Concurrent and Distributed Programming
(Declarative Computation Model)

Luis Quesada

Research Scientist
Cork Constraint Computation Centre

Invited Lecturer
Pontificia Universidad Javeriana

January 31, 2009

Remark: Most of the material in these slides has been taken from the book Concepts, Techniques, and Models of Computer Programming by Peter Van Roy and Seif Haridi.
Outline

1. Single-assignment store
2. Kernel Language
3. Kernel language semantics
4. Syntactic sugar
Outline

1. Single-assignment store
2. Kernel Language
3. Kernel language semantics
4. Syntactic sugar
Single-assignment variables

- **Single-assignment variables** (aka declarative variables) are variables that are initially unbound and that can be bound to one value only.

- A **single-assignment store** is a set of single-assignment variables.

  
  \[
  \begin{array}{c}
  x_1 \quad 314 \\
  x_2 \quad 1 \rightarrow 2 \rightarrow 3 \quad \text{nil} \\
  x_3 \quad \text{unbound}
  \end{array}
  \]

  
- For now, we assume that lists are values.
A **value store** is a store where all the variables are values.

Standard ML, Haskell and Scheme are among the languages that use a value store.

A **cell store** is a store where all the variables are cells (i.e., multiple assignment variables).

Object oriented languages such as Smalltalk, C++, and Java are among the languages that use a cell store.

In a single-assignment store, the fact that creating a variable and binding it are done separately allows us to have partial values!

If a variable is used before it is bound the execution of the program waits until the variable is bound.
A **variable identifier** is a textual name that refers to a store entity.

An **environment** is a mapping from variable identifiers to store entities.

\{X \rightarrow x_1\} is an example of an environment where the identifier X is mapped to the variable $x_1$:
A **partial value** is a data structure that contains unbound variables.

An example of a partial value:
The value obtained after binding $Y$ to 25:

![Diagram showing the value obtained after binding $Y$ to 25]
Variables can be bound to variables.

Assuming that Identifier $X$ refers to the variable $x_1$ and Identifier $Y$ refer to the variable $x_2$, the following is the situation obtained after executing the statement $X = Y$: 

![Diagram showing variable-variable binding]
The declarative kernel language

\[ \langle s \rangle ::= \]
\[ \text{skip} \quad \text{Empty statement} \]
\[ \langle s \rangle_1 \langle s \rangle_2 \quad \text{Statement sequence} \]
\[ \text{local } \langle x \rangle \text{ in } \langle s \rangle \text{ end} \quad \text{Variable creation} \]
\[ \langle x \rangle_1 = \langle x \rangle_2 \quad \text{Variable-variable binding} \]
\[ \langle x \rangle = \langle v \rangle \quad \text{Value creation} \]
\[ \text{if } \langle x \rangle \text{ then } \langle s \rangle_1 \text{ else } \langle s \rangle_2 \text{ end} \quad \text{Conditional} \]
\[ \text{case } \langle x \rangle \text{ of } \langle \text{pattern} \rangle \text{ then } \langle s \rangle_1 \text{ else } \langle s \rangle_2 \text{ end} \quad \text{Pattern matching} \]
\[ \{ \langle x \rangle \langle y \rangle_1 \ldots \langle y \rangle_n \} \quad \text{Procedure application} \]
**Value expressions in the declarative kernel language**

\[
\begin{align*}
\langle v \rangle & ::= \langle \text{number} \rangle \mid \langle \text{record} \rangle \mid \langle \text{procedure} \rangle \\
\langle \text{number} \rangle & ::= \langle \text{int} \rangle \mid \langle \text{float} \rangle \\
\langle \text{record}, \langle \text{pattern} \rangle \rangle & ::= \langle \text{literal} \rangle \\
& \quad \mid \langle \text{literal} \rangle(\langle \text{feature} \rangle_1: \langle x \rangle_1 \ldots \langle \text{feature} \rangle_n: \langle x \rangle_n) \\
\langle \text{procedure} \rangle & ::= \text{proc} \{ \$ \langle x \rangle_1 \ldots \langle x \rangle_n \} \langle s \rangle \text{ end} \\
\langle \text{literal} \rangle & ::= \langle \text{atom} \rangle \mid \langle \text{bool} \rangle \\
\langle \text{feature} \rangle & ::= \langle \text{atom} \rangle \mid \langle \text{bool} \rangle \mid \langle \text{int} \rangle \\
\langle \text{bool} \rangle & ::= \langle \text{true} \rangle \mid \langle \text{false} \rangle
\end{align*}
\]
The declarative model is **dynamic typed**, i.e., the variable type is only known when the variable is bound.
## Example of basic operations

