inv(y) op z) = = z
we obtain a new rule:
inv(y op inv(y''))
-> y'' op inv(y)
!.!.!
From y' op inv(y op y') = = inv(y)
and inv(y) op (y op z) = = z
we obtain a possible new rule:
inv(y'') op inv(y) = = inv(y op y'')
Ok? ?

These are the user's options when Knuth-Bendix asks what to do with a possible new rule:

one of:
Yes
Reverse it
Suppress it (put it on the list "BadEquations")

Treat it as Equation -> TRUE
Instead accept another equation from the terminal

The reversal decision was based on the structure-reducing characteristic of finite termination.

Ok? Reverse it [confirm]
!.!!!.!.!!
12/18. discarding rule
y' op inv(y op y') = inv(y)

.!!.!!.!!.!!.!.!.!..!.
13/18 discarding rule
inv(y op inv(y'')) = y'' op inv(y)

14/18 discarding rule
inv(y op y # 3) op y = inv(y # 3)

15/18 discarding rule
y' op inv(inv(y) op y') = y

16/18 discarding rule
inv(x op y') op (x op (y' op z)) = z

17/18 discarding rule
x op (y' op (inv(x op y') op z)) = z

18/18 discarding rule
x' op (y op (y' op inv(x' op (y op y')))) = e

Affirmed.
Leaving Grp1; now editing Basis.

3 U: **print type Grp1 ;**

type *Grp1*;

declare reflexive, x, y, z, y', x', y # 3, y'': Grp1;

interfaces e, inv(x), x op y: Grp1;

infix op;

axiom reflexive = reflexive = = TRUE;

axioms e op x = = x,
inv(x) op x = = e,
(x op y) op z = = x op (y op z);

```
rulelemmas inv(e) = = e,
      inv(inv(y)) = = y,
      inv(y op y'') = = inv(y'') op inv(y);

rulelemmas inv(y) op (y op z) = = z,
      z op e = = z,
      y op (inv(y) op z) = = z,
      y op inv(y) = = e;

end {
Grp1} ;

4 U: quit;
```

Type CONTINUE to return to AFFIRM.

3. Simple Send VCs

This transcript shows a verification of very simple message-passing system. The system is described by a Pascal-like program. It uses two abstract data types, SetOfElemType and ElemType. The data types are pre-defined and kept in <PVLIBRARY> and used to verify the program (see page 56, Appendix I, for a listing of the type SetOfElemType). More extensive annotation and explanation of the first part appear in the User's Guide.

Transcript file <RBATES>AFFIRMTRANSCRIPT.11-NOV-80.2
is open in the AFFIRM system <AFFIRM>AFFIRM.EXE.120

1 U:

We are increasing the AverageNameLength and the TerminalLineWidth so this transcript looks better for our printing device.

2 U: profile AverageNameLength = 10;

AverageNameLength: 10

3 U: profile TerminalLineWidth = 88;

TerminalLineWidth: 88

4 U: needs type SetOfElemType;

compiled for AFFIRM on 22-Aug-80 14:20:53
file created for AFFIRM on 22-Aug-80 14:20:16
SETOFELEMTYPECOMS

(File created under Affirm 111)
compiled for AFFIRM on 22-Aug-80 09:22:12
file created for AFFIRM on 22-Aug-80 09:22:00
ELEMTYPECOMS

(File created under Affirm 111)
<PVLIBRARY>ELEMTYPE.COM.2<PVLIBRARY>SETOFELEMTYPE.COM.2

5 U: print file <pvlibrary>simplesend.program ;

<PVLIBRARY>SIMPLESEND.PROGRAM.7:

```
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 rem NextToSend;
      end;
end;
```

6 U:

The units <u>pick</u> and <u>send</u> are not provable since we didn't supply their bodies.

7 U: **readp <pvlibrary>simplesend.program ;**

(Reading Pascal program units from <PVLIBRARY>SIMPLESEND.PROGRAM.7)
The program units are SendReceive, pick, send, and SimpleSend.
Type checking SendReceive... pick... send... SimpleSend...

8 U: **genvcs SimpleSend;**

The result lemma for SimpleSend is:

```
assume computesSimpleSend,    computes(SimpleSend(TotalToBeSent, ReceivedSoFar),
                     result(ReceivedSoFar1))
            imp ReceivedSoFar1 subset TotalToBeSent;
```

There are 3 verification conditions for SimpleSend:

```
theorem SimpleSend # 1,       ReceivedSoFar2 subset TotalToBeSent
              and RemainingToBeSent1 subset TotalToBeSent
              and RemainingToBeSent1 = NewSetOfElemType
            imp ReceivedSoFar2 subset TotalToBeSent;

theorem SimpleSend # 2,    NewSetOfElemType subset TotalToBeSent
            and TotalToBeSent subset TotalToBeSent;

theorem SimpleSend # 3,       ReceivedSoFar subset TotalToBeSent
              and RemainingToBeSent subset TotalToBeSent
              and RemainingToBeSent ~ = NewSetOfElemType
              and computes(pick(RemainingToBeSent, NextToSend),
                     result(NextToSend2))
              and computes(send(NextToSend2, ReceivedSoFar),
                     result(ReceivedSoFar3))
            imp    ReceivedSoFar3 subset TotalToBeSent
              and (RemainingToBeSent rem NextToSend2) subset TotalToBeSent
;
```

(VC1:)
Program SimpleSend is awaiting the proof of vcs SimpleSend # 1, SimpleSend # 2
, and SimpleSend # 3.

9 U: **genvcs pick,send;**

The result lemma for pick is:

```
assume computespick,     s ~ = NewSetOfElemType
```

```
        and computes(pick(s, it),
                result(it1))
      imp it1 in s;
```

There is 1 verification condition for pick:

```
theorem pick # 1,    s ~= NewSetOfElemType
        imp it in s;
(VC1:)
```

Program pick is awaiting the proof of vc pick # 1.

The result lemma for send is:

```
assume computessend,    computes(send(it, rec),
                 result(rec1))
        imp    rec1 = rec add it
           or rec1 = rec;
```

There is 1 verification condition for send:

```
theorem send # 1,    rec = rec add it
        or rec = rec;
(VC1:)
```

Program send is awaiting the proof of vc send # 1.

10 U:

Try to prove the first verification condition for the unit SimpleSend.

11 U: **try SimpleSend # 1;**

SimpleSend # 1 is untried.

TRUE
SimpleSend # 1 proved.

12 U: **next;**

There's more than one unproven ancestor. You may pick one of SimpleSend # 2 or SimpleSend # 3.

13 U:

We have proved SimpleSend# 1. The system doesn't want to pick either verification condition so we will pick number 2.

14 U: **try SimpleSend # 2;**

SimpleSend # 2 is untried.

```
all TotalToBeSent (    NewSetOfElemType subset TotalToBeSent
            and TotalToBeSent subset TotalToBeSent)
```

15 U: **invoke subset|all| ;**

TRUE
SimpleSend # 2 proved.

