Lab 4

EECS4312

September 9, 2015

Table of contents

Learning Outcomes

Topics

- In this Lab, you learn to specify systems using sets and functions.
- You apply your knowledge to the specification and validation of a phone system.

top.pvs

Develop 3 Theories

You must specify and prove three theories as shown in the `top.pvs` file below:

```
% Exercises for Lab4
% proveit --importchain --clean top.pvs
top : THEORY
BEGIN
   IMPORTING set_ex          % set exercises
   IMPORTING function_ex     % function exercises
   IMPORTING phone           % Phone Specification
END top
```

Preparation

Details are provided in the rest of these slides. Read them before coming to the Lab.

top.pvs

Develop 3 Theories

You must specify and prove three theories as shown in the `top.pvs` file below:

```
% Exercises for Lab4
% proveit --importchain --clean top.pvs
top : THEORY
BEGIN
   IMPORTING set_ex        % set exercises
   IMPORTING function_ex   % function exercises
   IMPORTING phone         % Phone Specification
END top
```

Preparation

Details are provided in the rest of these slides. Read them before coming to the Lab.

Result of running proveit on top.pvs


```
Proof summary for theory top
   Theory totals: 0 formulas, 0 attempted, 0 succeeded (0.00 s)

Proof summary for theory set_ex
   conj1...............................proved - complete   [shostak](0.02 s)
   conj2...............................proved - complete   [shostak](0.00 s)
   conj3...............................proved - complete   [shostak](0.05 s)
   conj4...............................proved - complete   [shostak](0.05 s)
   conj5...............................proved - complete   [shostak](0.03 s)
   conj6...............................proved - complete   [shostak](0.03 s)
   conj7...............................proved - complete   [shostak](0.05 s)
   conj8...............................proved - complete   [shostak](0.04 s)
   Theory totals: 8 formulas, 8 attempted, 8 succeeded (0.26 s)

Proof summary for theory function_ex
   check1_TCC1.........................proved - complete   [shostak](0.01 s)
   check1..............................proved - complete   [shostak](0.04 s)
   Theory totals: 2 formulas, 2 attempted, 2 succeeded (0.05 s)

Proof summary for theory phone
   FindAdd.............................proved - complete   [shostak](0.01 s)
   DelAdd..............................proved - complete   [shostak](0.05 s)
   Theory totals: 2 formulas, 2 attempted, 2 succeeded (0.05 s)

Grand Totals: 12 proofs, 12 attempted, 12 succeeded (0.37 s)
```

Submit your work 1

- Remove all the pdf files from your Lab directory. You must submit only the pvs files ***.pvs** and ***.prf**. Delete everything else.
- Rename the directory with your PVS files to `4312-lab4`, and then run the following command in the directory:
- `proveit --importchain --clean top.pvs`
- All theorems must be proven.

Submit your work 2

- Now submit your `4312-lab4` directory:
  `> submit 4312 lab4 4312-lab4`
- You will get confirmation of your submission.
- Ensure that you follow the instructions (and naming conventions) carefully and precisely to ensure that your submission can be checked.
- To obtain a grade on your quiz, you must complete and submit this Lab according to the instructions.

Preparation: PVS Functions and Set Theory

Read and Study:

- Study the *phone* specification in WIFT-95, especially the version using sets p16–20.
- Study functions and set theory in PVS (see the rest of these slides).
- See set theory in the attached slides `PVS-collection-Types.pdf`, especially slides 16–37
- You will need the concept of set equality (extensionality) on slide28.
- You must specify and prove the phone specification in Fig. 3, page 17 of the WIFT-95 tutorial. Call this theory *phone* (not *phone_3*)

Preparation: PVS Functions and Set Theory

Read and Study:

- Study the *phone* specification in WIFT-95, especially the version using sets p16–20.
- Study functions and set theory in PVS (see the rest of these slides).
- See set theory in the attached slides `PVS-collection-Types.pdf`, especially slides 16–37
- You will need the concept of set equality (extensionality) on slide28.
- You must specify and prove the phone specification in Fig. 3, page 17 of the WIFT-95 tutorial. Call this theory *phone* (not *phone_3*)

Preparation: PVS Functions and Set Theory

Read and Study:

