CS8440: State Transition Semantics

Bill Harrison

August 29, 2016

State Machines & Security Models

- Many security models define *"is secure"* in terms of state machines; intuition:
  - partition state space into secure and insecure states
  - ... system is secure iff only secure states are reachable from secure states

State Machines & Security Models

- Many security models define *"is secure"* in terms of state machines; intuition:
  - partition state space into secure and insecure states
  - ... system is secure iff only secure states are reachable from secure states
- Today: review state machine idea in the form of *transition semantics* for a programming language

State Machines & Security Models

- Many security models define *"is secure"* in terms of state machines; intuition:
  - partition state space into secure and insecure states
  - ... system is secure iff only secure states are reachable from secure states
- Today: review state machine idea in the form of *transition semantics* for a programming language
- Transition Semantics:
  - Define the meaning of a language with transition rules. Execute input program $\mathtt{p}$ in state $\mathtt{m}_0$:
  - $(\mathtt{p}, \mathtt{m}_0) \rightarrow (\mathtt{p}_1, \mathtt{m}_1) \rightarrow \ldots \rightarrow (\mathtt{p}_n, \mathtt{m}_n) \rightarrow \ldots$
  - The example we consider is described in: Gunter, "Semantics of Programming Languages: Structures and Techniques", pages 14-17.

The Simple Imperative Language with Loops

(Abstract Syntax of While)

$$
\begin{aligned}
&\mathtt{I} \in \mathtt{Identifier}\\
&\mathtt{N} \in \mathtt{Numeral}\\
&\mathtt{B} ::= \textbf{true} \mid \textbf{false} \mid \mathtt{B}\,\textbf{and}\,\mathtt{B} \mid \mathtt{B}\,\textbf{or}\,\mathtt{B} \mid \textbf{not}\,\mathtt{B} \mid \mathtt{E} < \mathtt{E} \mid \mathtt{E} = \mathtt{E}\\
&\mathtt{E} ::= \mathtt{N} \mid \mathtt{I} \mid \mathtt{E} + \mathtt{E} \mid \mathtt{E} * \mathtt{E} \mid \mathtt{E} - \mathtt{E} \mid -\mathtt{E}\\
&\mathtt{C} ::= \textbf{skip} \mid \mathtt{C}\,;\mathtt{C} \mid \mathtt{I}{:=}\mathtt{E} \mid\\
&\qquad\ \textbf{if}\,\mathtt{B}\,\textbf{then}\,\mathtt{C}\,\textbf{else}\,\mathtt{C}\,\textbf{fi} \mid \textbf{while}\,\mathtt{B}\,\textbf{do}\,\mathtt{C}\,\textbf{od}
\end{aligned}
$$

The Memory

(Memory maps I to Z)

$$
\begin{array}{ll}
\texttt{lookup m i} & = \langle \text{current value of i} \rangle \\
\texttt{m[i}\mapsto\texttt{v]} & = \langle \text{new memory s.t. i is bound to v} \rangle
\end{array}
$$

$$
\texttt{lookup m[i}\mapsto\texttt{v] j} = \begin{cases} v & i \text{ is } j \\ lookup\ m\ j & \text{otherwise} \end{cases}
$$

E and B semantics

$$
\begin{aligned}
\mathtt{ev}_{\mathrm{E}}\ \mathtt{n}\ \mathtt{m} &= \mathtt{n} \\
\mathtt{ev}_{\mathrm{E}}\ \mathtt{i}\ \mathtt{m} &= \mathtt{lookup}\ \mathtt{i}\ \mathtt{m} \\
\mathtt{ev}_{\mathrm{E}}\ {-\mathtt{e}}\ \mathtt{m} &= -(\mathtt{ev}_{\mathrm{E}}\ \mathtt{e}\ \mathtt{m}) \\
&\ldots \\
\mathtt{ev}_{\mathrm{B}}\ \textbf{true}\ \mathtt{m} &= \mathtt{true} \\
\mathtt{ev}_{\mathrm{B}}\ (\mathtt{e}_1 = \mathtt{e}_2)\ \mathtt{m} &= (\mathtt{ev}_{\mathrm{E}}\ \mathtt{e}_1\ \mathtt{m}\ =_{\mathrm{Z}}\ \mathtt{ev}_{\mathrm{E}}\ \mathtt{e}_2\ \mathtt{m}) \\
&\ldots
\end{aligned}
$$

