Announcements

- Presentation guidelines and papers up on Schedule page
  - 1. Select available paper/slot from list (first-come-first-serve)
  - 2. If available, I’ll assign and update, otherwise goto 1
- Presentations are in teams of two

Quiz 6

Outline

- Axiomatic semantics
- IMP
- Semantics
- Verification condition generation
  - Essentially, what we called “backwards reasoning” in Principles of Software
- SMT-LIB
- Catch-up: Monads

The IMP Language

- Expressions
  - $e := n \mid x \mid e_1 + e_2 \mid e_1 = e_2$
- Commands (i.e., statements, change state):
  - $c ::= x := e \mid c_1 ; c_2 \mid \text{if } (e) \text{ then } c_1 \text{ else } c_2 \mid \text{while } (e) \text{ do } c \mid \text{skip}$
  - A big-step operational semantics

Big-step Operational Semantics

- Operational semantics of expressions:
  - $[[n]](\sigma) = n$ // constant $n$ evaluates to $n$
  - $[[x]](\sigma) = \sigma(x)$ // variable $x$ evaluates to the value $n$ that $x$ maps to in $\sigma$
- Assignment: $\eta : x := E; \eta : \ldots$
  - $\eta ; \sigma \rightarrow (\eta, \sigma[x \leftarrow [E]([\sigma]))$
- Assignment: $\eta : x := E_1 \text{ Op } E_2 ; \eta : \ldots$
  - $\eta ; \sigma \rightarrow (\eta, \sigma[x \leftarrow [E_1][\sigma]) \text{ Op } [E_2]([\sigma]))$

Aside: Small-step Operational Semantics

- Assignment: $\eta : x := E; \eta : \ldots$
  - $\eta ; \sigma \rightarrow (\eta, \sigma[x \leftarrow [E]([\sigma]))$
- Assignment: $\eta : x := E_1 \text{ Op } E_2 ; \eta : \ldots$
  - $\eta ; \sigma \rightarrow (\eta, \sigma[x \leftarrow [E_1][\sigma]) \text{ Op } [E_2]([\sigma]))$
Aside: Small-step Operational Semantics

- Loop: \( q : \text{while} (b) \{ q : \ldots \} q : \ldots \)
  - if \([b](\sigma) = \text{True}\) then \((q, \sigma) \rightarrow (q', \sigma)\)
  - if \([b](\sigma) = \text{False}\) then \((q, \sigma) \rightarrow (q', \sigma)\)

- Conditional: \( q : \text{if} (b) \{ q : \ldots \} \text{else} \{ q' : \ldots \} \)
  - if \([b](\sigma) = \text{True}\) then \((q, \sigma) \rightarrow (q', \sigma)\)
  - if \([b](\sigma) = \text{False}\) then \((q, \sigma) \rightarrow (q', \sigma)\)

Static Semantics

\[
\{ P [e/x] \} x := e \{ P \}
\]

\[
\{ P \} c_1 \{ Q \} \{ Q \} c_2 \{ R \}
\]

\[
\{ P \wedge e \} c_1 \{ Q \} \{ P \wedge \neg e \} c_2 \{ Q \}
\]

\[
\{ P \} \text{if} (e) \text{then} c_1 \text{else} c_2 \{ Q \}
\]

Rule of consequence:

\[
P \Rightarrow P' \{ P' \} c \{ Q' \} \quad Q' \Rightarrow Q
\]

Soundness

- For each Hoare triple \( \{ P \} c \{ Q \} \) deduced by the static semantics
  \( \forall \sigma, \sigma'. (\sigma \vdash P \wedge (c, \sigma) \Rightarrow \sigma' \vdash Q) \)
  holds
- Notice how in each one of our theories, AI, types, AS we have
  - Dynamic semantics
  - Static semantics
  - Soundness (connecting the two)

Example

\[
\{ x > y \text{ and } x = x_0 \text{ and } y = y_0 \}
\]

\[
t = x - y;
\]

\[
\text{while} (t > 0) \{
  x = x - 1;
  y = y + 1;
  t = t - 1;
\}
\]

\[
\{ x = y_0 \text{ and } y = x_0 \}
\]
Example

\[
\{ x > y \text{ and } x = x_0 \text{ and } y = y_0 \}
\]
\[
\text{while } (t > 0) \{ \\
\quad \{ x_0 + t - 1 \text{ and } y_0 + t \text{ and } t \geq 0 \} \Rightarrow \{ x = x_0 + t - 1 \text{ and } y = x_0 - t \text{ and } t \geq 0 \} \\
\quad x = x - 1; \\
\quad y = y + 1; \\
\quad t = t - 1; \\
\quad \{ x_0 + t \text{ and } y_0 - t \text{ and } t \geq 0 \} \\
\} \Rightarrow \{ x = y_0 \text{ and } y = x_0 \} \quad 13
\]