16 U: **next;**

Going to unproven ancestor SimpleSend # 3.
SimpleSend # 3 is untried.

```
all ReceivedSoFar,
  TotalToBeSent,
  RemainingToBeSent, NextToSend, NextToSend2, ReceivedSoFar3
 (      ReceivedSoFar subset TotalToBeSent and RemainingToBeSent subset
TotalToBeSent
     and RemainingToBeSent ~= NewSetOfElemType
     and computes(pick(RemainingToBeSent, NextToSend),
             result(NextToSend2))
     and computes(send(NextToSend2, ReceivedSoFar),
             result(ReceivedSoFar3))
  imp     ReceivedSoFar3 subset TotalToBeSent
     and (RemainingToBeSent rem NextToSend2) subset TotalToBeSent)
```

17 U:

We are at the third verification condition. We will apply the computes lemma for pick and send.

18 U: **apply computespick;**

```
some s, it, it1
 (      s ~= NewSetOfElemType
     and computes(pick(s, it),
             result(it1))
  imp it1 in s)
```

19 U: **apply computessned;**

(computessned => computessend)

```
some it', rec, rec1
 (   computes(send(it', rec), result(rec1))
```

```
  imp    rec1 = rec add it'
     or rec1 = rec)
```

20 U: **put s=RemainingToBeSent, it=NextToSend, it1=NextToSend2, it'=NextToSend2, rec=ReceivedSoFar, rec1=ReceivedSoFar3 ;**

```
all ReceivedSoFar,
  TotalToBeSent,
  RemainingToBeSent, NextToSend, NextToSend2, ReceivedSoFar3
 (      computes(send(NextToSend2, ReceivedSoFar),
             result(ReceivedSoFar3))
     and     ReceivedSoFar3
          = ReceivedSoFar add NextToSend2
        or ReceivedSoFar3 = ReceivedSoFar
     and RemainingToBeSent ~= NewSetOfElemType
     and computes(pick(RemainingToBeSent, NextToSend),
             result(NextToSend2))
     and NextToSend2 in RemainingToBeSent
     and ReceivedSoFar subset TotalToBeSent
     and RemainingToBeSent subset TotalToBeSent
  imp    ReceivedSoFar3 subset TotalToBeSent
     and      RemainingToBeSent rem NextToSend2
        subset TotalToBeSent)
```

21 U: **declare s1:SetOfElemType;**

s1: SetOfElemType

22 U: **apply remSubset, s subset s1 imp s rem x subset s1;**

```
some s, s1, x
 (   s subset s1
  imp (s rem x) subset s1)
```

23 U: **put s=RemainingToBeSent,s1=TotalToBeSent,x=NextToSend2;**

```
all ReceivedSoFar,
  TotalToBeSent,
  RemainingToBeSent, NextToSend, NextToSend2, ReceivedSoFar3
 (      RemainingToBeSent subset TotalToBeSent
     and      RemainingToBeSent rem NextToSend2
        subset TotalToBeSent
     and computes(send(NextToSend2, ReceivedSoFar),
             result(ReceivedSoFar3))
     and     ReceivedSoFar3
          = ReceivedSoFar add NextToSend2
```

```
      or ReceivedSoFar3 = ReceivedSoFar
    and RemainingToBeSent ~= NewSetOfElemType
    and computes(pick(RemainingToBeSent, NextToSend),
          result(NextToSend2))
    and NextToSend2 in RemainingToBeSent
    and ReceivedSoFar subset TotalToBeSent
  imp ReceivedSoFar3 subset TotalToBeSent)
```

24 U: **suppose ReceivedSoFar3=ReceivedSoFar add NextToSend2;**

```
(yes:)
all ReceivedSoFar,
  TotalToBeSent,
  RemainingToBeSent, NextToSend, NextToSend2, ReceivedSoFar3
 (      ReceivedSoFar3
      = ReceivedSoFar add NextToSend2
    and RemainingToBeSent subset TotalToBeSent
    and     RemainingToBeSent rem NextToSend2
      subset TotalToBeSent
    and computes(send(NextToSend2, ReceivedSoFar),
          result(ReceivedSoFar3))
    and RemainingToBeSent ~= NewSetOfElemType
    and computes(pick(RemainingToBeSent, NextToSend),
          result(NextToSend2))
    and NextToSend2 in RemainingToBeSent
    and ReceivedSoFar subset TotalToBeSent
  imp ReceivedSoFar3 subset TotalToBeSent)
```

25 U: **replace ReceivedSoFar3;**

```
all ReceivedSoFar,
  TotalToBeSent,
  RemainingToBeSent, NextToSend, NextToSend2, ReceivedSoFar3
 (      ReceivedSoFar3
      = ReceivedSoFar add NextToSend2
    and RemainingToBeSent subset TotalToBeSent
    and     RemainingToBeSent rem NextToSend2
      subset TotalToBeSent
    and computes(send(NextToSend2, ReceivedSoFar),
          result(ReceivedSoFar add NextToSend2))
    and RemainingToBeSent ~= NewSetOfElemType
    and computes(pick(RemainingToBeSent, NextToSend),
          result(NextToSend2))
    and NextToSend2 in RemainingToBeSent
    and ReceivedSoFar subset TotalToBeSent
  imp (ReceivedSoFar add NextToSend2) subset TotalToBeSent)
```

26 U: **apply addSubset, s subset s1 and x in s1 imp s add x subset s1;**

```
some s, s1, x
 (   s subset s1 and x in s1
  imp (s add x) subset s1)
```

27 U: **put s=ReceivedSoFar,s1=TotalToBeSent,x=NextToSend2;**

```
all ReceivedSoFar,
  TotalToBeSent,
  RemainingToBeSent, NextToSend, NextToSend2, ReceivedSoFar3
 (      ReceivedSoFar subset TotalToBeSent
     and ~(NextToSend2 in TotalToBeSent)
     and  ReceivedSoFar3
        = ReceivedSoFar add NextToSend2
     and RemainingToBeSent subset TotalToBeSent
     and       RemainingToBeSent rem NextToSend2
        subset TotalToBeSent
     and computes(send(NextToSend2, ReceivedSoFar),
             result(ReceivedSoFar add NextToSend2))
     and RemainingToBeSent ~ = NewSetOfElemType
     and computes(pick(RemainingToBeSent, NextToSend),
             result(NextToSend2))
     and NextToSend2 in RemainingToBeSent
  imp (ReceivedSoFar add NextToSend2) subset TotalToBeSent)
```

28 U: **apply inSubset, x in s and s subset s1 imp x in s1;**

```
some x, s, s1
 (   x in s and s subset s1
  imp x in s1)
```

29 U: **search ;**

```
1/9:     s = TotalToBeSent
    and x = NextToSend2
2/9:     s = RemainingToBeSent
    and x = NextToSend2
   1/2: s1 = RemainingToBeSent
   2/2: s1 = TotalToBeSent
Proved by chaining and narrowing
using the substitution

   s = RemainingToBeSent
and x = NextToSend2
and s1 = TotalToBeSent
```

TRUE

30 U: **next;**

Going to leaf no:.

```
all ReceivedSoFar,
  TotalToBeSent,
  RemainingToBeSent, NextToSend, NextToSend2, ReceivedSoFar3
 (        ReceivedSoFar3
       ~= ReceivedSoFar add NextToSend2
    and RemainingToBeSent subset TotalToBeSent
    and      RemainingToBeSent rem NextToSend2
       subset TotalToBeSent
    and computes(send(NextToSend2, ReceivedSoFar),
            result(ReceivedSoFar3))
    and ReceivedSoFar3 = ReceivedSoFar
    and RemainingToBeSent ~= NewSetOfElemType
    and computes(pick(RemainingToBeSent, NextToSend),
            result(NextToSend2))
    and NextToSend2 in RemainingToBeSent
    and ReceivedSoFar subset TotalToBeSent
  imp ReceivedSoFar3 subset TotalToBeSent)
```

31 U:

This is the case where not(ReceivedSoFar3 = ReceivedSoFar add NextToSend2).