E and B semantics

$$
\begin{aligned}
\mathtt{ev}_{\mathrm{E}}\ \mathtt{n}\ \mathtt{m} &= \mathtt{n} \\
\mathtt{ev}_{\mathrm{E}}\ \mathtt{i}\ \mathtt{m} &= \mathtt{lookup}\ \mathtt{i}\ \mathtt{m} \\
\mathtt{ev}_{\mathrm{E}}\ {-\mathtt{e}}\ \mathtt{m} &= -(\mathtt{ev}_{\mathrm{E}}\ \mathtt{e}\ \mathtt{m}) \\
\ldots & \\
\mathtt{ev}_{\mathrm{B}}\ \textbf{true}\ \mathtt{m} &= \mathtt{true} \\
\mathtt{ev}_{\mathrm{B}}\ (\mathtt{e}_1 = \mathtt{e}_2)\ \mathtt{m} &= (\mathtt{ev}_{\mathrm{E}}\ \mathtt{e}_1\ \mathtt{m}\ =_{\mathrm{Z}}\ \mathtt{ev}_{\mathrm{E}}\ \mathtt{e}_2\ \mathtt{m}) \\
\ldots &
\end{aligned}
$$

Question: what are the types of $\mathtt{ev}_{\mathrm{E}}$ and $\mathtt{ev}_{\mathrm{B}}$?

Transition Semantics of While

$$(\mathtt{i} := \mathtt{e}, \mathtt{m}) \to \mathtt{m}[\mathtt{i} \mapsto \mathtt{ev}_{\mathtt{E}}\ \mathtt{e}\ \mathtt{m}] \qquad (\textbf{skip}, \mathtt{m}) \to \mathtt{m}$$

$$\frac{(c_1, m) \to (c_1', m')}{(c_1; c_2, m) \to (c_1'; c_2, m')} \qquad \frac{(c_1, m) \to m'}{(c_1; c_2, m) \to (c_2, m')}$$

$$\frac{ev_B\ \ b\ \ m = true}{(\textbf{if}\ \ b\ \textbf{then}\ \ c_1\ \textbf{else}\ \ c_2\ \textbf{fi}, m) \to (c_1, m)}$$

$$\frac{ev_B\ \ b\ \ m = false}{(\textbf{if}\ \ b\ \textbf{then}\ \ c_1\ \textbf{else}\ \ c_2\ \textbf{fi}, m) \to (c_2, m)}$$

Transition Semantics of While (cont'd)

$$\frac{ev_B\ \ b\ \ m = true}{(\textbf{while}\ b\ \textbf{do}\ c\ \textbf{od}, m) \rightarrow (c; \textbf{while}\ b\ \textbf{do}\ c\ \textbf{od}, m)}$$

$$\frac{ev_B\ \ b\ \ m = false}{(\textbf{while}\ b\ \textbf{do}\ c\ \textbf{od}, m) \rightarrow m}$$

In Class Exercise

Formulate the transition semantics for While in Haskell.

1. Define each of E, B, and C as **data** declarations;
2. Define the Memory data type next;
3. Define $\mathtt{ev}_{\mathrm{E}}$ and $\mathtt{ev}_{\mathrm{B}}$ as Haskell functions;
4. Finally, define the transitions for C as a function of type: $(\mathtt{C}, \mathtt{Memory}) \rightarrow \mathtt{Memory}$.

1. Define each of E, B, and C as **data** declarations

```
type Ident  = String
type Number = Int
```

1. Define each of E, B, and C as **data** declarations

```
type Ident  = String
type Number = Int

data E =
```

1. Define each of E, B, and C as **data** declarations

```
type Ident  = String
type Number = Int

data E = N Number
       | I Ident
       | Add E E | Mult E E | Subt E E | Inv E

data B =
```

1. Define each of E, B, and C as **data** declarations

```
type Ident  = String
type Number = Int

data E = N Number
       | I Ident
       | Add E E | Mult E E | Subt E E | Inv E

data B = T | F | Conj B B
       | Disj B B | Negt B | LTC E E | EQL E E

data C =
```

1. Define each of E, B, and C as **data** declarations

```
type Ident  = String
type Number = Int

data E = N Number
       | I Ident
       | Add E E | Mult E E | Subt E E | Inv E

data B = T | F | Conj B B
       | Disj B B | Negt B | LTC E E | EQL E E

data C = Skip
       | Asn Ident E
       | Seq C C
       | IfElse B C C | While B C
```