- Study the *phone* specification in WIFT-95, especially the version using sets p16–20.
- Study functions and set theory in PVS (see the rest of these slides).
- See set theory in the attached slides `PVS-collection-Types.pdf`, especially slides 16–37
- You will need the concept of set equality (extensionality) on slide28.
- You must specify and prove the phone specification in Fig. 3, page 17 of the WIFT-95 tutorial.
  Call this theory *phone* (not *phone_3*)

Preparation: PVS Functions and Set Theory

Read and Study:

- Study the *phone* specification in WIFT-95, especially the version using sets p16–20.
- Study functions and set theory in PVS (see the rest of these slides).
- See set theory in the attached slides `PVS-collection-Types.pdf`, especially slides 16–37
- You will need the concept of set equality (extensionality) on slide28.
- You must specify and prove the phone specification in Fig. 3, page 17 of the WIFT-95 tutorial.
  Call this theory *phone* (not *phone_3*)

Preparation: PVS Functions and Set Theory

Read and Study:

- Study the *phone* specification in WIFT-95, especially the version using sets p16–20.
- Study functions and set theory in PVS (see the rest of these slides).
- See set theory in the attached slides `PVS-collection-Types.pdf`, especially slides 16–37
- You will need the concept of set equality (extensionality) on slide28.
- You must specify and prove the phone specification in Fig. 3, page 17 of the WIFT-95 tutorial. Call this theory *phone* (not *phone_3*)

Using sets in PVS

- Sets are defined in the PVS prelude (`M-x vpf`)
- Some of the operations defined on sets are:

PVS Name	traditional notation or meaning
`member`	$\in$
`union`	$\cup$
`intersection`	$\cap$
`difference`	$\setminus$
`add`	add element to a set
`singleton`	constructs set with one element
`subset?`	$\subseteq$
`strict_subset?`	$\subset$
`emptyset`	$\emptyset$

Read more:

`PVS-collection-Types.pdf`

Using sets in PVS

- Sets are defined in the PVS prelude (`M-x vpf`)
- Some of the operations defined on sets are:

PVS Name	traditional notation or meaning
`member`	$\in$
`union`	$\cup$
`intersection`	$\cap$
`difference`	$\setminus$
`add`	add element to a set
`singleton`	constructs set with one element
`subset?`	$\subseteq$
`strict_subset?`	$\subset$
`emptyset`	$\emptyset$

Read more:

`PVS-collection-Types.pdf`

Using sets in PVS


```
set_ex : THEORY
  %% Using the theory set[T] from the prelude
  %% set[T] is the same as setof[T].
BEGIN
  RESOURCE: TYPE = {r1, r2, r3, r4}

  UNIVERSE: set[RESOURCE] =
    {r: RESOURCE| True}

  SET1: set[RESOURCE] =
    {r: RESOURCE| r = r1 OR r = r2}

  SET2: set[RESOURCE] =
    {r: RESOURCE| r = r3 OR r = r4}

  SET3: set[RESOURCE] =
    {r: RESOURCE| r = r1 OR r = r2 OR r = r3}

  conj1: CONJECTURE
    member(r1, SET1)

  conj2: CONJECTURE  % same as conj1
    SET1(r1)
```

```
  conj3: CONJECTURE  % same as conj1
    add(r4, SET3) = UNIVERSE


  conj4: CONJECTURE  % same as conj1
    remove(r3, SET3) = SET1

  conj5: CONJECTURE
    UNIVERSE = union(SET1, SET2)

  conj6: CONJECTURE
    intersection(SET1, SET2) = emptyset

  conj7: CONJECTURE
    intersection(SET2, SET3) = singleton(r3)

 conj8: CONJECTURE
    remove(r3, add(r3,SET1)) = SET1

END set_ex
```

What you must do

Create a file `set_ex.pvs` and prove the conjectures shown above in the `set_ex` theory.

