Applied Logic for Hardware and Software

Software Properties as Axioms and Theorems

proofs by induction

… or …

How to Write Reliable Software

and know it's reliable

Example: Multiplexor Function

- Problem: Multiplex two sequences to form a new sequence
  - mux [1, 2, 3, 4, 5] [6, 7, 8, 9, 10] = [1, 6, 2, 7, 3, 8, 4, 9, 5, 10]
  - mux [1, 2, 3, 4, 5] [6, 7, 8] = [1, 6, 2, 7, 3, 8, 4, 5]
  - mux [1, 2, 3, 4] [6, 7, 8, 9, 10] = [1, 6, 2, 7, 3, 8, 4, 9, 10]
- Notation
  - $[x_1, x_2, x_3]$ ←denotes a sequence with three elements
  - xs ++ ys ←denotes concatenation of the sequences xs and ys
    - ✓ $[x_1, x_2, x_3]$ ++ $[y_1, y_2, y_3, y_4]$ = $[x_1, x_2, x_3, y_1, y_2, y_3, y_4]$
- Informal specification of mux
  - mux $[x_1, x_2, \ldots x_n]$ $[y_1, y_2, \ldots y_n]$ = $[x_1, y_1, x_2, y_2, \ldots x_n, y_n]$
  - Note: append extra elements in either sequence to the end

Multiplexor Function —Testing

- Problem: multiplex two sequences to form a new sequence
  - mux [1, 2, 3, 4, 5] [6, 7, 8, 9, 10] = [1, 6, 2, 7, 3, 8, 4, 9, 5, 10]
  - mux [1, 2, 3, 4, 5] [6, 7, 8] = [1, 6, 2, 7, 3, 8, 4, 5]
  - mux [1, 2, 3, 4] [6, 7, 8, 9, 10] = [1, 6, 2, 7, 3, 8, 4, 9, 10]
- Notation
  - $[x_1, x_2, x_3]$ ←denotes a sequence with three elements
  - xs ++ ys ←denotes concatenation of the sequences xs and ys
    - ✓ $[x_1, x_2, x_3]$ ++ $[y_1, y_2, y_3, y_4]$ = $[x_1, x_2, x_3, y_1, y_2, y_3, y_4]$
- Informal specification of mux
  - mux $[x_1, x_2, \ldots x_n]$ $[y_1, y_2, \ldots y_n]$ = $[x_1, y_1, x_2, y_2, \ldots x_n, y_n]$
  - Note: append extra elements in either sequence to the end
- Suppose we've defined mux in a program and want to test it
  - Example test: mux [1, 2, 3] [4, 5, 6, 7] == [1, 4, 2, 5, 3, 6, 7]
- That's an okay test, but hard to generalize for automated testing
  - The following tests are wrong with lists of differing lengths
    - ✓ muxTest1 xs ys = $((\text{mux xs ys})_{2n+1} == xs_n$ if $n \leq$ length xs)
    - ✓ muxTest2 xs ys = $((\text{mux xs ys})_{2n} == ys_n$ if $n \leq$ length ys)

Multiplexor — More General Test

- Informal specification of mux
  - mux $[x_1, x_2, \ldots x_n]$ $[y_1, y_2, \ldots y_n] = [x_1, y_1, x_2, y_2, \ldots x_n, y_n]$
  - Note: append extra elements in either sequence to the end
- Suppose we've defined mux in a program and want to test it
  - Example test: mux [1, 2, 3] [4, 5, 6, 7] == [1, 4, 2, 5, 3, 6, 7]
- That's an okay test, but hard to generalize for automated testing
  - The following tests are wrong with lists of differing lengths
    - ✓ muxTest xs ys = $((\text{mux xs ys})_{2n+1} == xs_n$ if $n \leq$ length xs)
    - ✓ muxTest xs ys = $((\text{mux xs ys})_{2n} == ys_n$ if $n \leq$ length ys)
- Let's think about a test that is easier to generalize
  - mux $[x_1, x_2, \ldots x_{n+1}]$ $[y_1, y_2, \ldots y_{m+1}]$
    $= [x_1, y_1]$ ++ (mux $[x_2, \ldots x_{n+1}]$ $[y_2, \ldots y_{m+1}]$)

Multiplexor — Massive Testing

- Informal specification of mux
  - mux $[x_1, x_2, \ldots x_n]$ $[y_1, y_2, \ldots y_n] = [x_1, y_1, x_2, y_2, \ldots x_n, y_n]$
  - Note: append extra elements in either sequence to the end
- Suppose we've defined mux in a program and want to test it
  - Example test: mux [1, 2, 3] [4, 5, 6, 7] == [1, 4, 2, 5, 3, 6, 7]
- That's an okay test, but hard to generalize for automated testing
  - The following tests are wrong with lists of differing lengths
    - ✓ muxTest xs ys = $((\text{mux xs ys})_{2n+1} == xs_n$ if $n \leq$ length xs)
    - ✓ muxTest xs ys = $((\text{mux xs ys})_{2n} == ys_n$ if $n \leq$ length ys)
- Let's think about a test that is easier to generalize
  - mux $[x_1, x_2, \ldots x_{n+1}]$ $[y_1, y_2, \ldots y_{m+1}]$
    $= [x_1, y_1]$ ++ (mux $[x_2, \ldots x_{n+1}]$ $[y_2, \ldots y_{m+1}]$)
- We can correctly generalize that test as follows
  - muxTest (x : xs) (y : ys) =
    ((mux (x : xs) (y : ys)) == ([x, y] ++ (mux xs ys)))
- Notation
  - x : xs ←shorthand for [x] ++ xs
- We can use muxTest to run thousands of tests, automatically