Weakest Precondition

\[
\wp(x := e, Q) = Q[e/x] \\
\wp(c_1 ; c_2 , Q) = \wp(c_1 , \wp(c_2 , Q)) \\
\wp(\text{if } (e) \text{ then } c_1 \text{ else } c_2 , Q) = \\
\quad e \land \wp(c_1 , Q) \lor \neg e \land \wp(c_2 , Q) \\
\wp(\text{while } (e) \text{ do } c , Q) = \\
\quad W = e \Rightarrow \wp(c , W) \land \neg e \Rightarrow Q
\]

Outline

- SAT/SMT solvers
- Axiomatic semantics
  - IMP
  - Semantics
  - Verification condition generation
    - Essentially, what we called "backwards reasoning" in Principles of Software
- SMT-LIB

Spring 19 CSCI 4450/6450, A. Milanova
Verification Condition

Instead of weakest precondition we compute verification condition (vc). Stronger

\[ \text{vc(while (e) do c, Q) = Inv \land Inv \Rightarrow (e \Rightarrow \text{vc(c,Inv)} \land !e \Rightarrow Q)} \]

or

\[ \text{vc(while (e) do c, Q) = Inv \land \text{// Must hold before loop!}} \]
\[ (Inv \land e) \Rightarrow \text{vc(c,Inv)} \land \text{// Must hold locally for loop} \]
\[ (Inv \land !e) \Rightarrow Q \]

Example

\[ \text{i = 5; while (i > 0) \{ vc(while (i>0) \{ i = i-1; \}, (i=0))} \]
\[ \quad \text{inv} \land \text{inv} \Rightarrow (i \Rightarrow \text{vc(i,inv)} \land !i \Rightarrow Q) \]
\[ \text{i = i - 1; \}
\]
\[ \text{vc breaks into following assertions:} \]
\[ \text{True} \Rightarrow \text{wp(i=5; \{ i>0 \})} \]
\[ \text{i} \land \text{\quad \text{i=0}} \)
\[ \{ i = 0 \} \quad \text{i} \land \text{\quad \text{i>0}} \Rightarrow \text{vc(i=i-1,(i>0))} \]
\[ \text{equiv; i} \land \text{\quad \text{i>0}} \Rightarrow \text{i-1}\text{=}0 \land \quad \text{i} \land \text{\quad \text{i>0}} \Rightarrow \text{i=}0 \]

Another Example

\[ \{ x \geq 0 \} \]
\[ i = x; \]
\[ z = 0; \]
\[ \text{while (i != 0) \{ z = z+1; \}
\]
\[ i = i-1; \}
\[ \{ x = z \} \]

SMT-LIB

SMT-LIB is a language for specifying input to SMT solvers (e.g., Z3)

- (declare-const x Int) declare an integer constant x
- (assert (> x 0)) add >0 to known facts
- (check-sat) checks if there exist an assignment that makes all known facts true; returns (sat) or (unsat)
- (get-model) print this assignment

- https://rise4fun.com/z3/tutorial

- Your homework is to write a Tiny Dafny
- Given an IMP program \{ P \} c \{ Q \} generate verification conditions in SMT-LIB
- Verify conditions with Z3
- Yet another programming language, OCaml!
- Some pitfalls
  - Function calls: (f arg1 arg2) NOT f(arg1,arg2)!
  - == is reference equality. Use (String.equal s1 s2)
Suppose we need to verify \{ P \} \Rightarrow \{ Q \}

- Generate \( \text{vc}(c,Q) \)
- We need to verify that \( P \Rightarrow \text{vc}(c,Q) \) is valid
  - A logical formula is valid when true for all inputs

**Encoding**
- Duality of satisfiability and validity:
  - F is valid iff \( \neg F \) is unsatisfiable
  - Ask: is \( \{P \Rightarrow \text{vc}(c,Q)\} \) satisfiable
  - If (unsat) program is correct
  - If (sat) our program is incorrect, we'll get model

### Example

**requires:** \( x = 1 \lor x = -2 \)

**ensures:** \( y = 0 \)

\[
\begin{align*}
y &= x + 4; \\
\text{if } (x > 0) \{ \\
\quad y &= x^2 - 1; \\
\text{else } \{ \\
\quad y &= y + x; \\
\}
\end{align*}
\]

SMT-LIB code:

```smt
(declare-const x Int)
(assert (and
(= x 1)
(= x -2))
(= x 0)
(= (+ x 4) x)
)
(check-sat)
```

### Another Example

Is this formula valid?

\[(x>0 \land x+5 > 5) \lor (x<=0 \land (x=0 \Rightarrow x + x + 5 = 5))\]

SMT-LIB code:

```smt
(declare-const x Int)
(assert (and
(> x 0)
(> (+ x 5) 5)))
(assert (not
(and (<= x 0)
(= (+ x x) 5))))
(check-sat)
```

### Outline

- Axiomatic semantics
- IMP
- Semantics
- Verification condition generation
  - Essentially, what we called “backwards reasoning” in Principles of Software
- SMT-LIB
- Catch-up: Monads
Monads