Using sets in PVS

```
set_ex : THEORY
  %% Using the theory set[T] from the prelude
  %% set[T] is the same as setof[T].
BEGIN
  RESOURCE: TYPE = {r1, r2, r3, r4}

  UNIVERSE: set[RESOURCE] =
    {r: RESOURCE| True}

  SET1: set[RESOURCE] =
    {r: RESOURCE| r = r1 OR r = r2}

  SET2: set[RESOURCE] =
    {r: RESOURCE| r = r3 OR r = r4}

  SET3: set[RESOURCE] =
    {r: RESOURCE| r = r1 OR r = r2 OR r = r3}

  conj1: CONJECTURE
    member(r1, SET1)

  conj2: CONJECTURE  % same as conj1
    SET1(r1)
```

```
  conj3: CONJECTURE  % same as conj1
    add(r4, SET3) = UNIVERSE


  conj4: CONJECTURE  % same as conj1
    remove(r3, SET3) = SET1

  conj5: CONJECTURE
    UNIVERSE = union(SET1, SET2)

  conj6: CONJECTURE
    intersection(SET1, SET2) = emptyset

  conj7: CONJECTURE
    intersection(SET2, SET3) = singleton(r3)

 conj8: CONJECTURE
    remove(r3, add(r3,SET1)) = SET1

END set_ex
```

What you must do

Create a file `set_ex.pvs` and prove the conjectures shown above in the `set_ex` theory.

From PVS — can generate Latex Mathematics

set_ex: THEORY
BEGIN

$\text{RESOURCE}: \text{TYPE} = \{r_1, r_2, r_3, r_4\}$

$\text{UNIVERSE}: \text{set}[\text{RESOURCE}] = \{r: \text{RESOURCE} \mid \text{TRUE}\}$

$\text{SET1}: \text{set}[\text{RESOURCE}] = \{r: \text{RESOURCE} \mid r = r_1 \lor r = r_2\}$

$\text{SET2}: \text{set}[\text{RESOURCE}] = \{r: \text{RESOURCE} \mid r = r_3 \lor r = r_4\}$

$\text{SET3}: \text{set}[\text{RESOURCE}] =$
$\quad \{r: \text{RESOURCE} \mid r = r_1 \lor r = r_2 \lor r = r_3\}$

$\text{conj1}: \text{CONJECTURE } (r_1 \in \text{SET1})$

$\text{conj2}: \text{CONJECTURE } \text{SET1}(r_1)$

$\text{conj3}: \text{CONJECTURE } (\text{SET3} \cup \{r_4\}) = \text{UNIVERSE}$

$\text{conj4}: \text{CONJECTURE } (\text{SET3} \setminus \{r_3\}) = \text{SET1}$

$\text{conj5}: \text{CONJECTURE } \text{UNIVERSE} = (\text{SET1} \cup \text{SET2})$

$\text{conj6}: \text{CONJECTURE } (\text{SET1} \cap \text{SET2}) = \emptyset$

$\text{conj7}: \text{CONJECTURE } (\text{SET2} \cap \text{SET3}) = \text{singleton}(r_3)$

$\text{conj8}: \text{CONJECTURE } ((\text{SET1} \cup \{r_3\}) \setminus \{r_3\}) = \text{SET1}$

END set_ex

Specifications using Functions: `function_ex.pvs`


```
function_ex : THEORY
BEGIN
  n: posnat
  DOMAIN : TYPE = {d : posnat | d <= n}
  d: VAR DOMAIN
  RANGE: TYPE = {a,b,c}

  % This function is the same as f1 below
  f: [DOMAIN -> RANGE] =
    (LAMBDA d: (IF d = 1 then a ELSE c ENDIF))

  f1(d): RANGE =
    (IF d = 1 then a ELSE c ENDIF)

  % A slightly different function
  f2(d): RANGE =
  (IF d = 1 then a ELSIF d=2 THEN b ELSE c ENDIF)

  % n >= 2 needed for type correctness
  % First try without the antecedent to see what happens
  check1: CONJECTURE
    n >= 2 IMPLIES f2 = (f1 WITH [ 2 := b])

END function_ex
```

Lambda Function: f same as f_1


```
function1: THEORY  BEGIN
  n: posnat
  DOMAIN : TYPE = {d : posnat | d <= n}
  d: VAR DOMAIN
  RANGE:   TYPE = {a,b,c}

  f: [DOMAIN -> RANGE] =
    (LAMBDA d:  (IF d = 1 then a ELSE c ENDIF))

  f1(d):RANGE =
   (IF d = 1 then a ELSE c ENDIF)

  % A slightly different function
  f2(d):RANGE =
   (IF d = 1 then a ELSIF d=2 THEN b ELSE c ENDIF)
   ...
```

