Phone Book Example

John Rushby

Computer Science Laboratory
SRI International
Menlo Park CA USA

Phone Book Example

Requirements for an electronic phone book

- 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 the phone book

Formal Requirements Specification

How do we represent the phone book mathematically?

1. A set of (*name*, *number*) pairs.
   Adding and deleting entries via set addition and deletion

2. A total function (i.e., array) whose domain is the space of possible names and whose range is the space of all phone numbers.
   Adding and deleting entries via modification of function values

3. A partial function whose domain is just the names currently in phone book and whose range is the space of all phone numbers.
   Adding and deleting entries via modification of the function domain and values

Let's start with approach 2

Specifying the Book

- In traditional mathematical notation, we would write:

$$\begin{array}{rcl} \text{Let } N & : & \text{type (of names)} \\ P & : & \text{type (of phone numbers)} \\ \textit{book} & : & \text{type (of functions) } [N \rightarrow P] \end{array}$$

- How do we indicate that we do not have a phone number for all possible names, only for names of real people?

  Decide to use a special number, that could never really occur in real life, e.g. 000-0000; don't have to specify the value of this number we can just give it a name (e.g., n_0)

- Now can define an empty phone book:

$$\begin{array}{l} \textit{emptybook} : [N \rightarrow P] \\ n_0 : P \\ nm : \text{var } N \\ \textbf{axiom} : \forall\ nm : \textit{emptybook}(nm) = n_0 \end{array}$$

Accessing an Entry

N : type (of names)

P : type (of phone numbers)

B : type (of functions) $[N \rightarrow P]$

FindPhone : $[B \times N \rightarrow P]$

nm : var N

bk : var B

axiom : *FindPhone*$(bk, nm) = bk(nm)$

Note that *FindPhone* is a higher-order function since its first argument is a function

Specifying Adding/Deleting an Entry

N : type (of names)

P : type (of phone numbers)

B : type (of functions) $[N \to P]$

$n_0 : N$

nm, x : var N

pn : var P

bk : var B

AddPhone : $[B \times N \times P \to B]$

axiom : $\mathit{AddPhone}(bk, nm, pn)(x) = \begin{cases} bk(x) & \text{if } x \neq nm \\ pn & \text{if } x = nm \end{cases}$

DelPhone : $[B \times N \to B]$

axiom : $\mathit{DelPhone}(bk, nm)(x) = \begin{cases} bk(x) & \text{if } x \neq nm \\ n_0 & \text{if } x = nm \end{cases}$

PVS Notation

```
phone_1: THEORY
BEGIN

  N: TYPE                 % names
  P: TYPE                 % phone numbers
  B: TYPE = [N -> P]      % phone books

  n0: P
  emptybook: B
  emptyax: AXIOM   FORALL (nm: N): emptybook(nm) = n0

  FindPhone: [B, N -> P]
  Findax: AXIOM   FORALL (bk: B), (nm: N):  FindPhone(bk, nm) = bk(nm)

  nm: VAR N
  pn: VAR P
  bk: VAR B

  AddPhone: [B, N, P -> B]
  Addax: AXIOM   AddPhone(bk, nm, pn) = bk WITH [(nm) := pn]

  DelPhone: [B, N -> B]
  Delax: AXIOM   DelPhone(bk, nm) = bk WITH [(nm) := n0]

END phone_1
```

Challenging the Requirement Specification

- If you add a name and number and then look it up, do you get the right answer?

$$\textbf{lemma} : \mathit{FindPhone}(\mathit{AddPhone}(bk, nm, pn), nm) = pn$$

- If you add an entry and then delete it, is the phone book unchanged?

$$\textbf{lemma} : \mathit{DelPhone}(\mathit{AddPhone}(bk, nm, pn), nm) = bk$$

  - Not true unless $bk(name) = n_0$ beforehand
  - Is this what was intended?
  - Should we modify the specification of *AddPhone*?
  - Do we need a function *ChangePhone*?
  - Should we allow multiple numbers per name?

An Aside on Axioms

- Suppose we want to separate the functions of *adding* and *changing* a number
- To define these, useful to have a predicate

$$\mathit{Known?} : [B \times N \to \mathit{bool}]$$

$$\textbf{axiom} : \mathit{Known?}(bk, nm) \text{ iff } bk(nm) \neq n_0$$

- Suppose we also had axiom

$$\textbf{axiom} : \mathit{Known?}(\mathit{AddPhone}(bk, nm, pn), nm)$$

- We get an inconsistency—can prove *anything*
- Use axioms only where necessary; best to use *definitional* forms of specification (guaranteed not to introduce inconsistencies)
- PVS may generate proof obligations (TCCs) to ensure this guarantee

Some Deficiencies of First Specification

1. Our specification does not rule out the possibility of someone having a "n_0" phone number

2. We have not allowed multiple phone numbers per name

3. Our specification does not say anything about whether or not we should warn the user if *AddPhone* results in the same number being assigned to two people

How do we remedy these deficiencies?

