Declarative Computation Model

Kernel language semantics
Basic concepts, the abstract machine (CTM 2.4.1-2.4.2)

Carlos Varela
RPI
October 3, 2014

Adapted with permission from:
Seif Haridi
KTH
Peter Van Roy
UCL
Sequential declarative computation model

- The single assignment store
  - declarative (dataflow) variables
  - partial values (variables and values are also called entities)

- The kernel language syntax

- The kernel language semantics
  - The environment: maps textual variable names (variable identifiers) into entities in the store
  - Interpretation (execution) of the kernel language elements (statements) by the use of an abstract machine
  - Abstract machine consists of an execution stack of statements transforming the store
Kernel language syntax

The following defines the syntax of a statement, \( \langle s \rangle \) denotes a statement

\[
\langle s \rangle ::= \text{skip} \quad \text{empty statement} \\
\ | \quad \langle x \rangle = \langle y \rangle \quad \text{variable-variable binding} \\
\ | \quad \langle x \rangle = \langle v \rangle \quad \text{variable-value binding} \\
\ | \quad \langle s_1 \rangle \langle s_2 \rangle \quad \text{sequential composition} \\
\ | \quad \text{local } \langle x \rangle \text{ in } \langle s_1 \rangle \text{ end} \quad \text{declaration} \\
\ | \quad \text{if } \langle x \rangle \text{ then } \langle s_1 \rangle \text{ else } \langle s_2 \rangle \text{ end} \quad \text{conditional} \\
\ | \quad \{ \langle x \rangle \langle y_1 \rangle \ldots \langle y_n \rangle \} \quad \text{procedural application} \\
\ | \quad \text{case } \langle x \rangle \text{ of } \langle \text{pattern} \rangle \text{ then } \langle s_1 \rangle \text{ else } \langle s_2 \rangle \text{ end} \quad \text{pattern matching}
\]

\[
\langle v \rangle ::= \text{proc } \{ \$ \langle y_1 \rangle \ldots \langle y_n \rangle \} \langle s_1 \rangle \text{ end} \mid \ldots \quad \text{value expression}
\]

\[
\langle \text{pattern} \rangle ::= \ldots
\]
Examples

- local X in X = 1 end

- local X Y T Z in
  X = 5
  Y = 10
  T = (X >= Y)
  if T then Z = X else Z = Y end
  {Browse Z}
end

- local S T in
  S = proc {$ X Y} Y = X*X end
  {S 5 T}
  {Browse T}
end
Procedure abstraction

• Any statement can be abstracted to a procedure by selecting a number of the ’free’ variable identifiers and enclosing the statement into a procedure with the identifiers as parameters
• if \( X \geq Y \) then \( Z = X \) else \( Z = Y \) end
• Abstracting over all variables
  proc \{Max X Y Z\}
    if \( X \geq Y \) then \( Z = X \) else \( Z = Y \) end
  end
• Abstracting over \( X \) and \( Z \)
  proc \{LowerBound X Z\}
    if \( X \geq Y \) then \( Z = X \) else \( Z = Y \) end
  end
Computations (abstract machine)

- A computation defines how the execution state is transformed step by step from the initial state to the final state.
- A **single assignment store** $\sigma$ is a set of store variables, a variable may be unbound, bound to a partial value, or bound to a group of other variables.
- An **environment** $E$ is mapping from variable identifiers to variables or values in $\sigma$, e.g. \{X $\rightarrow$ $x_1$, Y $\rightarrow$ $x_2$\}.
- A **semantic statement** is a pair \( (\langle s \rangle, E) \) where $\langle s \rangle$ is a statement.
- $ST$ is a stack of **semantic statements**.
Computations (abstract machine)

- A computation defines how the execution state is transformed step by step from the initial state to the final state.
- The *execution state* is a pair
  \[(ST, \sigma)\]
- \(ST\) is a stack of semantic statements.
- A *computation* is a sequence of execution states
  \[(ST_0, \sigma_0) \rightarrow (ST_1, \sigma_1) \rightarrow (ST_2, \sigma_2) \rightarrow \ldots\]
Semantics

- To execute a program (i.e., a statement) $\langle s \rangle$ the initial execution state is
  \[( [ (\langle s \rangle, \emptyset) ] , \emptyset )\]
- $ST$ has a single semantic statement $(\langle s \rangle, \emptyset)$
- The environment $E$ is empty, and the store $\sigma$ is empty
- $[ ... ]$ denotes the stack
- At each step the first element of $ST$ is popped and execution proceeds according to the form of the element
- The final execution state (if any) is a state in which $ST$ is empty
• The semantic statement is
  \((\text{skip}, E)\)
• Continue to next execution step
• The semantic statement is 
  \((\text{skip}, E)\)
• Continue to next execution step
Sequential composition