Lambda Function: f same as f_1


```
function1: THEORY  BEGIN
  n: posnat
  DOMAIN : TYPE = {d : posnat | d <= n}
  d: VAR DOMAIN
  RANGE:   TYPE = {a,b,c}

  f: [DOMAIN -> RANGE] =
    (LAMBDA d:  (IF d = 1 then a ELSE c ENDIF))

  f1(d):RANGE =
   (IF d = 1 then a ELSE c ENDIF)

  % A slightly different function
  f2(d):RANGE =
   (IF d = 1 then a ELSIF d=2 THEN b ELSE c ENDIF)
   ...
```

Function Override: “WITH”

Functions, tuples, and records may be modified by means of the override expression.

```
f1(d):RANGE =
 (IF d = 1 then a ELSE c ENDIF)

f2(d):RANGE =
 (IF d = 1 then a ELSIF d=2 THEN b ELSE c ENDIF)

check1: CONJECTURE  f2 = (f1 WITH [ 2 := b])
```

Definition

```
(f1 WITH [ 2 := b])
=
(LAMBDA d: IF d = 2 THEN b ELSE f1(d) ENDIF)
```

Function Override: "WITH"

Functions, tuples, and records may be modified by means of the override expression.

```
f1(d):RANGE =
 (IF d = 1 then a ELSE c ENDIF)

f2(d):RANGE =
 (IF d = 1 then a ELSIF d=2 THEN b ELSE c ENDIF)

check1: CONJECTURE  f2 = (f1 WITH [ 2 := b])
```

Definition

```
(f1 WITH [ 2 := b])
=
(LAMBDA d: IF d = 2 THEN b ELSE f1(d) ENDIF)
```

Override Definition for use with Extension

PVS

```
DOMAIN: TYPE = 1..n
RANGE:  TYPE = {a,b,c}

f(d:DOMAIN):RANGE =
   (IF d = 1 then a ELSE c ENDIF)
```

In ordinary Mathematics we might write:

$$\texttt{(f WITH [2 := b])}(d_1) = \begin{cases} b & \text{if } d_1 = 2 \\ f(d_1) & \text{if } d_1 \neq 2 \end{cases}$$

Override Definition for use with Extension

PVS

```
DOMAIN: TYPE = 1..n
RANGE:  TYPE = {a,b,c}

f(d:DOMAIN):RANGE =
   (IF d = 1 then a ELSE c ENDIF)
```

In ordinary Mathematics we might write:

$$\texttt{(f WITH [2 := b])}\,(d_1) = \begin{cases} b & \text{if } d_1 = 2 \\ f(d_1) & \text{if } d_1 \neq 2 \end{cases}$$

Extensionality and sets


```
A: VAR set[real]
B: VAR set[real]
...

check1: CONJECTURE A = B
```

How to prove?

(apply_extensionality)

$$A = B \;\equiv\; x \in A \equiv x \in B$$

Extensionality and sets


```
A: VAR set[real]
B: VAR set[real]
...

check1: CONJECTURE A = B
```

How to prove?

(apply_extensionality)

$$A = B \;\equiv\; x \in A \equiv x \in B$$

Extensionality

In logic, **extensionality**, or extensional equality refers to principles that judge objects to be equal if they have the same external properties. It stands in contrast to the concept of **intensionality**, which is concerned with whether the internal definitions of objects are the same.

Example

f(n) = (n+5)*2
g(n) = 2*n + 10

These functions are *extensionally* equal; given the same input, both functions always produce the same value. But the definitions of the functions are not equal, and in that *intensional* sense the functions are not the same.

Extensionality

Example

f(n) = (n+5)*2
g(n) = 2*n + 10

How to prove?

f =g

(apply_extensionality)

f(x) =g(x), for any x

What next? Then expand f and g!

Extensionality

Example

f(n) = (n+5)*2
g(n) = 2*n + 10

How to prove?

f =g

(apply_extensionality)

f(x) =g(x), for any x

What next? Then expand f and g!