2. Define the Memory data type.

```
type Memory = [(Ident, Number)]

--
-- memory look-up
--
lkup :: Memory -> Ident -> Number
lkup ((x, n):ms) x' = if x == x' then n else lkup ms x'
lkup [ ] _          = error "oh snap, you did something bad!"
```

3. Define $\mathtt{ev}_{\mathtt{E}}$ and $\mathtt{ev}_{\mathtt{B}}$ as Haskell functions

```
--
-- evaluate a Boolean expression:
--
evB ::
```

3. Define $\mathtt{ev}_{\mathrm{E}}$ and $\mathtt{ev}_{\mathrm{B}}$ as Haskell functions

```
--
-- evaluate a Boolean expression:
--
evB :: Memory -> B -> Bool
```

3. Define ev_{E} and ev_{B} as Haskell functions

```
--
-- evaluate a Boolean expression:
--
evB :: Memory -> B -> Bool
evB _  T            = True
```

3. Define $\mathtt{ev}_{\mathrm{E}}$ and $\mathtt{ev}_{\mathrm{B}}$ as Haskell functions

```
--
-- evaluate a Boolean expression:
--
evB :: Memory -> B -> Bool
evB _  T           = True
evB _  F           = False
```

3. Define ev_{E} and ev_{B} as Haskell functions

```
--
-- evaluate a Boolean expression:
--
evB :: Memory -> B -> Bool
evB _  T            = True
evB _  F            = False
evB m (Conj b1 b2) = (evB m b1) && (evB m b2)
```

3. Define $\mathtt{ev}_{\mathrm{E}}$ and $\mathtt{ev}_{\mathrm{B}}$ as Haskell functions

```
--
-- evaluate a Boolean expression:
--
evB :: Memory -> B -> Bool
evB _  T             = True
evB _  F             = False
evB m (Conj b1 b2) = (evB m b1) && (evB m b2)
evB m (Disj b1 b2) = (evB m b1) || (evB m b2)
evB m (Negt b)       = not (evB m b)
evB m (LTC e1 e2)  = (evE m e1) < (evE m e2)
evB m (EQL e1 e2)  = (evE m e1) == (evE m e2)
--
-- evaluate an arithmetic expression:
--
evE ::
```

3. Define ev_E and ev_B as Haskell functions

```
--
-- evaluate a Boolean expression:
--
evB :: Memory -> B -> Bool
evB _  T            = True
evB _  F            = False
evB m (Conj b1 b2) = (evB m b1) && (evB m b2)
evB m (Disj b1 b2) = (evB m b1) || (evB m b2)
evB m (Negt b)      = not (evB m b)
evB m (LTC e1 e2)  = (evE m e1) < (evE m e2)
evB m (EQL e1 e2)  = (evE m e1) == (evE m e2)
--
-- evaluate an arithmetic expression:
--
evE :: Memory -> E -> Number
```

3. Define $\mathtt{ev}_{\mathtt{E}}$ and $\mathtt{ev}_{\mathtt{B}}$ as Haskell functions

```
--
-- evaluate a Boolean expression:
--
evB :: Memory -> B -> Bool
evB _  T              = True
evB _  F              = False
evB m (Conj b1 b2) = (evB m b1) && (evB m b2)
evB m (Disj b1 b2) = (evB m b1) || (evB m b2)
evB m (Negt b)       = not (evB m b)
evB m (LTC e1 e2)   = (evE m e1) < (evE m e2)
evB m (EQL e1 e2)   = (evE m e1) == (evE m e2)
--
-- evaluate an arithmetic expression:
--
evE :: Memory -> E -> Number
evE _ (N n)          = n
```

3. Define $\mathtt{ev}_{\mathrm{E}}$ and $\mathtt{ev}_{\mathrm{B}}$ as Haskell functions

```
--
-- evaluate a Boolean expression:
--
evB :: Memory -> B -> Bool
evB _  T            = True
evB _  F            = False
evB m (Conj b1 b2) = (evB m b1) && (evB m b2)
evB m (Disj b1 b2) = (evB m b1) || (evB m b2)
evB m (Negt b)     = not (evB m b)
evB m (LTC e1 e2)  = (evE m e1) < (evE m e2)
evB m (EQL e1 e2)  = (evE m e1) == (evE m e2)
--
-- evaluate an arithmetic expression:
--
evE :: Memory -> E -> Number
evE _ (N n)         = n
evE m (I x)         = lkup m x
```