32 U: **replace ReceivedSoFar3;**

SimpleSend # 3 is awaiting the proof of lemmas remSubset, addSubset, and inSubset.

TRUE

33 U: **print proof;**

```
theorem SimpleSend # 3,      ReceivedSoFar subset TotalToBeSent
            and RemainingToBeSent subset TotalToBeSent
            and RemainingToBeSent ~= NewSetOfElemType
            and computes(pick(RemainingToBeSent, NextToSend),
                   result(NextToSend2))
            and computes(send(NextToSend2, ReceivedSoFar),
                   result(ReceivedSoFar3))
          imp   ReceivedSoFar3 subset TotalToBeSent
            and (RemainingToBeSent rem NextToSend2) subset TotalToBeSent
;
```

SimpleSend # 3 uses computespick%, computessend%, remSubset?, addSubset?, and inSubset?.

```
proof tree:
18:|  SimpleSend # 3
        apply  computespick
19:   13  apply  computessend
20:   14  put      s = RemainingToBeSent
            and it = NextToSend
            and it1 = NextToSend2
            and it' = NextToSend2
            and rec = ReceivedSoFar
            and rec1 = ReceivedSoFar3
22:   15  apply  remSubset
23:   17  put      s = RemainingToBeSent
            and s1 = TotalToBeSent
            and x = NextToSend2
24:   18  suppose   ReceivedSoFar3
               = ReceivedSoFar add NextToSend2
25:    yes:
       19  replace  ReceivedSoFar3
26:     21  apply  addSubset
27:     23  put      s = ReceivedSoFar
              and s1 = TotalToBeSent
              and x = NextToSend2
28:     24  apply  inSubset
29:     26  put      s  =  RemainingToBeSent
              and x  =  NextToSend2
              and s1  =  TotalToBeSent  {search}
29:     (proven!)
32:    no:{SimpleSend # 3}
       20   replace  ReceivedSoFar3
->      (proven!)
```

34 U:

We have finished off SimpleSend# 3 and have only the 3 lemmas to prove. The next command will pick one of the lemmas.

35 U: **next;**

```
Going to lemma remSubset.
remSubset is untried.

all s, s1, x
 (    s subset s1
  imp (s rem x) subset s1)
```

36 U: **invoke subset|all| ;**

```
all s, s1, x, x'' (some x'
```

```
 (       x' in s imp x' in s1
      and x" in (s rem x)
  imp x" in s1))
```

37 U: **put x'=x" ;**

```
all s, s1, x, x"
 (       ~(x" in s)
      and    x"
          in s rem x
  imp x" in s1)
```

38 U: **apply remEqv, i in (s rem x) eqv i in s and i~=x;**

Please declare i (then type ok; to continue).

39 (1) U:

We have forgotten to declare i, so the system asks us to declare it.

40 (1) U: **declare i:ElemType;**

i: ElemType

41 (1) U: **ok ;**

```
some i, s', x'
 (    ~(i in (s' rem x'))
   eqv i in s' imp i = x')
```

42 U: **search;**

```
1/4:     s' = s
     and i = x"
   1/1: x' = x
Proved by chaining and narrowing
using the substitution

   s' = s
 and i = x"
 and x' = x

TRUE
remSubset is awaiting the proof of lemma remEqv.
```

43 U: **print proof ;**

```
theorem remSubset,      s subset s1
            imp (s rem x) subset s1;
remSubset uses remEqv?.

proof tree:
36:|   remSubset
         invoke  subset  | all |
37:   28   put  x' = x"
41:   29   apply  remEqv
42:   31   put     s' = s
           and i  =  x"
           and x'  =  x  {search}
42:-> (proven!)
```

44 U:

We have to prove the lemma remEqv.

45 U: **next;**

Going to lemma remEqv.
remEqv is untried.

```
all i, s, x
 (    ~(i in (s rem x))
  eqv i in s imp i = x)
```

46 U: **employ Induction(s);**

```
Case NewSetOfElemType: Prop(NewSetOfElemType) proven.
Case add: all ss, ii (    IH(ss)
             imp Prop(ss add ii)) remains to be shown.
(add:)
all ss', ii', i, x
 (    IH(ss', 30 {remEqv})
  imp if i = ii'
      then    ~  i
             in if ii' = x
                then ss' rem x
                else (ss' rem x) add ii'
           eqv i = x
      else    ~  i
             in if ii' = x
                then ss' rem x
                else (ss' rem x) add ii'
           eqv    i in ss'
              imp i = x)
(The 'cases' command is applicable)
```

47 U:

We could have turned on AutoCases so we wouldn't have to explictly do the cases command.

48 U: **cases;**

```
all ss', ii', i, x
 (   IH(ss', 30 {remEqv})
  imp if i = ii'
      then        ii' = x
               and ~(i in (ss' rem x))
           eqv i = x
      else     ~(i in (ss' rem x))
           eqv     i in ss'
               imp i = x)
```

49 U: **invoke IH;**

```
all ss', ii', i, x (some i', x'
 (if i' in ss' imp i' = x'
    then    i' in (ss' rem x')
         or if i = ii'
             then         ii' = x
                      and ~(i in (ss' rem x))
                  eqv i = x
             else     ~(i in (ss' rem x))
                  eqv     i in ss'
                      imp i = x
    else     i' in (ss' rem x')
         imp if i = ii'
              then        ii' = x
                      and ~(i in (ss' rem x))
                   eqv i = x
              else     ~(i in (ss' rem x))
                   eqv     i in ss'
                       imp i = x))
```

50 U: **search ;**

```
1/4: i' = i
   1/2: x' = ii'
   2/2: x' = x
2/4:    x' = ii'
    and i' = i
3/4:    x' = x
```

```
    and i' = i
4/4:     x' = x
    and i' = ii'
Unsuccessful.
```

51 U:

search couldn't reduce the expression to true. So we can either use the put command or the choose command. Let's use choose.

52 U: **choose 1,2 ;**

```
1/4: i' = i
   2/2: x' = x
all ss', ii', i, x
 (if i in ss'
   then        i = x
          and ~  i
              in ss' rem x
          and i = ii'
       imp ii' = x
   else        ~  i
              in ss' rem x
          and i = ii'
       imp    ii' = x
          eqv i = x)
```

53 U: **replace i;**

```
remEqv proved.
remSubset proved.

TRUE
```

54 U: **print status;**

The untried theorems are addSubset, inSubset, pick # 1, and send # 1.
No theorems are tried.
The assumed theorems are computespick, computessend, and computesSimpleSend.
The awaiting lemma proof theorems are pick, send, SimpleSend, and SimpleSend # 3.
The proved theorems are remEqv, remSubset, SimpleSend # 1, and SimpleSend # 2.