Extensionality

Example

f(n) = (n+5)*2
g(n) = 2*n + 10

How to prove?

f =g

(apply_extensionality)

f(x) =g(x), for any x

What next? Then expand f and g!

Function Override: “WITH”

```
f1(d):RANGE =
 (IF d = 1 then a ELSE c ENDIF)

f2(d):RANGE =
 (IF d = 1 then a ELSIF d=2 THEN b ELSE c ENDIF)

check1: CONJECTURE  f2 = (f1 WITH [ 2 := b])
```

How to prove?

(apply_extensionality)

Function Override: “WITH”


```
f1(d):RANGE =
 (IF d = 1 then a ELSE c ENDIF)

f2(d):RANGE =
 (IF d = 1 then a ELSIF d=2 THEN b ELSE c ENDIF)

check1: CONJECTURE  f2 = (f1 WITH [ 2 := b])
```

How to prove?

(apply_extensionality)

IF-THEN

IF(A,B,C) = (IF A THEN B ELSE C ENDIF)

How to prove?

Branching structure is typically expressed using the IF-connective or the CASES construct.

Since these IF and CASES branches could occur embedded within a formula, it must be lifted to the top level of the formula where the propositional simplification steps can be applied.

The (LIFT-IF) rule lifts the leftmost-innermost contiguous IF or CASES branching structure out to the top level.

(lift-if)

```
f( IF(A,B, IF(C,D,E) ) )
becomes
IF A THEN f(B) ELSE IF(C,f(D),f(E)) ENDIF
```

IF-THEN

IF(A,B,C) = (IF A THEN B ELSE C ENDIF)

How to prove?

Branching structure is typically expressed using the IF-connective or the CASES construct.

Since these IF and CASES branches could occur embedded within a formula, it must be lifted to the top level of the formula where the propositional simplification steps can be applied.

The (LIFT-IF) rule lifts the leftmost-innermost contiguous IF or CASES branching structure out to the top level.

(lift-if)

```
f( IF(A,B, IF(C,D,E) ) )
becomes
IF A THEN f(B) ELSE IF(C,f(D),f(E)) ENDIF
```

IF-THEN

IF(A,B,C) = (IF A THEN B ELSE C ENDIF)

How to prove?

Branching structure is typically expressed using the IF-connective or the CASES construct.

Since these IF and CASES branches could occur embedded within a formula, it must be lifted to the top level of the formula where the propositional simplification steps can be applied.

The (LIFT-IF) rule lifts the leftmost-innermost contiguous IF or CASES branching structure out to the top level.

(lift-if)

```
f( IF(A,B, IF(C,D,E) ) )
becomes
IF A THEN f(B) ELSE IF(C,f(D),f(E)) ENDIF
```

Function Override: "WITH"


```
    f2(x) = (f1 WITH [(2) := b])(x)
= <defn>
   f2(x) = (f1 WITH [(2) := b])(x)
= <LIFT-IF>
    IF x = 2 THEN f2(x) = b
           ELSE f2(x) = f1(x) ENDIF
```

TCC with Function Conjecture


```
function1: THEORY BEGIN
  n: posnat
  DOMAIN : TYPE = d : posnat | d <= n
  d: VAR DOMAIN
  RANGE: TYPE = a,b,c
  f1(d):RANGE =
   (IF d = 1 then a ELSE c ENDIF)
  f2(d):RANGE =
   (IF d = 1 then a ELSIF d=2 THEN b ELSE c ENDIF)

 check1: CONJECTURE  f2 = (f1 WITH [ 2 := b])
 % Subtype TCC generated ... expected type  DOMAIN
 % unfinished
 check1_TCC1: OBLIGATION 2 <= n;
```

How to fix?

TCC with Function Conjecture


```
function1: THEORY BEGIN
  n: posnat
  DOMAIN : TYPE = d : posnat | d <= n
  d: VAR DOMAIN
  RANGE: TYPE = a,b,c
  f1(d):RANGE =
   (IF d = 1 then a ELSE c ENDIF)
  f2(d):RANGE =
   (IF d = 1 then a ELSIF d=2 THEN b ELSE c ENDIF)

 check1: CONJECTURE  f2 = (f1 WITH [ 2 := b])
 % Subtype TCC generated ... expected type  DOMAIN
 % unfinished
 check1_TCC1: OBLIGATION 2 <= n;
```

How to fix?

