Declarative Computation Model

Single assignment store (VRH 2.2)
Kernel language syntax (VRH 2.3)

Carlos Varela
RPI
September 21, 2006
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

Single assignment store

• A single assignment store is a store (set) of variables
• Initially the variables are unbound, i.e. do not have a defined value
• Example: a store with three variables, \( x_1 \), \( x_2 \), and \( x_3 \)

**The Store**

\[
\begin{align*}
\text{x_1} & \quad \text{unbound} \\
\text{x_2} & \quad \text{unbound} \\
\text{x_3} & \quad \text{unbound}
\end{align*}
\]

Single assignment store (2)

• Variables in the store may be bound to values
• Example: assume we allow as values, integers and lists of integers

**The Store**

\[
\begin{align*}
\text{x_1} & \quad \text{unbound} \\
\text{x_2} & \quad \text{unbound} \\
\text{x_3} & \quad \text{unbound}
\end{align*}
\]

Single assignment store (3)

• Variables in the store may be bound to values
• Assume we allow as values, integers and lists of integers
• Example: \( x_1 \) is bound to the integer 314, \( x_2 \) is bound to the list \( [1 \ 2 \ 3] \), and \( x_3 \) is still unbound

**The Store**

\[
\begin{align*}
\text{x_1} & \quad 314 \\
\text{x_2} & \quad [1 \ 2 \ 3 \ \text{nil}] \\
\text{x_3} & \quad \text{unbound}
\end{align*}
\]

Declarative (single-assignment) variables

• A declarative variable starts out as being unbound when created
• It can be bound to exactly one value
• Once bound it stays bound through the computation, and is indistinguishable from its value

**The Store**

\[
\begin{align*}
\text{x_1} & \quad 314 \\
\text{x_2} & \quad [1 \ 2 \ 3 \ \text{nil}] \\
\text{x_3} & \quad \text{unbound}
\end{align*}
\]
Value store

- A store where all variables are bound to values is called a value store.
- Example: a value store where \(x_1\) is bound to integer 314, \(x_2\) to the list [1 2 3], and \(x_3\) to the record (labeled tree) `person(name:"George", age:25)`.
- Functional programming computes functions on values, needs only a value store.
- This notion of value store is enough for functional programming (ML, Haskell, Scheme).

Operations on the store (1)

Single assignment

\[\langle x \rangle = \langle v \rangle\]
- \(x_1 = 314\)
- \(x_2 = [1 2 3]\)
- This assumes that \(\langle x \rangle\) is unbound.

Single assignment (2)

\[\langle x \rangle = \langle v \rangle\]
- \(x_1 = 314\)
- \(x_2 = [1 2 3]\)
- The single assignment operation (\(=\)) constructs the \(\langle v \rangle\) in the store and binds the variable \(\langle x \rangle\) to this value.
- If the variable is already bound, the operation will test the compatibility of the two values.
- If the test fails an error is raised.

Variable identifiers

- Variable identifiers refer to store entities (variables or values).
- The environment maps variable identifiers to variables.
  - `declare X`: \(X\)
  - `local X in ...`: \(X\) is a variable identifier.
  - This corresponds to “environment” \(\{X \rightarrow x_1\}\).

Variable-value binding revisited (1)

- \(X = [1 2 3]\)
- Once bound the variable is indistinguishable from its value.
Variable-value binding revisited (2)

- $X = [1 \ 2 \ 3]$
- Once bound the variable is indistinguishable from its value
- The operation of traversing variable cells to get the value is known as dereferencing and is invisible to the programmer

\[
\begin{array}{c}
\text{The Store} \\
\end{array}
\]


text:

\[
\begin{array}{c}
X = [1 \ 2 \ 3] \\
\end{array}
\]

• The store contains the partial value: \texttt{person(name: ”George” age: x)}
• The identifier ‘Y’ refers to x

Partial Values

- A partial value is a data structure that may contain unbound variables
- Partial Values may be complete
  - declare \texttt{Y X}
  - \texttt{X = person(name: ”George” age: Y)}

\[
\begin{array}{c}
\text{The Store} \\
\end{array}
\]

• \texttt{Y = 25}
• All variables (X and Y) are bound to \([1 \ 2 \ 3]\)