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
<th>Argument type</th>
</tr>
</thead>
<tbody>
<tr>
<td>A==B</td>
<td>Equality comparison</td>
<td>Value</td>
</tr>
<tr>
<td>A=B</td>
<td>Nonequality comparison</td>
<td>Value</td>
</tr>
<tr>
<td>{Is Procedure P}</td>
<td>Test if procedure</td>
<td>Value</td>
</tr>
<tr>
<td>A&lt;=B</td>
<td>Less than or equal comparison</td>
<td>Number or Atom</td>
</tr>
<tr>
<td>A&lt;B</td>
<td>Less than comparison</td>
<td>Number or Atom</td>
</tr>
<tr>
<td>A&gt;=B</td>
<td>Greater than or equal comparison</td>
<td>Number or Atom</td>
</tr>
<tr>
<td>A&gt;B</td>
<td>Greater than comparison</td>
<td>Number or Atom</td>
</tr>
<tr>
<td>A+B</td>
<td>Addition</td>
<td>Number</td>
</tr>
<tr>
<td>A-B</td>
<td>Subtraction</td>
<td>Number</td>
</tr>
<tr>
<td>A*B</td>
<td>Multiplication</td>
<td>Number</td>
</tr>
<tr>
<td>A div B</td>
<td>Division</td>
<td>Int</td>
</tr>
<tr>
<td>A mod B</td>
<td>Modulo</td>
<td>Int</td>
</tr>
<tr>
<td>A/B</td>
<td>Division</td>
<td>Float</td>
</tr>
<tr>
<td>{Arity R}</td>
<td>Arity</td>
<td>Record</td>
</tr>
<tr>
<td>{Label R}</td>
<td>Label</td>
<td>Record</td>
</tr>
<tr>
<td>R.F</td>
<td>Field selection</td>
<td>Record</td>
</tr>
</tbody>
</table>
Outline

1. Single-assignment store
2. Kernel Language
3. Kernel language semantics
4. Syntactic sugar
Variable identifiers and static scoping

- A **local statement** creates a variable and associates it with its identifier.
- The **scope** of an identifier is the program region in which the identifier refers to a particular variable.
- This program displays 2, and then 1 since the identifier $X$ refers to different variables through the execution.
Informal semantics of procedures (I)

- This procedure binds $Z$ to the maximum of $X$ and $Y$:

  \[
  \text{proc } \{ \text{Max } X \ Y \ ?Z \} \\
  \quad \text{if } X \geq Y \ \text{then } Z = X \ \text{else } Z = Y \ \text{end} \\
  \text{end}
  \]

- Calling $\{ \text{Max } 3 \ 5 \ C \}$ binds $C$ to 5 since $X$, $Y$ and $Z$ are bound 5, 6 and $C$ respectively.

- This way of passing parameters is called call by reference.
What does the call {LB 5 Z} bind Z to?

local Y LB in
    Y=10
    proc {LB X ?Z}
        if X>=Y then Z=X else Z=Y end
    end
    local Y=15 Z in
        {LB 5 Z}
    end
end
What does the call \{LB 5 Z\} bind Z to?

```
local Y LB in
  Y=10
  proc {LB X ?Z}
    if X>=Y then Z=X else Z=Y end
  end
  local Y=15 Z in
    {LB 5 Z}
  end
end
```

It binds it to 10 since it is the binding \( Y=10 \) at the procedure definition that is important.
The abstract machine (Definitions)

- An **abstract machine** is theoretical model of a computer.
- A **single-assignment store** $\sigma$ is a set of single-assignment variables.
- An **environment** $E$ is a mapping from identifiers to variables in a store.
- A **semantic statement** is a pair $(\langle s \rangle, E)$ where $\langle s \rangle$ is a statement and $E$ is an environment.
- An **execution state** is a pair $(ST, \sigma)$, where $ST$ is a stack of semantic statements.
- A **computation** is a sequence of execution state.
Operation on environments

- **Adjunction:** $E + \{\langle x \rangle \rightarrow x\}$ denotes an environment $E'$ constructed from $E$ by adding the mapping $\{\langle x \rangle \rightarrow x\}$, which overrides any other mapping from the identifier $\langle x \rangle$.