Multiplexor – Defining Properties

- Let's think about a test that is easier to generalize
  - mux $[x_1, x_2, \ldots x_{n+1}]$ $[y_1, y_2, \ldots y_{m+1}]$
    $= [x_1, y_1]$ ++ (mux $[x_2, \ldots x_{n+1}]$ $[y_2, \ldots y_{m+1}]$)
- We can correctly generalize that test as follows
  - muxTest (x : xs) (y : ys) = ((mux (x : xs) (y : ys))==([x, y] ++ (mux xs ys)))
- Notation
  - x : xs ←shorthand for [x] ++ xs
- We can use muxTest to run thousands of tests, automatically
- How about some corner cases?
  - mux xs [ ] = xs
  - mux [ ] (y : ys) = (y : ys)

Multiplexor – Defining Properties

- Let's think about a test that is easier to generalize
  - mux $[x_1, x_2, \ldots x_{n+1}]$ $[y_1, y_2, \ldots y_{m+1}]$
    $= [x_1, y_1]$ ++ (mux $[x_2, \ldots x_{n+1}]$ $[y_2, \ldots y_{m+1}]$)
- We can correctly generalize that test as follows
  - muxTest (x : xs) (y : ys) = ((mux (x : xs) (y : ys))==([x, y] ++ (mux xs ys)))
- Notation
  - x : xs ←shorthand for [x] ++ xs
- We can use muxTest to run thousands of tests, automatically
- How about some corner cases?
  - mux xs [ ] = xs
  - mux [ ] (y : ys) = (y : ys)
- Now we have three simple properties that mux must satisfy
  - mux (x : xs) (y : ys) = [x, y] ++ (mux xs ys)
  - mux xs [ ] = [ ]
  - mux [ ] (y : ys) = (y : ys)

Multiplexor – Inductive Definition

- Let's think about a test that is easier to generalize
  - mux $[x_1, x_2, \ldots x_{n+1}]$ $[y_1, y_2, \ldots y_{m+1}]$
    $= [x_1, y_1]$ ++ (mux $[x_2, \ldots x_{n+1}]$ $[y_2, \ldots y_{m+1}]$)
- We can correctly generalize that test as follows
  - muxTest (x : xs) (y : ys) = ((mux (x : xs) (y : ys))==([x, y] ++ (mux xs ys)))
- Notation
  - x : xs ←shorthand for [x] ++ xs
- We can use muxTest to run thousands of tests, automatically
- How about some corner cases?
  - mux xs [ ] = xs
  - mux [ ] (y : ys) = (y : ys)
- Now we have three simple properties that mux must satisfy
  - mux (x : xs) (y : ys) = [x, y] ++ (mux xs ys)
  - mux xs [ ] = [ ]
  - mux [ ] (y : ys) = (y : ys)
- Surprise ! All properties of mux derive from these
  - That is, these properties form an inductive definition of mux

How to Define Functions

- Now we have three simple properties that mux must satisfy
  - mux (x : xs) (y : ys) = [x, y] ++ (mux xs ys)
  - mux xs [ ] = xs
  - mux [ ] (y : ys) = (y : ys)
- Surprise ! All properties of mux derive from these
  - In other words, these simple properties define mux
- How to write inductive definitions
  1. Write simple equations that the desired function must satisfy
  2. Make sure there is an equation for each expected form of input
  3. Make sure the input for each inductive reference is closer to a non-inductive case than the input form covered by the equation
- Writing reliable programs is as simple as that!

Axiomatic vs Derived Properties

- Now we have three simple properties that mux must satisfy
  - mux (x : xs) (y : ys) = [x, y] ++ (mux xs ys) {Axiom M_1}
  - mux xs [ ] = xs {Axiom $M_{0.2}$}
  - mux [ ] (y : ys) = (y : ys) {Axiom $M_{0.1}$}

  axioms

  Surprise ! All properties of mux derive from these
  - In other words, these simple properties define mux
- How to write inductive definitions
  1. Write simple equations that the desired function must satisfy
  2. Make sure there is an equation for each expected form of input
  3. Make sure the input for each inductive reference is closer to a non-inductive case than the input form covered by the equation
- Writing reliable programs is as simple as that!
  - Even better, you can derive additional properties by induction
  - Such as, for example, these properties

    $\{P_1\}$ if (elem x xs) then (elem x (mux xs ys))

    $\{P_2\}$ if (elem y ys) then (elem y (mux xs ys))

    $\{P_3\}$ if (elem z (mux xs ys)) then ((elem z xs) || (elem z ys))
- Notation
  - (elem x xs) is True iff x is an element of the sequence xs