Variable to variable binding

- It is to perform a single assignment between variables
- Example:
  - X = Y
  - X = [1 2 3]
- The operations equates the two variables (forming an equivalence class)

\[
\begin{array}{c}
\text{The Store} \\
\end{array}
\]

\[
\begin{array}{c}
X \rightarrow x_1 \\
Y \rightarrow x_2 \\
\end{array}
\]
Summary
Variables and partial values

- **Declarative variable:**
  - is an entity that resides in a single-assignment store, that is initially unbound, and can be bound to exactly one (partial) value
  - it can be bound to several (partial) values as long as they are compatible with each other

- **Partial value:**
  - is a data-structure that may contain unbound variables
  - when one of the variables is bound, it is replaced by the (partial) value it is bound to
  - a complete value, or value for short is a data structure that does not contain any unbound variables

Declaration and use of variables (2)

- An attempt to use the variable will wait (suspends) until another activity binds the variable (Oz/Mozart)
- Declarative (single assignment) variables that have this property are called **dataflow** variables
- It allows multiple operations to proceed concurrently giving the correct result
- Example: \( A = 23 \) running concurrently with \( B = A+1 \)
- Functional (concurrent) languages do not allow the separation between declaration and binding (ML, Haskell, and Erlang)

Kernel language syntax

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

\[
\begin{align*}
    (s) & \ ::= \ \text{skip} \quad \text{empty statement} \\
    & \ | \ (x) \ = \ (y) \quad \text{variable-variable binding} \\
    & \ | \ (x) \ = \ v \quad \text{variable-value binding} \\
    & \ | \ (s_1) \ (s_2) \quad \text{sequential composition} \\
    & \ | \ \text{local} (x) \ in \ (s_1) \ \text{end} \quad \text{declaration} \\
    & \ | \ \text{if} (x) \ \text{then} \ (s_1) \ \text{else} \ (s_2) \ \text{end} \quad \text{conditional} \\
    & \ | \ \text{case} (x) \ \text{of} \ (\text{pattern}) \ \text{then} \ (s_1) \ \text{else} \ (s_2) \ \text{end} \quad \text{pattern matching} \\
    (v) & \ ::= \ ... \quad \text{value expression} \\
    (\text{pattern}) & \ ::= \ ...
\end{align*}
\]

Variable identifiers

- \( (x), (y), (z) \) stand for variables
- In the concrete kernel language variables begin with upper-case letter followed by a (possibly empty) sequence of alphanumeric characters or underscore
- Any sequence of printable characters within back-quote
- Examples:
  - X
  - Y1
  - Hello_World
  - 'hello this is a $5 bill' (back-quote)

Values and types

- A **data type** is a set of values and a set of associated operations
- Example: Int is the the data type "Integer", i.e set of all integer values
- 1 is **of type** Int
- Int has a set of operations including +, -, *, div, etc
- The model comes with a set of basic types
- Programs can define other types, e.g., **abstract data types** ADT
**Data types**

- **Value**
  - Number
  - Record
  - Procedure

- **Number**
  - Int
  - Float

- **Literal**
  - Char

- **Record**

- **Procedure**
  - Tuple
  - List
  - Atom
  - Boolean
  - True
  - False

**Data types (2)**

**Value expressions**

\[
\begin{align*}
(v) & \;::=\; (\text{procedure}) \mid (\text{record}) \mid (\text{number}) \\
(\text{procedure}) & \;::=\; \text{proc} \left[ (y_1) \ldots (y_n) \right] (v) \;\text{end} \\
(\text{record}), (\text{pattern}) & \;::=\; (\text{literal}) \\
& \qquad \mid (\text{literal}) \left[ \left( \text{feature}_1: (x_1) \ldots \left( \text{feature}_n: (x_n) \right) \right) \right] \\
(\text{literal}) & \;::=\; (\text{atom}) \mid (\text{bool}) \\
(\text{feature}) & \;::=\; (\text{int}) \mid (\text{atom}) \mid (\text{bool}) \\
(\text{bool}) & \;::=\; \text{true} \mid \text{false} \\
(\text{number}) & \;::=\; (\text{int}) \mid (\text{float})
\end{align*}
\]