Deficiency 1

Our specification does not rule out the possibility of someone having a "n_0" phone number

There are several ways to overcome this problem

- Use a "disjoint union" for the range type of the phone book, so that n_0 is not an ordinary number
- Use a "predicate subtype" to identify the phone numbers different to n_0 and allow only the subtype in *AddPhone*
- Use one of the other representations for the phone book (e.g., partial functions—requires a different specification language)
- Reconsider requirements

Predicate Subtypes

- Can define the type GP of *Good Phone Numbers*:

$$GP : \mathsf{type} = \{pn : P | pn \neq n_0\}$$

- Then define *AddPhone definitionally* as:

gp : var GP

$\mathit{AddPhone}(bk, nm, gp) : B =$

if $\mathit{Known?}(bk, nm)$ **then** bk **else** bk **with** $[(nm){:=}gp]$ **endif**

- Notice the flawed axiom we had before is no longer admissible

$$\textbf{axiom} : \mathit{Known?}(\mathit{AddPhone}(bk, nm, pn), nm)$$

(PVS generates the impossible proof obligation $\forall pn : pn \neq n_0$)

- But the following is a provably true

$$\mathit{Known?}(\mathit{AddPhone}(bk, nm, gp), nm)$$

Deficiency 2

- We have not allowed multiple phone numbers per name
- The original requirements did not specify whether this is needed or not
- Suppose, after conferring with the customer, we decide to allow multiple numbers
- Change the range type of the phone book to a set of numbers
- This solves Deficiency 1 as well
  (empty set of numbers indicates name not in the book)

New Specification (sets)

N : type (of names)

P : type (of phone numbers)

B : type (of functions) $[N \rightarrow \text{setof}[P]]$

nm, x : var N

$\mathit{emptybook}(nm) : \text{setof}[P] = \phi_P$

pn : var P

bk : var B

$\mathit{FindPhone}(bk, nm) : \text{setof}[P] = bk(nm)$

$\mathit{AddPhone}(bk, nm, pn) : B = bk$ **with** $[(nm) := bk(nm) \cup \{pn\}]$

$\mathit{DelPhone}(bk, nm) : B = bk$ **with** $[(nm) := \phi_P]$

Some Observations

- Our specification is abstract; the functions are defined over uninterpreted domains.

- The axioms and definitions used here are constructive—we could execute them
(could also use pseudocode for these kinds of specifications, but would lack an assertion language for challenges, and the deductive apparatus to formally check their proofs)

- Other specifications and representations may involve nonconstructive axioms
e.g., set of pairs: $\mathit{FindPhone}(bk, nm) = \{pn | (nm, pn) \in bk\}$

- And more sophisticated (not directly implementable) types

More Observations

- As requirements are formalized, many things that are usually left out of English specifications are encountered and explicitly documented

- The formal process exposes ambiguities and deficiencies in the requirements—must chose between

$$book : [N \rightarrow P]$$

$$book : [N \rightarrow setof[P]]$$

- Challenges and scrutiny reveal deficiencies in the formal specification

- The process of formalizing the requirements can reveal problems and deficiencies and lead to a better English requirements document as well

Deficiency 3

- Suppose we wish to avoid ever assigning the same number to two people
- Could "program" this into the specification of each function that changes the phone book
- But really want to establish the property as an *invariant* of the specification
- Could systematically generate the proof obligations to ensure this is so, but the activity would be error-prone
- Could build a tool to do it, but that would be special-purpose
- Solution: do it with predicate subtypes

PVS Notation: subtype invariant

```
phone_4 : THEORY
  BEGIN

  N: TYPE                                  % names
  P: TYPE                                  % phone numbers
  B: TYPE = [N -> setof[P]]                % phone books
  VB: TYPE = {b:B | (FORALL (x,y:N): x /= y => disjoint?(b(x), b(y)))}

  nm, x: VAR N
  pn: VAR P
  bk: VAR VB

  FindPhone(bk,nm): setof[P] = bk(nm)

  UnusedPhoneNum(bk,pn): bool =
     (FORALL nm: NOT member(pn,FindPhone(bk,nm)))

  AddPhone(bk,nm,pn): VB =
    IF UnusedPhoneNum(bk,pn) THEN bk WITH [(nm) := add(pn, bk(nm))]
      ELSE bk
    ENDIF
```

PVS Notation: Proof Obligation

```
AddPhone_TCC1: OBLIGATION
      (FORALL (bk: VB, nm: N, pn: P):
         UnusedPhoneNum(bk, pn) IMPLIES
           (FORALL (x, y: N):
              x /= y =>
                disjoint?[P](bk WITH [(nm) := add[P](pn, bk(nm))](x),
                             bk WITH [(nm) := add[P](pn, bk(nm))](y))));
```

Yet More Observations

- There are many different ways to write formal specifications

- Some ways of writing them bias the feasible implementation more than others

- One goal is to minimize this bias, and yet be complete
  - Abstract specifications are more likely to highlight substance than those cluttered with implementation concerns
  - *But requires real judgment and experience to pick right level*
  - Constructive specifications may be executable as prototypes—useful in some domains, distraction in others

- Mechanized support allows powerful checks on consistency, and active validation through "challenges" to the specification