55 U: **print proof ;**

```
theorem remEqv,     i in (s rem x)
          eqv    i in s
              and i ~= x;
```

proof tree:
46:! remEqv
    employ Induction(s)
-> NewSetOfElemType:
    Immediate
48: add:
    33 cases
49: 34 invoke IH
52: 35 put i' = i
        and x' = x {choose}
53: 36 replace i
-> (proven!)

56 U: **next ;**

There's more than one unproven ancestor. You may pick one of addSubset or inSubset.

57 U:

We still have 2 more lemmas to prove before our proof of SimpleSend# 3 will be completed.

58 U: **try addSubset;**

addSubset is untried.

all s, s1, x
(s subset s1 and x in s1
imp (s add x) subset s1)

59 U: **invoke subset|all| ;**

all s, s1, x, x'' (some x'
(x' in s imp x' in s1
and x in s1
and (x'' = x) or x'' in s
imp x'' in s1))

60 U: **put x'=x'' ;**

all s, s1, x, x''
(~(x'' in s)
and x in s1
and x'' = x
imp x'' in s1)

61 U: **replace x;**

addSubset proved.

TRUE

62 U: **print proof ;**

```
theorem addSubset,        s subset s1
                 and x in s1
             imp (s add x) subset s1;
```

```
proof tree:
59:!  addSubset
        invoke  subset  | all |
60:   37   put  x' = x''
61:   38   replace  x
->    (proven!)
```

63 U: **next ;**

Going to unproven ancestor inSubset.
inSubset is untried.

```
all x, s, s1
 (   x in s and s subset s1
  imp x in s1)
```

64 U:

The last lemma for SimpleSend#3.

65 U: **invoke subset;**

```
all x, s, s1 (some x'
 (      x in s
     and x' in s imp x' in s1
  imp x in s1))
```

66 U: **search ;**

1/1: x' = x
Proved by chaining and narrowing
using the substitution

x' = x

TRUE
inSubset proved.

SimpleSend # 3 proved.
Program SimpleSend verified!

67 U:

That's it, we have proved the unit SimpleSend.

68 U: **print proof;**

```
theorem inSubset,        x in s
               and s subset s1
            imp x in s1;

proof tree:
65:!  inSubset
        invoke  subset
66:   39  put x' = x {search}
66:-> (proven!)
```

69 U:

This print command will print any theorems that we have proven in this session.

70 U: **print proof theorems;**

```
theorem inSubset,        x in s
               and s subset s1
            imp x in s1;

proof tree:
65:!  inSubset
        invoke  subset
66:   39  put x' = x {search}
66:-> (proven!)


theorem addSubset,        s subset s1
               and x in s1
            imp (s add x) subset s1;

proof tree:
59:!  addSubset
        invoke  subset | all |
60:   37  put x' = x''
61:   38  replace x
     (proven!)
```

```
theorem remEqv,     i in (s rem x)
          eqv     i in s
             and i ~= x;

proof tree:
46:!  remEqv
        employ Induction(s)
     NewSetOfElemType:
       Immediate
48:    add:
       33   cases
49:      34   invoke IH
52:      35   put     i' = i
                and x' = x {choose}
53:      36   replace i
       (proven!)
```

```
theorem remSubset,     s subset s1
            imp (s rem x) subset s1;
remSubset uses remEqv!.

proof tree:
36:!  remSubset
        invoke  subset  | all |
37:   28   put  x' = x''
41:   29   apply remEqv
42:   31   put     s' = s
             and i = x''
             and x' = x {search}
42:   (proven!)
```

```
theorem SimpleSend # 3,         ReceivedSoFar subset TotalToBeSent
                and RemainingToBeSent subset TotalToBeSent
                and RemainingToBeSent ~ = NewSetOfElemType
                and computes(pick(RemainingToBeSent, NextToSend),
                        result(NextToSend2))
                and computes(send(NextToSend2, ReceivedSoFar),
                        result(ReceivedSoFar3))
             imp    ReceivedSoFar3 subset TotalToBeSent
                and (RemainingToBeSent rem NextToSend2) subset TotalToBeSent
```

;
SimpleSend # 3 uses computespick%, computessend%, remSubset!, addSubset!, and inSubset!.

proof tree:

```
18:!  SimpleSend # 3
        apply computespick
19:   13  apply computessend
20:   14  put   s = RemainingToBeSent
           and it = NextToSend
           and it1 = NextToSend2
           and it' = NextToSend2
           and rec = ReceivedSoFar
           and rec1 = ReceivedSoFar3
22:   15  apply remSubset
23:   17  put   s = RemainingToBeSent
           and s1 = TotalToBeSent
           and x = NextToSend2
24:   18  suppose  ReceivedSoFar3
             = ReceivedSoFar add NextToSend2
25:   yes:
       19  replace ReceivedSoFar3
26:     21  apply addSubset
27:     23  put   s = ReceivedSoFar
             and s1 = TotalToBeSent
             and x = NextToSend2
28:     24  apply inSubset
29:     26  put   s = RemainingToBeSent
             and x = NextToSend2
             and s1 = TotalToBeSent {search}
29:     (proven!)
32:   no:{SimpleSend # 3}
       20  replace ReceivedSoFar3
       (proven!)
```

theorem SimpleSend # 2,   NewSetOfElemType subset TotalToBeSent
          and TotalToBeSent subset TotalToBeSent;

proof tree:

```
15:!  SimpleSend # 2
        invoke  subset  | all |
15:   (proven!)
```

theorem SimpleSend # 1,   ReceivedSoFar2 subset TotalToBeSent

```
          and RemainingToBeSent1 subset TotalToBeSent
          and RemainingToBeSent1 = NewSetOfElemType
        imp ReceivedSoFar2 subset TotalToBeSent;

proof tree:
11:!  (proven!)



theorem computesSimpleSend,    computes(SimpleSend(TotalToBeSent, ReceivedSoFar),
                    result(ReceivedSoFar1))
           imp ReceivedSoFar1 subset TotalToBeSent;



theorem SimpleSend, verification(SimpleSend);
SimpleSend uses SimpleSend # 1!, SimpleSend # 2!, SimpleSend # 3!, and computesSimpleSend%.

proof tree:
8:!   SimpleSend

11:!   VC1:
       Immediate
15:!   VC2:
       SimpleSend # 2
          invoke  subset  | all |
15:     (proven!)
18:!   VC3:
       SimpleSend # 3
          apply  computespick
19:      13  apply  computessend
20:      14  put     s = RemainingToBeSent
               and it = NextToSend
               and it1 = NextToSend2
               and it' = NextToSend2
               and rec = ReceivedSoFar
               and rec1 = ReceivedSoFar3
22:      15  apply  remSubset
23:      17  put     s = RemainingToBeSent
               and s1 = TotalToBeSent
               and x = NextToSend2
24:      18  suppose   ReceivedSoFar3
                 = ReceivedSoFar add NextToSend2
25:       yes:
          19  replace  ReceivedSoFar3
26:         21  apply  addSubset
27:         23  put     s = ReceivedSoFar
```

```
            and s1 = TotalToBeSent
            and x = NextToSend2
28:      24  apply  inSubset
29:      26  put    s = RemainingToBeSent
            and x = NextToSend2
            and s1 = TotalToBeSent {search}
29:      (proven!)
32:    no:{SimpleSend, VC3:}
       20  replace  ReceivedSoFar3
       (proven!)
%    computes:{SimpleSend}
     computesSimpleSend
```

```
theorem computespick,       s ~= NewSetOfElemType
            and computes(pick(s, it),
                   result(it1))
         imp it1 in s;
```

```
theorem pick # 1,    s ~= NewSetOfElemType
       imp it in s;
```

```
theorem pick, verification(pick);
pick uses pick # 1? and computespick%.

proof tree:
9:|   pick
?     VC1:
      pick # 1

%     computes:
      computespick
```

```
theorem computessend,     computes(send(it, rec),
                   result(rec1))
         imp    rec1 = rec add it
```

```
                or rec1 = rec;


theorem send # 1,    rec  =  rec add it
            or rec = rec;


theorem send, verification(send);
send uses send # 1? and computessend%.

proof tree:
9:|   send
?      VC1:
        send # 1

%      computes:
        computessend
```