TCC with Function Conjecture

How to fix?

```
check1: CONJECTURE  f2 = (f1 WITH [ 2 := b])
% Subtype TCC generated ... expected type  DOMAIN
% unfinished
 check1_TCC1: OBLIGATION 2 <= n;
```

Fixes

- Add an assumption:
  `CONJECTURE n >= 2 => f2 = (f1 WITH [ 2 := b])`
- Or add an Axiom (not generally recommended)

TCC with Function Conjecture

How to fix?

```
check1: CONJECTURE  f2 = (f1 WITH [ 2 := b])
% Subtype TCC generated ... expected type  DOMAIN
% unfinished
 check1_TCC1: OBLIGATION 2 <= n;
```

Fixes

- Add an assumption:
  `CONJECTURE n >= 2 => f2 = (f1 WITH [ 2 := b])`
- Or add an Axiom (not generally recommended)

PVS Proof Steps with Override

```
check1 :
   |-------
{1}   f2 = (f1 WITH [(2) := b])
Rule? (apply-extensionality)
this yields  2 subgoals: check1.1 :
   |-------
{1}   f2(x!1) = (f1 WITH [(2) := b])(x!1)
[2]   f2 = (f1 WITH [(2) := b])
Rule? (delete 2)
   |-------
[1]   f2(x!1) = (f1 WITH [(2) := b])(x!1)

Rule? (lift-if)
this simplifies to:  check1.1 :
   |-------
1   IF x!1 = 2 THEN f2(x!1) = b ELSE f2(x!1) = f1(x
```

PVS Proof Steps with Override


```
check1 :
  |-------
{1}   f2 = (f1 WITH [(2) := b])
Rule? (apply-extensionality)
this yields  2 subgoals: check1.1 :
  |-------
{1}   f2(x!1) = (f1 WITH [(2) := b])(x!1)
[2]   f2 = (f1 WITH [(2) := b])
Rule? (delete 2)
  |-------
[1]   f2(x!1) = (f1 WITH [(2) := b])(x!1)

Rule? (lift-if)
this simplifies to:  check1.1 :
  |-------
1   IF x!1 = 2 THEN f2(x!1) = b ELSE f2(x!1) = f1(x
```

PVS Proof Steps with Override

```
check1 :
   |-------
{1}   f2 = (f1 WITH [(2) := b])
Rule? (apply-extensionality)
this yields  2 subgoals: check1.1 :
   |-------
{1}   f2(x!1) = (f1 WITH [(2) := b])(x!1)
[2]   f2 = (f1 WITH [(2) := b])
Rule? (delete 2)
   |-------
[1]   f2(x!1) = (f1 WITH [(2) := b])(x!1)

Rule? (lift-if)
this simplifies to:  check1.1 :
   |-------
1   IF x!1 = 2 THEN f2(x!1) = b ELSE f2(x!1) = f1(x
```

PVS Proof Steps with Override


```
check1 :
   |-------
{1}   f2 = (f1 WITH [(2) := b])
Rule? (apply-extensionality)
this yields  2 subgoals: check1.1 :
   |-------
{1}   f2(x!1) = (f1 WITH [(2) := b])(x!1)
[2]   f2 = (f1 WITH [(2) := b])
Rule? (delete 2)
   |-------
[1]   f2(x!1) = (f1 WITH [(2) := b])(x!1)

Rule? (lift-if)
this simplifies to:  check1.1 :
   |-------
1   IF x!1 = 2 THEN f2(x!1) = b ELSE f2(x!1) = f1(x
```

PVS Proof Steps with Override


```
check1 :
   |-------
{1}   f2 = (f1 WITH [(2) := b])
Rule? (apply-extensionality)
this yields  2 subgoals: check1.1 :
   |-------
{1}   f2(x!1) = (f1 WITH [(2) := b])(x!1)
[2]   f2 = (f1 WITH [(2) := b])
Rule? (delete 2)
   |-------
[1]   f2(x!1) = (f1 WITH [(2) := b])(x!1)

Rule? (lift-if)
this simplifies to:  check1.1 :
   |-------
1   IF x!1 = 2 THEN f2(x!1) = b ELSE f2(x!1) = f1(x
```