- **Restriction:** $E|_{\langle x_1 \rangle \rightarrow x_1, \ldots, \langle x \rangle \rightarrow x_n}$ denotes an environment $E'$ such that $\text{dom}(E') = \text{dom}(E) \cap \{\langle x_1 \rangle, \ldots, \langle x \rangle\}$ and $E'(%x\rangle) = E(%x\rangle)$ for all $%x\rangle \in \text{dom}(E')$. 
Semantics of nonsuspendable statements (I)

- **The skip statement.** The execution of the corresponding semantics statement has not effect in any of the components of the system:

\[(\text{skip, } E)\mid ST, \sigma) \mapsto (ST, \sigma)\]

- \(\text{(skip, } E)\mid ST, \sigma)\) denotes a stack of semantic statements whose top element is \((\text{skip, } E)\).
- \(S_1 \mid S_2 \mid \ldots \mid S_n \mid ST\) denotes a stack of semantic statements whose top elements are \(S_1, S_2, \ldots, S_n\) and the remaining elements are the ones in \(ST\).
Sequential composition. The two statements in the composition give place to two semantic statements with the same environment:

\[
((\langle s \rangle_1 \langle s \rangle_2, E) | ST, \sigma) \mapsto ((\langle s \rangle_1, E) | (\langle s \rangle_2, E) | ST, \sigma)
\]

Variable declaration. A new variable is created in the store and the environment is updated accordingly:

\[
((\text{local } \langle x \rangle \text{ in } \langle s \rangle \text{ end}, E) | ST, \sigma) \mapsto ((\langle s \rangle, E + \{ \langle x \rangle \mapsto x \}) | ST, \sigma \cup \{ x \})
\]

The elements in $\sigma$ can be of three types:

- $x$, which denotes that $x$ is unbound.
- $x = v$, which denotes that $x$ is bound to the value $v$.
- $x = y$, which denotes that $x$ is bound to the variable $y$. 
Semantics of nonsuspendable statements (III)

- **Variable-variable binding.** The two variables are bound in the store:
  \[
  (((\langle x \rangle_1 = \langle x \rangle_2, E)|ST, \sigma) \mapsto (ST, \sigma \cup \{x_1 = x_2\})
  \]

- **Variable-value binding.** The variable is bound the value in the store:
  \[
  (((\langle x \rangle = v, E)|ST, \sigma) \mapsto (ST, \sigma \cup \{x = v\})
  \]
Semantics of nonsuspendable statements (IV)

- **Procedure creation.** The creation of a procedure is a particular case of variable-value binding since

  \[
  \text{proc } \{ \langle x \rangle \langle y \rangle_1 \ldots \langle y \rangle_n \} \langle s \rangle \text{ end}
  \]

  is syntactic sugar for

  \[
  \langle x \rangle = \text{proc } \{ \$ \langle y \rangle_1 \ldots \langle y \rangle_n \} \langle s \rangle \text{ end}
  \]

- A **free identifier** of a procedure is an identifier that occurs in the body of the procedure, but it is neither among the formal parameters nor declared in the body.

- The **contextual environment** of a procedure is an environment that maps the free identifiers of the procedure to their corresponding variables.

  \[
  ((\langle x \rangle = \text{proc } \{ \$ \langle y \rangle_1 \ldots \langle y \rangle_n \} \langle s \rangle \text{ end}, E)|ST, \sigma) \mapsto \\
  (ST, \sigma \cup \{ x = (\text{proc } \{ \$ \langle y \rangle_1 \ldots \langle y \rangle_n \} \langle s \rangle \text{ end}, CE) \})
  \]

  \[CE\] is the contextual environment of the procedure.
The execution of any of the following statements will block if \( \langle x \rangle \) is not bound:

\[
\langle s \rangle ::= ... \\
| \text{if } \langle x \rangle \text{ then } \langle s \rangle_1 \text{ else } \langle s \rangle_2 \text{ end} \\
| \text{case } \langle x \rangle \text{ of } \langle \text{pattern} \rangle \text{ then } \langle s \rangle_1 \text{ else } \langle s \rangle_2 \text{ end} \\
| \{ \langle x \rangle \langle y \rangle_1 \ldots \langle y \rangle_n \}
\]