71 U: print status SimpleSend;

SimpleSend is proved.

72 U: quit;

Type CONTINUE to return to AFFIRM.

4. Proof of Rotate Twice

This property of the Rotate family is considerably more difficult to prove than many others, due to a tricky subsidiary deduction. This transcript is also supposed to illustrate a realistic proof attempt, instead of a polished proof. The transcript shows the proof of one branch of the proof, including many false starts. Notice the use of the profile entries. Unlike the other proofs in this volume, Rotate Twice shows a number of profile entries (including automechanisms such as AutoNext) turned <u>on</u>.

Transcript file <GERHART>AFFIRMTRANSCRIPT.8-NOV-80.2
is open in the AFFIRM system <AFFIRM>AFFIRM.EXE.120

1 U:

In this transcript, we will be proving a difficult property of a Rotate operation on sequences. The basis data type for sequences doesn't matter, so we will use a readily available one, for Integer.

2 U: **needs type sequenceofinteger;**

compiled for AFFIRM on 7-Nov-80 12:03:27
file created for AFFIRM on 7-Nov-80 12:03:18
SEQUENCEOFINTEGERCOMS
<GERHART>SEQUENCEOFINTEGER.COM.1

3 U:

the <u>needs</u> command found the type in my directory (normally it would be retrieved from <PVLIBRARY>). Here it is:

4 U: **print type sequenceofi$;**

(sequenceofi$ => SequenceOfInteger)
The $(Escape) denotes the rest of the type name.

type *SequenceOfInteger*;

declare dummy, ss, s, s1, s2: SequenceOfInteger;
declare k, ii, i, i1, i2, j: Integer;

interfaces NewSequenceOfInteger, s apr i, i apl s, seq(i), s1 join s2,
LessFirst(s), LessLast(s): SequenceOfInteger;

```
infix   join, apl, apr;

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

infix   in;

interfaces Length(s), First(s), Last(s): Integer;

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

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

axiom   seq(i) = = NewSequenceOfInteger apr i;

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

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

axiom   LessLast(s apr i) = = s;

axiom   isNew(s) = = (s = NewSequenceOfInteger);

axioms  i in NewSequenceOfInteger = = FALSE,
     i in (s apr i1) = = (i in s or (i = i1));

axioms  Length(NewSequenceOfInteger) = = 0,
     Length(s apr i) = = Length(s) + 1;

axiom   First(s apr i) = = if s = NewSequenceOfInteger
               then i
               else First(s);

axiom   Last(s apr i) = = i;

rulelemmas NewSequenceOfInteger = i apl s = = FALSE,
```

```
     i apl s = NewSequenceOfInteger = = FALSE;

rulelemmas s join (s1 apr i) = = (s join s1) apr i,
     s join NewSequenceOfInteger = = 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 = NewSequenceOfInteger
          then NewSequenceOfInteger
          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 = NewSequenceOfInteger
              then i
              else Last(s);

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

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

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

end {SequenceOfInteger};
```

5 U:

Since this is a demo transcript, somenbody may be interested in the profile.

6 U: **profile ;**

Do you want to Modify, See, or Read your profile? see [confirm]

your terminal characteristics:
  TerminalLineWidth is 92

the printout modes:
  LessOutputDesired is On
  AverageNameLength is 5
  AxiomGrouping is On
  LemmaGrouping is On
  InterfaceGrouping is On
  DefineGrouping is On
  NeedsGrouping is On
  NewPP is On
  SchemaGrouping is On
  UseORinProps is On
  DummyVarName is dummy
  ErrorTokensOutput is 15
  ReportFlag is Off
  RuleLHSPercentage is 49
  ShowNormint is Off
  ShowRuleSimplification is On
  ShowRules is Off
  TypesInInterfaces is Variables

the event history flags:
  HistoryWindowSize is 6

the executive flags:
  TextEditor is XED
  Timer is Off
  CompileOption is Redefine
  UsingTed is Off

the information about files:
  FreezeFileName is Frozen-AFFIRM
  LoadNeededTypes is On

the spelling correction flags:
  DontAskJustTake is 40

the theorem prover flags:

AutoResume is Off
ListAppliedExprs is On
AutoCases is On

the Interlisp mode flags:
GarbageCollectionMessage is Off
GarbageCollectionPages is 40
BreakAccess is Off

the transcript information:
AnnotatingTranscript is On
TranscriptFileName is AFFIRMTRANSCRIPT

the profile information:
SaveOnlyChangedEntries is On
UserProfileFileName is --AFFIRMUserProfile--
EnquireAfterFreeze is Off
EnquireInitially is Off
ReadAnotherProfileFile is On

the Knuth-Bendix flags:
CautiousCompletion is Off
KBAxiomOrder is Inverse

the automatically-applied event flags:
AutoAnnotate is On
AutoCompile is Off
AutoFix is Off
AutoFreeze is Ask
AutoGenvcs is Ask
AutoInfix is Off
AutoInvokeIH is Off
AutoNext is On
AutoNormint is On
AutoPrintProof is Ask
AutoPrintProofTheorems is Ask
AutoReplace is Off
AutoSave is Off
AutoSearch is Off
AutoSufficient is Off
AutoUndo is Off

NOW do you want to modify the current profile? no [confirm]

Do you wish to save this profile? no [confirm]

7 U:

Actually, I do want to modify the terminal line width to display better. The easy way is:

8 U: **profile terminalline$=$= =88;**

```
 " = "?
   TerminalLineWidth: 88
 = EVALQT

Unexpected PROFILEENTRYNAME found in routine NIL.
(undefined routine!)
Parameters: (none)
   EVALQT can't be set to 88.
New value:
(... profile aborted: effects undone)
```

9 U:

Well, that brought up AFFIRM's breakfast. I typed a bad character. Trying again.

10 U: **profile terminallinewidth=88;**

```
   TerminalLineWidth: 88
```

11 U:

Now I need an environment.

12 U: **adopt sequenceofinte$;**

```
(sequenceofinte$ => SequenceOfInteger)
New environment:
dummy, ss, s, s1, s2: SequenceOfInteger
k, ii, i, i1, i2, j: Integer
```

13 U:

The concept of interest is sequence rotation, both ways.