3. Define $\mathtt{ev}_{\mathrm{E}}$ and $\mathtt{ev}_{\mathrm{B}}$ as Haskell functions

```haskell
--
-- evaluate a Boolean expression:
--
evB :: Memory -> B -> Bool
evB _  T              = True
evB _  F              = False
evB m (Conj b1 b2) = (evB m b1) && (evB m b2)
evB m (Disj b1 b2) = (evB m b1) || (evB m b2)
evB m (Negt b)      = not (evB m b)
evB m (LTC e1 e2)  = (evE m e1) < (evE m e2)
evB m (EQL e1 e2)  = (evE m e1) == (evE m e2)
--
-- evaluate an arithmetic expression:
--
evE :: Memory -> E -> Number
evE _ (N n)          = n
evE m (I x)          = lkup m x
evE m (Add n1 n2)  = (evE m n1) + (evE m n2)
```

3. Define $\mathtt{ev}_{\mathrm{E}}$ and $\mathtt{ev}_{\mathrm{B}}$ as Haskell functions

```
--
-- evaluate a Boolean expression:
--
evB :: Memory -> B -> Bool
evB _  T            = True
evB _  F            = False
evB m (Conj b1 b2) = (evB m b1) && (evB m b2)
evB m (Disj b1 b2) = (evB m b1) || (evB m b2)
evB m (Negt b)     = not (evB m b)
evB m (LTC e1 e2)  = (evE m e1) < (evE m e2)
evB m (EQL e1 e2)  = (evE m e1) == (evE m e2)
--
-- evaluate an arithmetic expression:
--
evE :: Memory -> E -> Number
evE _ (N n)        = n
evE m (I x)        = lkup m x
evE m (Add n1 n2)  = (evE m n1) + (evE m n2)
evE m (Mult n1 n2) = (evE m n1) * (evE m n2)
evE m (Subt n1 n2) = (evE m n1) - (evE m n2)
evE m (Inv n)      = -(evE m n)
```

4. Define C transitions as a function of type: $(\texttt{C}, \texttt{Memory}) \rightarrow \texttt{Memory}$

```
type Trans  = (C, Memory) -> Memory

exec :: Trans
exec (Skip,m)              = ...
exec (Asn i e,m)          = ...
exec (Seq c1 c2,m)        = ...
exec (IfElse b c1 c2,m) = ...
exec (While b c,m)        = ...
```

$$(\mathtt{i} := \mathtt{e}, \mathtt{m}) \rightarrow \mathtt{m}[\mathtt{i} \mapsto \mathtt{ev_E} \ \mathtt{e} \ \mathtt{m}] \qquad (\textbf{skip}, \mathtt{m}) \rightarrow \mathtt{m}$$

$$\frac{(c_1, m) \rightarrow (c_1', m')}{(c_1 \ ; \ c_2, \ m) \rightarrow (c_1' \ ; \ c_2, m')} \qquad \frac{(c_1, m) \rightarrow m'}{(c_1 \ ; \ c_2, \ m) \rightarrow (c_2, m')}$$

$$\frac{ev_B \ \ b \ \ m \ = \ true}{(\textbf{if} \ \ b \ \ \textbf{then} \ \ c_1 \ \ \textbf{else} \ \ c_2 \ \ \textbf{fi}, m) \rightarrow (c_1, m)} \qquad \frac{ev_B \ \ b \ \ m \ = \ false}{(\textbf{if} \ \ b \ \ \textbf{then} \ \ c_1 \ \ \textbf{else} \ \ c_2 \ \ \textbf{fi}, m) \rightarrow (c_2, m)}$$

4. Define C transitions as a function of type: $(\mathtt{C}, \mathtt{Memory}) \rightarrow \mathtt{Memory}$

```
type Trans  = (C, Memory) -> Memory

exec :: Trans
exec (Skip,m)             = m
exec (Asn i e,m)          = (i,evE m e) : m
exec (Seq c1 c2,m)        = exec (c2,exec (c1,m))
exec (IfElse b c1 c2,m) = if evB m b
                              then exec (c1,m)
                              else exec (c2,m)
exec (While b c,m)        = if evB m b
                              then exec (While b c,exec (c,m))
                              else m
```