These statements have an activation condition, i.e., a condition that must be true for execution to continue.
Semantics of suspendable statements (II)

- **Conditional.** Given the current state

\[
((\text{if } \langle x \rangle \text{ then } \langle s \rangle_1 \text{ else } \langle s \rangle_2 \text{ end}, E)|ST, \sigma)
\]

there are three possible outcomes:

- If \( \langle x \rangle \) is bound to a value that is not boolean then an exception is raised.
- If \( \langle x \rangle \) is bound to \textit{true}:
  \[
  \mapsto (((\langle s \rangle_1, E)|ST, \sigma)
  \]
- If \( \langle x \rangle \) is bound to \textit{false}:
  \[
  \mapsto (((\langle s \rangle_2, E)|ST, \sigma)
  \]
Pattern matching. Given the current state

\[(\text{case } \langle x \rangle \text{ of } \langle \text{lit} \rangle (\langle \text{feat} \rangle_1 : \langle x \rangle_1 \ldots \langle \text{feat} \rangle_n : \langle x \rangle_n) \text{ then } \langle s \rangle_1 \text{ else } \langle s \rangle_2 \text{ end, } E)\mid ST, \sigma)\]

there are two possible outcomes:

- If the label of \(E(\langle x \rangle)\) is \(\langle \text{lit} \rangle\) and its arity is \([\langle \text{feat} \rangle_1 \ldots \langle \text{feat} \rangle_n]\):

  \[\Rightarrow (\langle s \rangle_1, E + \{\langle x \rangle_1 \rightarrow E(\langle x \rangle), \langle \text{feat} \rangle_1 \ldots \langle x \rangle_n \rightarrow E(\langle x \rangle), \langle \text{feat} \rangle_n\})\mid ST, \sigma)\]

- Otherwise:

  \[\Rightarrow (\langle s \rangle_2, E)\mid ST, \sigma)\]
**Procedure application.** Given the current state

$$(((\{\langle x \rangle \langle y \rangle_1 \ldots \langle y \rangle_n\}, E) \mid ST, \sigma)$$

there are two possible outcomes:

- If $E(\langle x \rangle)$ is not bound to a procedure, or if it is bound to a procedure with a different number of arguments then raise an exception.
- Otherwise:

  $$\mapsto (((\langle s \rangle, CE + \{\langle z \rangle_1 \mapsto E(\langle y \rangle_1) \ldots \langle z \rangle_n \mapsto E(\langle y \rangle_n)\}) \mid ST, \sigma)$$

Where $\langle s \rangle$ is the body of the procedure, $\langle z \rangle_1 \ldots \langle z \rangle_n$ are its formal parameters and $CE$ is its contextual environment.
Translating to the kernel language

local Max C in

proc {Max X Y Z}
    if \(X \geq Y\) then \(Z = X\) else \(Z = Y\) end
end

\{Max 3 5 C\}
end

\[
\begin{align*}
\{s\} & \equiv \\
\{s\}_1 & \equiv \\
\{s\}_2 & \equiv \{\text{Max } A B C\} \\
\end{align*}
\]
An example of a reduction

\[
\begin{align*}
\langle s \rangle & \equiv \\
& \begin{cases}
\text{local Max in} \\
\text{local A in} \\
\text{local B in} \\
\text{local C in} \\
\text{Max.proc \{X Y Z\}} \\
\text{local T in} \\
T=\{X=Y\} \\
\langle s_1 \rangle & \equiv \\
\{s_1, A \equiv \text{if T then Z=X else Z=Y end end} \} \\
\langle s_2 \rangle & \equiv \{\text{Max A B C} \} \\
\end{cases}
\end{align*}
\]