14 U: **define Rotate(s,i)= =if isNew(s) or i=0 then s**

(... command input aborted)

15 U: **define Rotate(s,i)= =if isNew(s) then s else if i=0 then s else if i<0 then Rotate(LessFirst(s) apr First(s), i+1) else Rotate(Last(s) apl LessLast(s), i-1);**

Please provide an interface declaration for Rotate (then type ok; to continue).

16 (1) U: **interface Rotate(s,i):SequenceOfInteger;**

17 (1) U: **ok;**

```
define Rotate(s, i)
      = = if (s = NewSequenceOfInteger) or (i = 0)
          then s
          else if i<0
               then Rotate(LessFirst(s) apr First(s), i + 1)
               else Rotate(Last(s) apl LessLast(s), i-1);
```

18 U:

Note the change in form, adding an 'or' for the first two conditionals.

19 U:

The property of interest is:

20 U: **try RTwice, Rotate(Rotate(s,i),j) = Rotate(s,i + j);**

Making node RTwice a theorem.
RTwice is untried.

all s, i, j (Rotate(Rotate(s, i), j) = Rotate(s, i + j))

21 U:

The usual approach is to dive into an induction on one of the integer parameters. Let's try i.

22 U: **employ Induction(i);**

Case 0: Prop(0) remains to be shown.
Case DIFFERENCE: all ii ((ii <= 0) and IH(ii) imp Prop(ii-1)) remains to be shown.
Case PLUS: all ii ((0 <= ii) and IH(ii) imp Prop(ii + 1)) remains to be shown.
(0:)
all s, j (Rotate(Rotate(s, 0), j) = Rotate(s, j))

23 U: **invoke Rotate|2|;**

TRUE
Going to leaf DIFFERENCE:.

all ii', s, j

```
 (   (ii' <= 0) and IH(ii', 1 {RTwice})
  imp  Rotate(Rotate(s, ii'-1), j)
     = Rotate(s, ii' + j - 1))
```

24 U: **invoke Rotate|2|;**

```
all ii', s, j
 (   (ii' <= 0) and IH(ii', 1 {RTwice})
  imp if s = NewSequenceOfInteger
      then Rotate(s, j) = Rotate(s, ii' + j - 1)
      else  Rotate(Rotate(LessFirst(s) apr First(s), ii'), j)
         = Rotate(s, ii' + j - 1))
```

25 U: **invoke Rotate|2,3|;**

```
all ii', s, j
 (   (ii' <= 0) and IH(ii', 1 {RTwice})
  imp if s = NewSequenceOfInteger
      then Rotate(s, j) = s
      else if   Rotate(LessFirst(s) apr First(s), ii') = NewSequenceOfInteger
             or j = 0
           then  Rotate(LessFirst(s) apr First(s), ii')
              = Rotate(s, ii' + j - 1)
           else if j<0
                then  Rotate(   LessFirst(Rotate(LessFirst(s) apr First(s)

                                  ii'))
                           apr First(Rotate(LessFirst(s) apr First(s),
                                  ii')),
                          j + 1)
                    = Rotate(s, ii' + j - 1)
                else  Rotate(   Last(Rotate(LessFirst(s) apr First(s),
                                  ii'))
                           apl LessLast(Rotate(LessFirst(s) apr First(s)

                                  ii')),
                          j-1)
                    = Rotate(s, ii' + j - 1))
```

26 U:

Did that wrong, wanted 1,2 Rotates.

27 U: **undo;**

invoke undone.

28 U: **invoke Rotate|1,2|;**

```
all ii', s, j
 (   (ii' <= 0) and IH(ii', 1 {RTwice})
  imp   s = NewSequenceOfInteger
      or  Rotate(Rotate(LessFirst(s) apr First(s), ii'), j)
         = Rotate(s, ii' + j - 1))
```

29 U:

Time to use the induction.

30 U: **invoke IH;**

```
all ii', s, j (some s', j'
 (       ii' <= 0
      and Rotate(Rotate(s', ii'), j') = Rotate(s', ii' + j')
  imp   s = NewSequenceOfInteger
      or  Rotate(Rotate(LessFirst(s) apr First(s), ii'), j)
         = Rotate(s, ii' + j - 1)))
```

31 U: **put s' = LessFirst(s) apr First(s) and j' = j;**

```
all ii', s, j
 (       ii' <= 0
      and  Rotate(Rotate(LessFirst(s) apr First(s), ii'),
               j)
          = Rotate(LessFirst(s) apr First(s), ii' + j)
  imp   s = NewSequenceOfInteger
      or  Rotate(Rotate(LessFirst(s) apr First(s), ii'),
              j)
         = Rotate(s, ii' + j - 1))
```

32 U: **replace;**

```
all ii', s, j
 (       ii' <= 0
      and  Rotate(Rotate(LessFirst(s) apr First(s), ii'),
               j)
          = Rotate(LessFirst(s) apr First(s), ii' + j)
  imp   s = NewSequenceOfInteger
      or  Rotate(LessFirst(s) apr First(s), ii' + j)
         = Rotate(s, ii' + j - 1))
```

33 U:

Now it's down to making the last Rotates match.

34 U: **invoke Rotate|-1|;**

```
all ii', s, j
 (      ii' <= 0
     and  Rotate(Rotate(LessFirst(s) apr First(s), ii'),
            j)
        = Rotate(LessFirst(s) apr First(s), ii' + j)
  imp    s = NewSequenceOfInteger
     or if ii' + j - 1 = 0
          then  Rotate(LessFirst(s) apr First(s),
                  ii' + j)
             = s
          else    ii' + j <= 0
              or  Rotate(LessFirst(s) apr First(s),
                    ii' + j)
                = Rotate(Last(s) apl LessLast(s),
                    ii' + j - 2))
```

35 U:

Let's delete the used equality. This is not recommended style, but we have never implemented the corresponding non-editor command.

36 U: **@;**

```
tty:
1*5 2 (delete 3)
ok
Please summarize what you did, end with ';'
deleted hypotheses;

all ii', s, j
 (   ii' <= 0
  imp    s = NewSequenceOfInteger
     or if ii' + j - 1 = 0
          then  Rotate(LessFirst(s) apr First(s),
                  ii' + j)
             = s
          else    ii' + j <= 0
              or  Rotate(LessFirst(s) apr First(s),
                    ii' + j)
                = Rotate(Last(s) apl LessLast(s),
                    ii' + j - 2))
```

37 U: **suppose isNew(s);**

```
(yes:)
TRUE
Going to leaf no:.

all ii', s, j
 (   (s ~= NewSequenceOfInteger) and (ii' <= 0)
  imp if ii' + j - 1 = 0
       then  Rotate(LessFirst(s) apr First(s),
                ii' + j)
            = s
       else   ii' + j <= 0
            or  Rotate(LessFirst(s) apr First(s),
                  ii' + j)
               = Rotate(Last(s) apl LessLast(s),
                  ii' + j - 2))
```

38 U:

I want to break up the cases and re-arrange the expression.