Derived Properties are Theorems

- Now we have three simple properties that mux must satisfy
  - mux (x : xs) (y : ys) = [x, y] ++ (mux xs ys) {Axiom M_1}
  - mux xs [ ] = xs {Axiom $M_{0.2}$}
  - mux [ ] (y : ys) = (y : ys) {Axiom $M_{0.1}$}

axioms

- Surprise ! All properties of mux derive from these
  - In other words, these two simple properties define mux
- How to write inductive definitions
  1. Write simple equations that the desired function must satisfy
  2. Make sure there is an equation for each expected form of input
  3. Make sure the input for each inductive reference is closer to a non-inductive case than the input form covered by the equation
- Writing reliable programs is as simple as that!
  - Even better, you can derive additional properties by induction
  - Such as, for example, these properties

    {P_1} if (elem x xs) then (elem x (mux xs ys))

    {P_2} if (elem y ys) then (elem y (mux xs ys))

    {P_3} if (elem z (mux xs ys)) then ((elem z xs) || (elem z ys))

theorems

- Notation
  - (elem x xs) is True iff x is an element of the sequence xs

Property Derived by Induction

- Now we have three simple properties that mux must satisfy
  - mux (x : xs) (y : ys) = [x, y] ++ (mux xs ys) {Axiom M_1}
  - mux xs [ ] = xs {Axiom $M_{0.2}$}
  - mux [ ] (y : ys) = (y : ys) {Axiom $M_{0.1}$}
  - elem x [ ] = False {Axiom E_o}
  - elem y (x : xs) = (y == x) || (elem y xs) {Axiom E_1}
- Theorem P_1: if (elem x xs) then (elem x (mux xs ys))
  that is, (elem x xs) $\rightarrow$ (elem x (mux xs ys)) {as Boolean formula}
  - Base case

  (elem x [ ]) $\rightarrow$ (elem x (mux [ ] ys))
  = False $\rightarrow$ (elem x (mux [ ] ys)) {Axiom E_o}
  = True {Boolean algebra}

Property Derived by Induction

- Now we have three simple properties that mux must satisfy
  - mux (x : xs) (y : ys) = [x, y] ++ (mux xs ys) {Axiom M_1}
  - mux xs [ ] = xs {Axiom $M_{0.2}$}
  - mux [ ] (y : ys) = (y : ys) {Axiom $M_{0.1}$}
  - elem x [ ] = False {Axiom E_o}
  - elem y (x : xs) = (y == x) || (elem y xs) {Axiom E_1}
- Theorem P_1: if (elem x xs) then (elem x (mux xs ys))
  that is, (elem x xs) $\rightarrow$ (elem x (mux xs ys)) {as Boolean formula}

- Inductive case when ys $\neq$ [ ] Note: proof is simpler when ys = [ ]

(elem x (x_1: xs)) $\rightarrow$ (elem x (mux (x_1: xs) (z: zs)) {rewrite non-empty ys as (z: zs)}

= ((x==x_1) || (elem x xs)) $\rightarrow$ (elem x ([x_1, z] ++ (mux xs zs))) {Axioms E_1 and M_1}

= ((x==x_1) || (elem x xs)) $\rightarrow$ (elem x (x_1: z: (mux xs zs))) {shorthand for ++}

= ((x==x_1) || (elem x xs)) $\rightarrow$ ((x==x_1) || (x==z) || (elem x (mux xs zs))) {Ax E_1}

= ((x==x_1) $\rightarrow$ ((x==x_1) || (x==z) || (elem x (mux xs zs)))) && {Boolean algebra}
((elem x xs) $\rightarrow$ ((x==x_1) || (x==z) || (elem x (mux xs zs))))

= True && {more Boolean algebra}
((elem x xs) $\rightarrow$ ((x==x_1) || (x==z) || (elem x (mux xs zs))))

= True {induction hypothesis, along with some Boolean algebra}

Using Logic to Ensure Reliable Software

- Pencil and paper proofs are just for practice
  - Enables students to learn methods of reasoning
  - Logic course at Oklahoma Univ includes hundreds of examples
    - ✓ Course/results: http://www.cs.ou.edu/~beseme/sfwIsDMpaper.pdf
    - ✓ Textbook: Discrete Math Using a Computer (Springer 2006)
  - Proofs follow the rigorous style of mux example presented here
- Real-life software engineering requires mechanized logic
  - Getting all the details right in proofs isn't humanly possible
  - Proofs with incorrect details are not proofs
  - Non-proofs don't guarantee software properties
  - Therefore, paper-and-pencil proofs provide no guarantees
- Mechanized logic verifies all the details, down to bit level
  - So, it's feasible to know software properties for certain
  - Also, mechanized logic can partially automate proofs
  - This makes mechanized logic useful in the practice of SE
- Practical mechanized logic for industry: ACL2
  - SE course at Oklahoma Univ uses ACL2 with Proof Pad
    - ✓ http://www.cs.ou.edu/~rlpage/SEcollab/EngrSwJFP.pdf

The End