**Numbers**

- **Integers**
  - 314, 0
  - -10 (minus 10)
- **Floats**
  - 1.0, 3.4, 2.0e2, 2.0E2 \((2 \times 10^2)\)

**Atoms and booleans**

- A sequence starting with a lower-case character followed by characters or digits, …
  - person
  - peter
  - ‘Seif Haridi’
- **Booleans**:
  - true
  - false

**Records**

- Compound representation (data-structures)
  - \(\{ (\text{feature}_1: (x_1) \ldots (\text{feature}_n: (x_n)) \} \)
  - \(\{\}\) is a literal
- **Examples**
  - person\(\{\text{age}: X1 \text{ name}: X2\}\)
  - person\(\{X1 2X2\}\)
  - \(\{1; H T\}\)
  - nil
  - person
Syntactic sugar (tuples)

- Tuples
  \[ \langle l \rangle \langle x_1 \rangle \ldots \langle x_n \rangle \] (tuple)
- This is equivalent to the record
  \[ \langle l \rangle (1: \langle x_1 \rangle \ldots n: \langle x_n \rangle) \]
- Example:
  \[ \text{person('George' 25)} \]
  This is the record
  \[ \text{person(1: 'George' 2: 25)} \]

Syntactic sugar (lists)

- Lists
  \[ \langle x_1 \rangle | \langle x_2 \rangle \] (a cons with the infix operator ‘|’)
- This is equivalent to the tuple
  \[ \langle \langle x_1 \rangle \langle x_2 \rangle \rangle \]
- Example:
  \[ H | T \]
  This is the tuple
  \[ \langle H T \rangle \]

Strings

- A string is a list of character codes enclosed with double quotes
- Ex. "E=mc^2"
- Means the same as [69 61 109 99 94 50]

Procedure declarations

- According to the kernel language
  \[ (x) = \text{proc } \{ \langle y_1 \rangle \ldots \langle y_n \rangle \} (x) \text{ end} \]
  is a legal statement
- It binds \( x \) to a procedure value
- This statement actually declares (introduces) a procedure
- Another syntactic variant which is more familiar is
  \[ \text{proc } \{ \langle y_1 \rangle \ldots \langle y_n \rangle \} (x) \text{ end} \]
  This introduces (declares) the procedure \( x \)
Operations of basic types

- **Arithmetics**
  - Floating point numbers: +, -, *, /, mod (the remainder after a division, e.g. 10 mod 3 = 1)
  - Integers: +, -, *, div (integer division, i.e. truncate fractional part), mod (remainder after division, e.g. 10 mod 3 = 1)

- **Record operations**
  - Arity, Label, and ".
  - X = person(name: "George", age: 25)
  - X.age = 25

- **Comparisons**
  - Boolean comparisons: ==, !=
  - Numeric comparisons: =<, <, >, >=

Value expressions

(v) ::= (procedure) | (record) | (number) | (basicExpr) | ...
(basicExpr) ::= (valueExpr) | ...
(numberExpr) ::= (v1) + (v2) | ...

Syntactic sugar (multiple variables)

- **Multiple variable introduction**
  
  ```
  local X Y in (statement) end
  ```

  - is transformed to
  
  ```
  local X in
  local Y in (statement) end
  ```

Syntactic sugar (basic expressions)

- **Basic expression nesting**
  
  ```
  if (basicExpr) then (statement) else (statement) end
  ```

  - is transformed to
  
  ```
  local T in
  T = (basicExpr)
  if T then (statement) else (statement) end
  ```

  - where T is a fresh ("new") variable identifier

Syntactic sugar (variables)

- **Variable initialization**
  
  ```
  local X = (value) in (statement) end
  ```

  - is transformed to
  
  ```
  local X = (value) in (statement) end
  ```

Exercises

23. Using Oz, perform a few basic operations on numbers, records, and booleans (see Appendix B1-B3)
24. Explain the behavior of the `declare` statement in the interactive environment. Give an example of an interactive Oz session where "declare" and "declare ... in" produce different results. Explain why.
25. VRH Exercise 2.9.1
26. Describe what an anonymous procedure is, and write one in Oz. When are anonymous procedures useful?