- A way to cleanly compose computations
  - E.g., f may return a value of type a or Nothing
  Composing computations becomes tedious:
    case (f s) of
    Nothing → Nothing
    Just m → case (f m) ...
- In Haskell, monads cleanly encapsulate IO and other imperative features

An Example: Cloned Sheep

type Sheep = ...
father :: Sheep → Maybe Sheep
father = ...
mother :: Sheep → Maybe Sheep
mother = ...
(Note: a sheep may have both parents, or just one)
maternalGrandfather :: Sheep → Maybe Sheep
maternalGrandfather s = case (mother s) of
  Nothing → Nothing
  Just m → father m

An Example

mothersPaternalGrandfather :: Sheep → Maybe Sheep
mothersPaternalGrandfather s = case (mother s) of
  Nothing → Nothing
  Just m → case (father m) of
  Nothing → Nothing
  Just gf → father gf

- Tedious, unreadable, difficult to maintain
- Monads help!

The Monad Type Class

Haskell’s Monad class requires 2 operations, >>= (bind) and return
class Monad m where
  // >>= (the bind operation) takes a monad // m a, and a function that takes a and turns // it into a monad m b, and returns m b
  (>>=) :: m a → (a → m b) → m b
  // return encapsulates a value into the monad
  return :: a → m a

The Maybe Monad

instance Monad Maybe where
  Nothing >>= f = Nothing
  (Just x) >>= f = f x
  return = Just

Cloned Sheep example:
mothersPaternalGrandfather s =
  (return s) >>= mother >>= father >>= father
(Note: if at any point, some function returns Nothing, Nothing gets cleanly propagated.)

The List Monad

- The List type constructor is a monad
  lis >>= f = concat (map f lis)
  return x = [x]

  Note: concat:[[a]] → [a]
e.g., concat [[1,2],[3,4],[5,6]] yields [1,2,3,4,5,6]

  Use any f s.t. f::a->[b]. f may return a list of
  0,1,2,… elements of type b, e.g.,
  > f x = [x+1]
  > [1,2,3] >>= f // yields ?
The List Monad

parents :: Sheep → [Sheep]
parents s = MaybeToList (mother s) ++
            MaybeToList (father s)

grandParents :: Sheep → [Sheep]
grandParents s = (parents s) >>= parents

do Notation (Syntactic Sugar)

> f x = x+1
> g x = x*5
> do { x <- [1,2,3]; y <- (return . f) x; (return . g) y }
   is syntactic sugar for
> [1,2,3] >>= (return . f) >>= (return . g)

> do { x <- [1,2,3]; return x } is sugar for
> [1,2,3] >>= (x -> return x)

do Notation

do { x <- expr; ... corresponds to expr >>= |x -> ... }

Note: do notation makes the argument of >>= (as in expr >>= f) explicit. Gives us an explicit reference to data encapsulated in monad

(return s) >>= mother >>= father >>= father
in do notation becomes:
do { m <- mother s; f <- father m; gf <- father f; return gf }
is syntactic sugar for
m = mother s >>= (m -> (father m) >>= (f -> (father f) >>= (gf -> return gf)))))

List Comprehensions

[ 2*i | i <- [1..] ] yields ?

[(i,j) | i <- [1,2], j <- [1..4] ] yields ?

[(i,j) | i <- [1,2], j <- [1..4] ] is syntactic sugar for
do { i <- [1,2]; j <- [1..4]; return (i,j) } which in turn
is syntactic sugar for?
[1,2] >>= (x -> [1..4] >>= (y -> return (x,y)))

Monad Laws

1. (return x) >>= f  <=>  f x
2. m >>= return  <=>  m
3. m >>= f >>= g  <=>  m >>= (x -> f x >>= g)

Adherence to monad laws is a responsibility of
the programmer who wrote the Monad instance
Ensure correctness of do notation!

So What is the Point of the Monad…

- Conveniently chains (binds) computation
- Encapsulates “mutable” state. E.g., IO:
  openFile :: FilePath -> IOMode -> Handle
  hClose :: Handle -> () -- void
  hasEOF :: Handle -> Bool
  hGetChar :: Handle -> Char

These operations break “referentially transparency”.
For example, hGetChar returns different value
when called twice in a row.
The IO Monad

- **IO a**: Computation that does some IO producing a value of type `a`. E.g., `(IO Char)`, `(IO String)`
- Unlike other monads (e.g., `Maybe`) there is no way to make `IO a` into an `a`
- The monad encapsulates “mutable” IO state
  - … and, there is no “rep exposure” of this state!
  - Access to state is only through well-defined monadic operations (e.g., `hGetChar`)

### Example

```haskell
getFileContents :: String -> IO String
getFileContents filename = do
  h <- openFile filename ReadMode
  reversed_cs <- readFileContents h []
  hClose h
  return (reverse reversed_cs)
```

### Other useful functions

- `readFile :: FilePath -> IO String`
- `writeFile :: FilePath -> String -> IO ()`

E.g.

```haskell
main = do
  [f,g] <- getArgs
  s <- readFile f
  writeFile g s
```