- The semantic statement is
  $$(\langle s_1 \rangle \langle s_2 \rangle , E)$$
- Push $$(\langle s_2 \rangle , E)$$ and then push $$(\langle s_1 \rangle , E)$$ on $ST$
- Continue to next execution step

\[
\begin{array}{c|c}
(\langle s_1 \rangle \langle s_2 \rangle , E) & ST \\
\end{array}
\quad + \quad
\begin{array}{c|c}
\sigma & \\
\end{array}
\quad \rightarrow \quad
\begin{array}{c|c}
(\langle s_1 \rangle , E) & \\
(\langle s_2 \rangle , E) & ST \\
\end{array}
\quad + \quad
\begin{array}{c|c}
\sigma & \\
\end{array}
\]
Calculating with environments

- $E$ is mapping from identifiers to entities (both store variables and values) in the store.

- The notation $E(\langle y \rangle)$ retrieves the entity $x$ associated with the identifier $\langle y \rangle$ from the store.

- The notation $E + \{ \langle y \rangle_1 \rightarrow x_1, \langle y \rangle_2 \rightarrow x_2, \ldots, \langle y \rangle_n \rightarrow x_n \}$ denotes a new environment $E'$ constructed from $E$ by adding the mappings $\{ \langle y \rangle_1 \rightarrow x_1, \langle y \rangle_2 \rightarrow x_2, \ldots, \langle y \rangle_n \rightarrow x_n \}$.

- $E'(\langle z \rangle)$ is $x_k$ if $\langle z \rangle$ is equal to $\langle y \rangle_k$, otherwise $E'(\langle z \rangle)$ is equal to $E(\langle z \rangle)$.

- The notation $E\mid\{\langle y \rangle_1, \langle y \rangle_2, \ldots, \langle y \rangle_n\}$ denotes the projection of $E$ onto the set $\{\langle y \rangle_1, \langle y \rangle_2, \ldots, \langle y \rangle_n\}$, i.e., $E$ restricted to the members of the set.
Calculating with environments (2)

• \( E = \{ X \rightarrow 1, \ Y \rightarrow [2 \ 3], \ Z \rightarrow x_i \} \)

• \( E' = E + \{ X \rightarrow 2 \} \)

• \( E'(X) = 2, \quad E(X) = 1 \)

• \( E|_{\{X,Y\}} \) restricts \( E \) to the 'domain' \( \{X,Y\} \), i.e., it is equal to \( \{ X \rightarrow 1, \ Y \rightarrow [2 \ 3] \} \)
Calculating with environments (3)

- local X in
  
  \[
  X = 1 \quad (E)
  \]

  local X in
  
  \[
  X = 2 \quad (E')
  \]

  \{Browse X\}

  end \quad (E)

  \{Browse X\}

  end
Lexical scoping

- Free and bound identifier occurrences
- An identifier occurrence is *bound* with respect to a statement $\langle s \rangle$ if it is in the scope of a declaration inside $\langle s \rangle$
- A variable identifier is declared either by a ‘local’ statement, as a parameter of a procedure, or implicitly declared by a case statement
- An identifier occurrence is *free* otherwise
- In a running program every identifier is bound (i.e., declared)
Lexical scoping (2)

• proc \{P X\}
  local Y in Y = 1 \{Browse Y\} end
  X = Y
end

Free Occurrences
Bound Occurrences
Lexical scoping (3)

- `local Arg1 Arg2 in
  Arg1 = 111*111
  Arg2 = 999*999
  Res = Arg1*Arg2
end`

This is not a runnable program!
Lexical scoping (4)

- `local Res in`
  - `local Arg1 Arg2 in`
    - `Arg1 = 111*111`
    - `Arg2 = 999*999`
    - `Res = Arg1*Arg2`
  - `{Browse Res}`
- `end`
Lexical scoping (5)

```
local P Q in
  proc {P} {Q} end
  proc {Q} {Browse hello} end
local Q in
  proc {Q} {Browse hi} end
  {P}
end
end
```
42. Translate the following function to the kernel language:

```latex
fun \{AddList L1 L2\}
  case L1 of H1|T1 then
    case L2 of H2|T2 then
      H1+H2|{AddList T1 T2}
    end
  else nil end
end
```

43. Translate the following function call to the kernel language:

```latex
{Browse \{Max 5 7\}}
```
Exercises

44. Explain the difference between static scoping and dynamic scoping. Give an example program that produces different results with static and dynamic scoping.

45. Think of a reason why static scoping may be preferable to dynamic scoping. Think of a reason why dynamic scoping may be preferable to static scoping.