PVS Proof Steps with Override

```
|-------
1    IF x!1 = 2 THEN f2(x!1) = b
                    ELSE f2(x!1) = f1(x!1) ENDIF
Rule? (then (split) (flatten))
this yields  2 subgoals: check1.1.1 :
-1  x!1 = 2
  |-------
1   f2(x!1) = b
Rule? (replace -1 1)
this simplifies to: check1.1.1 :
[-1]  x!1 = 2
  |-------
1   f2(2) = b
Rule? (expand "f2")
this simplifies to:  check1.1.1 :
[-1]  x!1 = 2
  |-------
1   TRUE which is trivially true ... etc.
```

PVS Proof Steps with Override


```
|-------
1    IF x!1 = 2 THEN f2(x!1) = b
                   ELSE f2(x!1) = f1(x!1) ENDIF
Rule? (then (split) (flatten))
this yields  2 subgoals: check1.1.1 :
-1   x!1 = 2
   |-------
1    f2(x!1) = b
Rule? (replace -1 1)
this simplifies to: check1.1.1 :
[-1]  x!1 = 2
   |-------
1    f2(2) = b
Rule? (expand "f2")
this simplifies to:  check1.1.1 :
[-1]  x!1 = 2
   |-------
1    TRUE which is trivially true ... etc.
```

PVS Proof Steps with Override


```
 |-------
1    IF x!1 = 2 THEN f2(x!1) = b
                    ELSE f2(x!1) = f1(x!1) ENDIF
Rule? (then (split) (flatten))
this yields  2 subgoals: check1.1.1 :
-1   x!1 = 2
   |-------
1    f2(x!1) = b
Rule? (replace -1 1)
this simplifies to: check1.1.1 :
[-1]  x!1 = 2
   |-------
1    f2(2) = b
Rule? (expand "f2")
this simplifies to:  check1.1.1 :
[-1]  x!1 = 2
   |-------
1    TRUE which is trivially true ... etc.
```

PVS Proof Steps with Override


```
 |-------
1    IF x!1 = 2 THEN f2(x!1) = b
                        ELSE f2(x!1) = f1(x!1) ENDIF
Rule? (then (split) (flatten))
this yields  2 subgoals: check1.1.1 :
-1   x!1 = 2
   |-------
1    f2(x!1) = b
Rule? (replace -1 1)
this simplifies to: check1.1.1 :
[-1]   x!1 = 2
   |-------
1    f2(2) = b
Rule? (expand "f2")
this simplifies to:  check1.1.1 :
[-1]   x!1 = 2
   |-------
1    TRUE which is trivially true ... etc.
```

PVS Proof Steps with Override


```
 |-------
1    IF x!1 = 2 THEN f2(x!1) = b
                   ELSE f2(x!1) = f1(x!1) ENDIF
Rule? (then (split) (flatten))
this yields  2 subgoals: check1.1.1 :
-1   x!1 = 2
  |-------
1    f2(x!1) = b
Rule? (replace -1 1)
this simplifies to: check1.1.1 :
[-1]   x!1 = 2
  |-------
1    f2(2) = b
Rule? (expand "f2")
this simplifies to:  check1.1.1 :
[-1]   x!1 = 2
  |-------
1    TRUE which is trivially true ... etc.
```

PVS Proof Steps with Override


```
 |-------
1   IF x!1 = 2 THEN f2(x!1) = b
                      ELSE f2(x!1) = f1(x!1) ENDIF
Rule? (then (split) (flatten))
this yields  2 subgoals: check1.1.1 :
-1   x!1 = 2
   |-------
1   f2(x!1) = b
Rule? (replace -1 1)
this simplifies to: check1.1.1 :
[-1]  x!1 = 2
   |-------
1   f2(2) = b
Rule? (expand "f2")
this simplifies to:  check1.1.1 :
[-1]  x!1 = 2
   |-------
1   TRUE which is trivially true ... etc.
```