1 \( \{((\langle s \rangle, \phi)), \phi \} \)
2 \( \{((\langle s_1 \rangle, \{\text{Max} \rightarrow m, A \rightarrow a, B \rightarrow b, C \rightarrow c\}\}), \{m, a, b, c\} \} \)
3 \( \{((\{\text{Max A B C}\}, \{\text{Max} \rightarrow m, A \rightarrow a, B \rightarrow b, C \rightarrow c\}\}), \{m = (\text{proc} \{X Y Z\} \langle s_3 \rangle \text{ end, } \phi), a = 3, b = 5, c\} \} \)
4 \( \{((\langle s_3 \rangle, \{X \rightarrow a, Y \rightarrow b, Z \rightarrow c\}\}), \{m = (\text{proc} \{X Y Z\} \langle s_3 \rangle \text{ end, } \phi), a = 3, b = 5, c\} \} \)
5 \( \{((\langle s_4 \rangle, \{X \rightarrow a, Y \rightarrow b, Z \rightarrow c, T \rightarrow t\}\}), \{m = (\text{proc} \{X Y Z\} \langle s_3 \rangle \text{ end, } \phi), a = 3, b = 5, c, t = \text{false}\} \) \)
6 \( \{m = (\text{proc} \{X Y Z\} \langle s_3 \rangle \text{ end, } \phi), a = 3, b = 5, c = 5, t = \text{false}\} \)
Another example of a reduction involving free variables

```
local LowerBound Y C in
  Y=5
  proc {LowerBound X ?Z}
    if X>=Y then Z=X else Z=Y end
  end
  {LowerBound 3 C}
end
```

After some reduction steps....

1. \[
\left[ \left( \left\{ \text{LowerBound } A \ C \right\}, \left\{ Y \to y, \text{LowerBound } \to lb, A \to a, C \to c \right\} \right) \right],
\]
\[
\text{lb} = \left( \text{proc } \{ \text{x } z \} \text{ if } x \geq y \text{ then } z=x \text{ else } z=y \text{ end end, } \{ Y \to y \} \right),
\]
\[
y = 5, a = 3, c \right)
\]

2. \[
\left( \left( \text{if } x \geq y \text{ then } z=x \text{ else } z=y \text{ end, } \{ Y \to y, x \to a, z \to c \} \right) \right],
\]
\[
\text{lb} = \left( \text{proc } \{ \text{x } z \} \text{ if } x \geq y \text{ then } z=x \text{ else } z=y \text{ end end, } \{ Y \to y \} \right),
\]
\[
y = 5, a = 3, c \right)
Avoiding declaration of local variables in the definition of composed values:

```plaintext
person(name:"George" age:25)

local A B in A="George" B=25 X=person(name:A age:B) end
```

The declaration of a local variable and its initialization can be done in one step:

```plaintext
local tree(key:A left:B right:C value:D)=T in <statement> end

local A B C D in T=tree(key:A left:B right:C value:D) <statement> end
```
Expressions

An expression is syntactic sugar for a sequence of operations that returns a value.

\[
\text{expression} ::= \text{variable} \mid \text{int} \mid \text{float} \mid \\
\mid \text{expression} \\text{evalBinOp} \text{expression} \\
\mid \text{expression} \\text{evalBinOp} \\text{expression} \\
\mid \text{expression} \\text{evalBinOp} \text{expression} \\
\mid \ldots
\]

\[
\text{evalBinOp} ::= \text{-}+ \mid \text{-}- \mid \text{*-} \mid \text{/}/ \mid \text{div} \mid \text{mod} \\
\mid \text{-}== \mid \text{-}=\text{=} \mid \text{<} \mid \text{<=} \mid \text{>} \mid \text{>=} \mid \ldots
\]
An if statement can be nested and the condition can be an expression.

\[
\begin{align*}
\langle \text{statement} \rangle & ::= \text{if} \langle \text{expression} \rangle \text{ then } \langle \text{inStatement} \rangle \\
& \quad \{ \text{elseif} \langle \text{expression} \rangle \text{ then } \langle \text{inStatement} \rangle \} \\
& \quad [ \text{else} \langle \text{inStatement} \rangle \] \text{ end} \\
& \quad | \ldots \\
\langle \text{inStatement} \rangle & ::= [ \{ \langle \text{declarationPart} \rangle \} + \text{ in } ] \langle \text{statement} \rangle
\end{align*}
\]
Nested case statements