39 U: **split;**

```
(first:)
all ii', s, j
 (      (s ~= NewSequenceOfInteger) and (ii' <= 0)
     and ii' + j - 1 = 0
  imp  Rotate(LessFirst(s) apr First(s), ii' + j)
     = s)
```

40 U:

A rearrangement of the Integer expression is needed. This requires explicitly applying an "Integer Fact" lemma, as Affirm's Integer Simplifier doesn't handle this case.

41 U: **apply AddSwitch, i+j=k eqv (i=k-j and j=k-i);**

```
some i, k, j' (   i+j' = k
          eqv (i = k-j') and (j' = k-i))
```

42 U: **put i=ii'+j and j'=-1 and k=0;**

```
all ii', s, j
 (      ii' + j = 1
     and -1 = -(ii' + j)
     and ii' + j - 1 = 0
     and s ~= NewSequenceOfInteger
```

```
    and ii' <= 0
  imp  Rotate(LessFirst(s) apr First(s), ii' + j)
    = s)
```

43 U: **replace;**

```
all ii', s, j
 (      (ii' + j = 1) and (s ~= NewSequenceOfInteger)
     and ii' <= 0
  imp Rotate(LessFirst(s) apr First(s), 1) = s)
```

44 U: **invoke Rotate;**

```
all ii', s, j
 (      (ii' + j = 1) and (s ~= NewSequenceOfInteger)
     and ii' <= 0
  imp Rotate(First(s) apl LessFirst(s), 0) = s)
```

45 U: **invoke Rotate;**

```
all ii', s, j
 (      (ii' + j = 1) and (s ~= NewSequenceOfInteger)
     and ii' <= 0
  imp First(s) apl LessFirst(s) = s)
```

46 U:

And that's a normal form property.

47 U: **employ NormalForm(s);**

```
Case NewSequenceOfInteger: Prop(NewSequenceOfInteger) proven.
Case apr: all ss, ii (Prop(ss apr ii)) remains to be shown.
(apr:)
all ss', ii', ii, j
 (   (ii + j = 1) and (ii <= 0)
  imp (ss' = NewSequenceOfInteger) or (First(ss') apl LessFirst(ss') = ss'))
```

48 U:

Whoops, it requires Induction, so it should be a lemma.

49 U: **undo;**

employ undone.

50 U: **apply FirstSplit, ~isNew(s) imp First(s) apl LessFirst(s)=s;**

```
some s' ((s' = NewSequenceOfInteger) or (First(s') apl LessFirst(s') = s'))
```

51 U: **search;**

```
1/1: s' = s
Proved by chaining and narrowing
using the substitution

s' = s

TRUE
Going to leaf second:.

all ii', s, j
 (   (s ~ = NewSequenceOfInteger) and (ii' <= 0)
  imp   ii' + j - 1 = 0
      or ii' + j <= 0
      or  Rotate(LessFirst(s) apr First(s), ii' + j)
        = Rotate(Last(s) apl LessLast(s),
              ii' + j - 2))
```

52 U:

The integer terms show that ii'+j>=2 in the Rotate equality, which will allow invoking it through to match. First, let's draw out the fact.

53 U: **suppose ii'+j>=2;**

```
(yes:)
all ii', s, j
 (      (2 <= ii' + j) and (s ~ = NewSequenceOfInteger)
      and ii' <= 0
  imp   ii' + j - 1 = 0
      or  Rotate(LessFirst(s) apr First(s), ii' + j)
        = Rotate(Last(s) apl LessLast(s),
              ii' + j - 2))
```

54 U: **invoke Rotate;**

```
all ii', s, j
 (      (2 <= ii' + j) and (s ~ = NewSequenceOfInteger)
      and ii' <= 0
  imp   ii' + j - 1 = 0
      or  Rotate(First(s) apl LessFirst(s),
```

```
            ii' + j - 1)
        = Rotate(Last(s) apl LessLast(s),
            ii' + j - 2))
```

55 U: **invoke Rotate;**

```
all ii', s, j
 (      (2 <= ii' + j) and (s ~ = NewSequenceOfInteger)
       and ii' <= 0
   imp    ii' + j - 1 = 0
       or if LessFirst(s) = NewSequenceOfInteger
           then  Rotate(NewSequenceOfInteger apr First(s),
                    ii' + j - 2)
                = Rotate(Last(s) apl LessLast(s),
                    ii' + j - 2)
           else  Rotate(    Last(LessFirst(s))
                    apl First(s) apl LessLast(LessFirst(s)),
                    ii' + j - 2)
                = Rotate(Last(s) apl LessLast(s),
                    ii' + j - 2))
```

56 U:

All these selections should simplify with NormalForm.

57 U: **employ NormalForm(s);**

Case NewSequenceOfInteger: Prop(NewSequenceOfInteger) proven.
Case apr: all ss, ii (Prop(ss apr ii)) remains to be shown.

```
 (apr:)
all ss', ii', ii, j
 (   (2 <= ii + j) and (ii <= 0)
   imp    ii + j - 1 = 0
       or if ss' = NewSequenceOfInteger
           then  Rotate(NewSequenceOfInteger apr ii',
                    ii + j - 2)
                = Rotate(ii' apl ss', ii + j - 2)
           else  Rotate(ii' apl (First(ss') apl LessFirst(ss')),
                    ii + j - 2)
                = Rotate(ii' apl ss', ii + j - 2))
```

58 U: **replace;**

```
all ss', ii', ii, j
 (   (2 <= ii + j) and (ii <= 0)
   imp    (ii + j - 1 = 0) or (ss' = NewSequenceOfInteger)
       or  Rotate(ii' apl (First(ss') apl LessFirst(ss')),
```

```
        ii + j - 2)
      = Rotate(ii' apl ss', ii + j - 2))
```

59 U: **apply FirstSplit;**

```
some s ((s = NewSequenceOfInteger) or (First(s) apl LessFirst(s) = s))
```

60 U: **put s=ss';**

```
all ss', ii', ii, j
 (      ss' ~= NewSequenceOfInteger
     and First(ss') apl LessFirst(ss') = ss'
     and 2 <= ii + j
     and ii <= 0
  imp   ii + j - 1 = 0
     or  Rotate(ii' apl (First(ss') apl LessFirst(ss')),
          ii + j - 2)
       = Rotate(ii' apl ss', ii + j - 2))
```

61 U: **replace;**

```
TRUE
Going to leaf no:.

all ii', s, j
 (      (ii' + j < 2) and (s ~= NewSequenceOfInteger)
     and ii' <= 0
  imp   ii' + j - 1 = 0
     or ii' + j <= 0
     or  Rotate(LessFirst(s) apr First(s), ii' + j)
       = Rotate(Last(s) apl LessLast(s),
          ii' + j - 2))
```

62 U: **apply AddSwitch;**

```
some i, k, j' (    i + j' = k
        eqv (i = k-j') and (j' = k-i))
```

63 U: **put i=ii'+j and j'=-1 and k=0;**

```
all ii', s, j
 (if ii' + j = 1
   then       -1 ~= -(ii' + j)
         and ii' + j - 1 ~= 0
```

```
          and ii' + j < 2
          and s ~ = NewSequenceOfInteger
          and ii' <= 0
      imp  Rotate(LessFirst(s) apr First(s),
               ii' + j)
          = Rotate(Last(s) apl LessLast(s),
               ii' + j - 2)
  else        ii' + j - 1 ~ = 0
          and ii' + j < 2
          and s ~ = NewSequenceOfInteger
          and ii' <= 0
      imp   ii' + j <= 0
          or  Rotate(LessFirst(s) apr First(s),
                 ii' + j)
            = Rotate(Last(s) apl LessLast(s),
                 ii' + j - 2))
```

64 U: **replace;**

```
TRUE
Going to leaf PLUS:.

all ii', s, j
 (   (0 <= ii') and IH(ii', 1 {RTwice})
  imp  Rotate(Rotate(s, ii' + 1), j)
     = Rotate(s, ii' + j + 1))
```

65 U:

Now we have to do the same thing on this side.