Multi-step Override

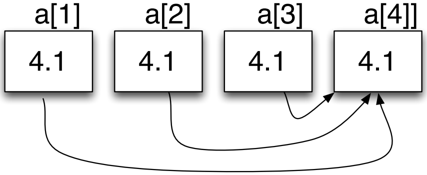

```
f1(d):RANGE =
 (IF d = 1 then a ELSE c ENDIF)

f2(d):RANGE =
 (IF d = 1 then a ELSIF d=2 THEN b ELSE c ENDIF)


check3: CONJECTURE
  f2 = (f1 WITH [ 2 := a, 2 := b])

% Shows that overriding applies
% in the order in which they appear
% i.e. (f1 WITH [ 2 := a, 2 := b])
% =
%     ((f1 WITH [ 2 := a]) WITH [2:=b])
%  =  f1 WITH [2 := b]
```

Multi-step Override


```
f1(d):RANGE =
 (IF d = 1 then a ELSE c ENDIF)

f2(d):RANGE =
 (IF d = 1 then a ELSIF d=2 THEN b ELSE c ENDIF)


check3: CONJECTURE
  f2 = (f1 WITH [ 2 := a, 2 := b])

% Shows that overriding applies
% in the order in which they appear
% i.e. (f1 WITH [ 2 := a, 2 := b])
% =
%    ((f1 WITH [ 2 := a]) WITH [2:=b])
%  =  f1 WITH [2 := b]
```

The problem of Aliasing

a: ARRAY[REAL_REF]



a[4] := 7.3

Mathematical Specification

```
a: [1..4 -> real]  % 1..4 is not PVS notation
a WITH [4 := 7.3]
```

Array a changes only at index 4!

Aliasing and Null Reference

> *"I call it my billion-dollar mistake. It was the invention of the null reference in 1965. At that time, I was designing the first comprehensive type system for references in an object oriented language (ALGOL W). My goal was to ensure that all use of references should be absolutely safe, with checking performed automatically by the compiler. But I couldn't resist the temptation to put in a null reference, simply because it was so easy to implement. This has led to innumerable errors, vulnerabilities, and system crashes, which have probably caused a billion dollars of pain and damage in the last forty years."*

(Turing award winner C. A. R. Hoare in 2009)

Aliasing and Null Reference

> *Ten years ago, researchers into formal methods (and I was the most mistaken among them) predicted that the programming world would embrace with gratitude every assistance promised by formalisation to solve the problems of reliability that arise when programs get large and more safety-critical. Programs have now got very large and very critical well beyond the scale which can be comfortably tackled by formal methods. There have been many problems and failures, but these have nearly always been attributable to inadequate analysis of requirements or inadequate management control. It has turned out that the world just does not suffer significantly from the kind of problem that our research was originally intended to solve.*

(Turing award winner C. A. R. Hoare in 1996)

The problem of Aliasing

What's the hardest bit about writing an app?

Survey period: **28 Jul 2014** to **4 Aug 2014**

We'll assume writing the actual code is the easy bit...

Option	Votes	%
Interpreting specs (or lack thereof)	792	48.41
Getting the architecture right (without redoing it 3 times)	553	33.80
Dealing with the compiler / framework / libraries / OS etc	225	13.75
Ensuring it works on different hardware / browsers / systems	488	29.83
Ensuring it's fast / small / resource friendly enough	242	14.79
Getting the User Experience and UI / graphics spot on	543	33.19
Testing sufficiently	504	30.81
Dealing with the client	549	33.56
Other	80	4.89
Responses	**1636**	

Respondents were allowed to choose more than one answer; totals may not add up to 100%

Phone Specification and Validation

Specification

- A phone book shall store the phone numbers of a city
- It shall be possible to retrieve a phone number given a name
- It shall be possible to add and delete entries from a phone book

Validating the Specification

- If I add a name *nm* with phone number *pn* to a phone book and look up the name *nm*, I should get back the phone number *pn*
- The result of adding a name and then deleting it is the same as just deleting it

Phone Specification and Validation

Specification

- A phone book shall store the phone numbers of a city
- It shall be possible to retrieve a phone number given a name
- It shall be possible to add and delete entries from a phone book

Validating the Specification

- If I add a name *nm* with phone number *pn* to a phone book and look up the name *nm*, I should get back the phone number *pn*
- The result of adding a name and then deleting it is the same as just deleting it