McMaster University
Department of Computing and Software
Dr. W. Kahl

SFWR ENG 3E03
Rules
Fall 2006

Design and Selection of Programming Languages

Operational Semantics Rules

Assignment: $$\frac{\sigma(e) \Rightarrow v}{\sigma(x := e) \Rightarrow \sigma \oplus \{x \mapsto v\}}$$

Sequence: $$\frac{\sigma_1(s_1) \Rightarrow \sigma_2 \qquad \sigma_2(s_2) \Rightarrow \sigma_3}{\sigma_1(s_1;\ s_2) \Rightarrow \sigma_3}$$

Conditional: $$\frac{\sigma(b) \Rightarrow \mathsf{True} \qquad \sigma(s_1) \Rightarrow \sigma_1}{\sigma(\textbf{if } b \textbf{ then } s_1 \textbf{ else } s_2 \textbf{ fi}) \Rightarrow \sigma_1} \qquad \frac{\sigma(b) \Rightarrow \mathsf{False} \qquad \sigma(s_2) \Rightarrow \sigma_2}{\sigma(\textbf{if } b \textbf{ then } s_1 \textbf{ else } s_2 \textbf{ fi}) \Rightarrow \sigma_2}$$

while: $$\frac{\sigma(b) \Rightarrow \mathsf{True} \qquad \sigma(s) \Rightarrow \sigma_1 \qquad \sigma_1(\textbf{while } b \textbf{ do } s \textbf{ od}) \Rightarrow \sigma_2}{\sigma(\textbf{while } b \textbf{ do } s \textbf{ od}) \Rightarrow \sigma_2} \qquad \frac{\sigma(b) \Rightarrow \mathsf{False}}{\sigma(\textbf{while } b \textbf{ do } s \textbf{ od}) \Rightarrow \sigma}$$

Axiomatic Semantics Rules

Assignment: $$\{\ P[x_1 \setminus e_1, \ldots, x_n \setminus e_n]\ \}\ (x_1, \ldots, x_n) := (e_1, \ldots e_n)\ \{\ P\ \}$$

Logical consequence: $$\frac{P \Rightarrow P' \qquad \{\ P'\ \}\ S\ \{\ Q'\ \} \qquad Q' \Rightarrow Q}{\{\ P\ \}\ S\ \{\ Q\ \}}$$

Sequence: $$\frac{\{\ P\ \}\ S_1\ \{\ R\ \} \qquad \{\ R\ \}\ S_2\ \{\ Q\ \}}{\{\ P\ \}\ S_1\ ;\ S_2\ \{\ Q\ \}}$$

Conditional: $$\frac{\{\ P \wedge b\ \}\ S_1\ \{\ Q\ \} \qquad \{\ P \wedge \neg b\ \}\ S_2\ \{\ Q\ \}}{\{\ P\ \}\ \textbf{if } b \textbf{ then } S_1 \textbf{ else } S_2 \textbf{ fi}\ \{\ Q\ \}}$$

while-Loop: $$\frac{\{\ INV \wedge b\ \}\ S\ \{\ INV\ \}}{\{\ INV\ \}\ \textbf{while } b \textbf{ do } S \textbf{ od}\ \{\ INV \wedge \neg b\ \}}$$

Assignment Tactics: $$\dfrac{\dfrac{<<\ Reasoning\ >>}{P\ \Rightarrow\ Q[x \setminus e]} \qquad \dfrac{True}{\{\ Q[x \setminus e]\ \}\ x := e\ \{\ Q\ \}}\ \text{(Assignment Ax .)}}{\{\ P\ \}\ x := e\ \{\ Q\ \}}\ \text{(Left Cons .)}$$

```
module SImPLEval where
import SImPL
import qualified Data.Map as Map
import Data.Map (Map) -- only the type constructor is imported unqualified

data Value1 = ValInt Integer | ValBool Bool
type State1 = Map Variable Value1

evalExpr   :: Expression → State1 → Maybe Value1
evalExpr (Var v) s = Map.lookup v s
evalExpr (Value (LitInt i)) s = Just (ValInt i) -- better: function litToVal
evalExpr (Value (LitBool b)) s = Just (ValBool b)
evalExpr (Binary (MkArithOp op) e1 e2) s = case (evalExpr e1 s, evalExpr e2 s) of
   (Just (ValInt v1), Just (ValInt v2)) → evalArithOp op v1 v2
   _                                    → Nothing
evalExpr (Binary (MkRelOp op) e1 e2) s = case (evalExpr e1 s, evalExpr e2 s) of
   (Just (ValInt v1), Just (ValInt v2)) → evalRelOp op v1 v2
   _                                    → Nothing
evalExpr (Binary (MkBoolOp op) e1 e2) s = case (evalExpr e1 s, evalExpr e2 s) of
   (Just (ValBool b1), Just (ValBool b2)) → evalBoolOp op b1 b2
   _                                      → Nothing
evalExpr (Unary Not e) s = case evalExpr e s of
  Just (ValBool b) → Just (ValBool (not b))
  _                → Nothing

evalArithOp :: ArithOp → Integer → Integer → Maybe Value1
evalRelOp :: RelOp → Integer → Integer → Maybe Value1
evalBoolOp :: BoolOp → Bool → Bool → Maybe Value1
   -- implementations omitted since not relevant for Final

interpStmt :: Statement → (State1 → Maybe State1)
interpStmt (Assignment var e) s = case evalExpr e s of
  Just val → Just (Map.insert var val s)
  Nothing → Nothing
interpStmt (Conditional cond sThen sElse) s = case evalExpr cond s of
   Just (ValBool True ) → (interpStmt sThen) s
   Just (ValBool False) → (interpStmt sElse) s
   _                    → Nothing
interpStmt (Loop cond body) s = case evalExpr cond s of
   Just (ValBool False) → Just s
   Just (ValBool True ) → case interpStmt body s of
      Just s1 → interpStmt (Loop cond body) s1
      _ → Nothing
   Just (ValInt i) → Nothing
   Nothing → Nothing
interpStmt (MkBlock stmts) s = interpBlock stmts s

interpBlock :: [Statement] → (State1 → Maybe State1)
interpBlock [] = Just
interpBlock (stmt : stmts) = λ s → case interpStmt stmt s of
  Just s1 → interpBlock stmts s1
  Nothing → Nothing
```