A case statement can be nested and the value to match can be an expression.

\[
\begin{align*}
\langle \text{statement}\rangle & \ ::= \ \text{case} \ \langle \text{expression}\rangle \\
& \quad \text{of} \ \langle \text{pattern}\rangle \ [ \ \text{andthen} \ \langle \text{expression}\rangle \ ] \ \text{then} \ \langle \text{inStatement}\rangle \\
& \quad \{ \ \text{else} \ \langle \text{inStatement}\rangle \ \} \\
& \quad \ldots
\end{align*}
\]

\[
\begin{align*}
\langle \text{pattern}\rangle & \ ::= \ \langle \text{variable}\rangle \ | \ \langle \text{atom}\rangle \ | \ \langle \text{int}\rangle \ | \ \langle \text{float}\rangle \\
& \quad \langle \text{string}\rangle \ | \ \text{unit} \ | \ \text{true} \ | \ \text{false} \\
& \quad \langle \text{label}\rangle \ ~ \{ \[ \ \langle \text{feature}\rangle \ : \ ~ \] \ \langle \text{pattern}\rangle \ \} \ [ \ ~ \ldots \ ~ ] \ ~ \}
\end{align*}
\]

\[
\begin{align*}
\langle \text{consBinOp}\rangle & \ ::= \ ~ \# ~ \mid ~ \mid ~ \mid
\end{align*}
\]
Using syntactic sugar for case statements

case Xs#Ys
  of nil#Ys then ⟨s⟩_1
  [] Xs=nil then ⟨s⟩_2
  [] (X|Xr)#(Y|Yr) andthen X=<Y then ⟨s⟩_3
  else ⟨s⟩_4 end
end

case Xs of nil then ⟨s⟩_1
else
  case Ys of nil then ⟨s⟩_2
  else
    case Xs of X|Xr then
      case Ys of Y|Yr then
        if X=<Y then ⟨s⟩_3 else ⟨s⟩_4 end
      else ⟨s⟩_4 end
      else ⟨s⟩_4 end
    end
  end
end
A function definition differs from a procedure definition in two ways: it is introduced with keyword `fun` and the body must end with an expression.

```
fun {F X1 ... XN} ⟨statement⟩ ⟨expression⟩ end
```

```
proc {F X1 ... XN ?R} ⟨statement⟩ R=⟨expression⟩ end
```
In a function/procedure call actual parameters are evaluated before the call takes place:

\[
\begin{align*}
Q & \begin{cases}
F & X_1 \ldots X_N \\
\end{cases} \\
& \ldots \\
\end{align*}
\]

\[
\begin{align*}
\text{local} & \quad R \quad \text{in} \\
& \begin{cases}
F & X_1 \ldots X_N \quad R \\
Q & R \quad \ldots \\
\end{cases} \\
\text{end}
\end{align*}
\]
In a variable-value binding, the binding takes place before any function call involved in the definition of the value:

\[
Ys = \{ F \; X \} \mid \{ \text{Map} \; Xr \; F \}
\]

\[
\text{local} \; Y \; Yr \; \text{in}
\]

\[
Ys = Y \mid Yr
\]

\[
\{ F \; X \; Y \}
\]

\[
\{ \text{Map} \; Xr \; F \; Yr \}
\]

\[
\text{end}
\]
List the sequence of states involved in the execution of the following program and elaborate on the reduction rules applied:

```
local ShiftLeft ShiftRight GenericPascal OpList R in
fun {ShiftLeft L} case L of H|T then
    case H|(ShiftLeft T)
    else [0] end end
fun {ShiftRight L} O|L end
fun {GenericPascal Op N}
    if N==1 then [1]
    else L in
        L={(GenericPascal Op N-1)
        (OpList Op {ShiftLeft L} {ShiftRight L})
    end end

fun {OpList Op L1 L2}
    case L1 of H1|T1 then
        case L2 of H2|T2 then
            (Op H1 H2)|(OpList Op T1 T2)
        end
    else nil end
end
(GenericPascal fun($ X Y) X+Y end 3 R)
end
```