66 U: **print proof;**

```
theorem RTwice, Rotate(Rotate(s, i), j) = Rotate(s, i + j);
RTwice uses FirstSplit? and AddSwitch?.

proof tree:
22:|  RTwice
        employ Induction(i)
23:    0: 2   invoke Rotate | 2 |
23:      (proven!)
24:    DIFFERENCE:
       3   invoke Rotate | 2 |
24:      6   cases
28:      7   invoke Rotate | 1 , 2 |
30:      8   invoke IH
31:      9   put (s' = LessFirst(s) apr First(s)) and (j' = j)
32:      10  replace
```

```
34:     11  invoke Rotate | - 1 |
34:     12  cases
36:     13  @ {deleted hypotheses}
37:     14  suppose isNew(s)
37:      yes:
         Immediate
39:      no:16  split
41:        first:
           17  apply AddSwitch
42:          20  put    i = ii'+j
                    and (j' = -1) and (k=0)
43:          21  replace
44:          22  invoke Rotate
45:          23  invoke Rotate
50:          24  apply FirstSplit
51:          26  put s' = s {search}
51:          (proven!)
53:        second:{RTwice, DIFFERENCE:, no:}
           18  suppose ii' + j >= 2
54:          yes:
             28  invoke Rotate
55:            30  invoke Rotate
55:            31  cases
57:            32  employ NormalForm(s)
              NewSequenceOfInteger:
               Immediate
57:            apr:
               33  cases
58:              34  replace
59:              35  apply FirstSplit
60:              36  put s=ss'
61:              37  replace
               (proven!)
62:          no:{RTwice, DIFFERENCE:, no:, second:}
             29  apply AddSwitch
63:            38  put    i = ii'+j
                    and (j' = -1) and (k=0)
64:            39  replace
             (proven!)
?->   PLUS:{RTwice}
      4
```

----

theorem FirstSplit, ~isNew(s) imp First(s) apl LessFirst(s) = s;

theorem AddSwitch,   i+j = k
          eqv (i = k-j) and (j = k-i);

----

67 U: **assume Addswitch;**

(Addswitch => AddSwitch)

68 U: **try firstsplit;**

(firstsplit => FirstSplit)
FirstSplit is untried.

all s ((s = NewSequenceOfInteger) or (First(s) apl LessFirst(s) = s))

69 U: **employ Induction(s);**

Case NewSequenceOfInteger: Prop(NewSequenceOfInteger) proven.
Case apr: all ss, ii (IH(ss) imp Prop(ss apr ii)) remains to be shown.
(apr:)
all ss', ii' (IH(ss', 25 {FirstSplit})
imp (ss' = NewSequenceOfInteger) or (First(ss') apl LessFirst(ss') = ss')
)

70 U: **invoke IH;**

TRUE
FirstSplit proved.
Automatically print the proof of FirstSplit? no [confirm]

Appendix I
Types Used in Proofs

I.1. SetOfElemType

```
type SetOfElemType;

needs type ElemType;

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

interfaces NewSetOfElemType, s add x, s rem i, s diff s1, s int s1, s union s1
      : SetOfElemType;

infix   union, diff, int, rem, add;

interfaces i in s, isNewSetOfElemType(s), s subset s1, Induction(s), NormalForm(s)
      : Boolean;

infix   subset, in;

axioms   reflexive = reflexive = = TRUE,
      NewSetOfElemType = s add i = = FALSE,
      s add i = NewSetOfElemType = = FALSE;

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

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

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

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

axiom   isNewSetOfElemType(s) = = (s = NewSetOfElemType);

define  s = s1 = = (s subset s1 and s1 subset s),

    s subset s1
      = = all x' (x' in s imp x' in s1);

schemas 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} ;
```

I.2. SequenceOfInteger

type *SequenceOfInteger*;

declare dummy, ss, s, s1, s2: SequenceOfInteger;
declare k, ii, i, i1, i2, j: Integer;

interfaces NewSequenceOfInteger, s apr i, i apl s, seq(i), s1 join s2, LessFirst(s),
LessLast(s), dedup(s), reverse(s), Rotate(s, k), Initial(s, k),
LessInitial(s, k), deletepth(s, k), seqrange(i, j), sequpto(i)
: SequenceOfInteger;

infix join, apl, apr;

interfaces isNewSequenceOfInteger(s), s1 subseq s2, FirstInduction(s), Induction(s),
NormalForm(s), i in s, nodups(s), disjoint(s1, s2): Boolean;

infix in, subseq;

interfaces Length(s), First(s), Last(s), pth(s, k): Integer;

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

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

axiom seq(i) = = NewSequenceOfInteger apr i;

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

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

axiom LessLast(s apr i) = = s;

axioms dedup(NewSequenceOfInteger) = = NewSequenceOfInteger,

```
        dedup(s apr i)
          == if i in s
             then dedup(s)
             else dedup(s) apr i;

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

axiom   isNewSequenceOfInteger(s) == (s = NewSequenceOfInteger);

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

axioms  i in NewSequenceOfInteger == FALSE,
        i in (s apr i1) == (i in s or (i = i1));

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

axioms  disjoint(NewSequenceOfInteger, s) == TRUE,
        disjoint(s apr i, s1) == (disjoint(s, s1) and ~(i in s1));

axioms  Length(NewSequenceOfInteger) == 0,
        Length(s apr i) == Length(s) + 1;

axiom   First(s apr i) == if s = NewSequenceOfInteger
                          then i
                          else First(s);

axiom   Last(s apr i) == i;

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

rulelemmas s join (s1 apr i) == (s join s1) apr i,
        s join NewSequenceOfInteger == 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 = NewSequenceOfInteger
```

```
                    then NewSequenceOfInteger
                    else i apl LessLast(s);

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

rulelemma nodups(i apl s) = = (nodups(s) and ~(i in s));

rulelemma First(i apl s) = = i;

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

define  Rotate(s, k)
      = = if (s = NewSequenceOfInteger) or (k = 0)
          then s
          else Rotate(LessFirst(s) apr First(s), k-1),

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

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

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

    seqrange(i, j)
      = = if j<i
          then NewSequenceOfInteger
          else seqrange(i, j-1) apr j,

    pth(s, k)
      = = if k = 1
          then First(s)
          else pth(LessFirst(s), k-1),

    sequpto(i) = = seqrange(1, i);

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

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

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

end {*SequenceOfInteger*} ;

References

[Musser 80] Musser, D. R., "On proving inductive properties of abstract data types," in *Proceedings of the Seventh ACM Symposium on Principles of Programming Languages*, ACM SIGPLAN, 1980.