**Declarative Computation Model**

Single assignment store (VRH 2.2)

Kernel language syntax (VRH 2.3)

Carlos Varela
RPI
October 4, 2007

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

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

---

**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

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

---

**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

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

---

**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

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

\[ \{x\} = \{v\} \]
• \( x_1 = 314 \)
• \( x_2 = [1, 2, 3] \)
• This assumes that \( \{x\} \) is unbound.

Single-assignment (2)

\[ \{x\} = \{v\} \]
• \( x_1 = 314 \)
• \( x_2 = [1, 2, 3] \)
• The single assignment operation ("\(=\)"") constructs the \( \{v\} \) in the store and binds the variable \( \{x\} \) 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`;
  • `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

Partial Values

- A partial value is a data structure that may contain unbound variables
- The store contains the partial value: `person(name: 'George' age: x)`
- The identifier 'Y' refers to $x_2$

Partial Values (2)

Partial Values may be complete

- `declare Y X
  X = person(name: 'George' age: Y)
  Y = 25`

Variable to variable binding

- $(x_1) \equiv (x_2)$
- It is to perform a single assignment between variables
- Example:
  - $X = Y$
  - $X = [1 \ 2 \ 3]$
  - The operation equates the two variables (forming an equivalence class)

Variable to variable binding (2)

- $(x_1) \equiv (x_2)$
  - It is to perform a single assignment between variables
  - Example:
    - $X = Y$
    - $X = [1 \ 2 \ 3]$
  - All variables ($X$ and $Y$) are bound to $[1 \ 2 \ 3]$
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**

- Assume that variables can be declared (introduced) and used separately
- What happens if we try to use a variable before it is bound?
  1. Use whatever value happens to be in the memory cell occupied by the variable (C, C++)
  2. The variable is initialized to a default value (Java), use the default
  3. An error is signaled (Prolog). Makes sense if there is a single activity running (pure sequential programs)
  4. An attempt to use the variable will wait (suspends) until another activity binds the variable (Oz/Mozart)

**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

 Syntax: (s) ::= ...

- **Value expression**
  - (v) ::= ...

- **Pattern matching**
  - case (x) of (pattern) then (s1) else (s2) end

- **Procedural application**
  - ' { x1 y1 ... yn } ' (back-quote)

- **Declarative composition**
  - local (x) in (s1) end

- **Conditional**
  - if (x) then (s1) else (s2) end

- **Variable-variable binding**
  - (x) = (y)

- **Sequential composition**
  - (s1) (s2)

- **Empty statement**
  - skip

- **Declaration**
  - (x) in (s) end

**Variable identifiers**

- (x), (y), (z) stand for variables
- In the concrete kernel language variables begin with uppercase 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**
    - Int, Float, Literal
  - **Record**
    - Tuple, Literal
  - **Procedure**
    - List, Atom, Boolean

- **Data types (2)**
  - **Value**
    - **Number**
      - Int, Float, Literal
    - **Record**
      - Tuple, Literal
    - **Procedure**
      - List, Atom, Boolean

Value expressions

\[(v) ::= (\text{procedure}) \mid (\text{record}) \mid (\text{number})\]

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

Numbers

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

Atoms and booleans

- **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)**
  - \(\{0: x_1 \ldots x_n\}\)
  - \(\{\}\) is a literal
- **Examples**
  - person(age:X1 name:X2)
  - person(X1 X2)
  - \[['1', 'H':'T']\]
  - 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
  \( \langle l: \langle x_1 \rangle \ldots n: \langle x_n \rangle \rangle \)

- Example:
  \( \text{person('George' 25)} \)
  \( \text{This is the record} \)
  \( \text{person('George' 2:25)} \)

Syntactic sugar (lists)

- Lists
  \( \langle x_1 \rangle \langle x_2 \rangle \) (a cons with the infix operator ‘\( \mid \)’)

- This is equivalent to the tuple
  \( \langle \langle x_1 \rangle \langle x_2 \rangle \rangle \)

- Example:
  \( H \mid T \)
  \( \text{This is the tuple} \)
  \( \langle H T \rangle \)

Syntactic sugar (lists)

- Lists
  \( \langle x_1 \rangle \langle x_2 \rangle \langle x_3 \rangle \)
  \( \text{‘\( \mid \)’ associates to the right} \)
  \( \langle x_1 \rangle \langle x_2 \rangle \langle x_3 \rangle \langle \text{nil} \rangle \)

- Example:
  \( \text{1 \mid 2 \mid 3 \mid \text{nil}} \)

  \( \text{Is} \)
  \( 1 \mid (2 \mid (3 \mid \text{nil})) \)

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
  \( \langle x \rangle \equiv \text{proc} \{ \langle y_1 \rangle \ldots \langle y_n \rangle \} (x) \) \text{ end} \)
  is a legal statement

- It binds \( \langle x \rangle \) to a procedure value

- This statement actually declares (introduces) a procedure

- Another syntactic variant which is more familiar is
  \( \text{proc} \{ \langle x \rangle \langle y_1 \rangle \ldots \langle y_n \rangle \} (x) \) \text{ end} \)

- This introduces (declares) the procedure \( \langle x \rangle \)
Operations of basic types

• Arithmetics
  – Floating point numbers: +, -, *, and /.
  – Integers: +, -, *, div (integer division, i.e. truncate fractional part),
    mod (the remainder after a division, e.g. 10 mod 3 = 1)
• Record operations
  – Arity, Label, and "."
  – X = person(name: "George" age: 25)
  – {Arity X} = [age name]
  – {Label X} = person, X.age = 25
• Comparisons
  – Boolean comparisons, including ==, = (equality)
  – Numeric comparisons, =<, <, >, >=, compares integers, floats, and
    atoms

Value expressions

\( (v) ::= \langle \text{procedure} \rangle | \langle \text{record} \rangle | \langle \text{number} \rangle | \langle \text{basicExpr} \rangle \)

\( \langle \text{basicExpr} \rangle ::= \ldots | \langle \text{numberExpr} \rangle \ldots \)

\( \langle \text{numberExpr} \rangle ::= (x_1) + (x_2) \ldots \)

Syntactic sugar (multiple variables)

• Multiple variable introduction
  \[ \text{local } X \text{ Y in (statement) end} \]
  
  is transformed to
  \[ \text{local } X \text{ in} \\
  \text{local } Y \text{ in (statement) end} \]

Syntactic sugar (basic expressions)

• Basic expression nesting
  \[ \text{if (basicExpr) then (statement), else (statement)_2 end} \]

  is transformed to
  \[ \text{local } T \text{ in} \\
  \text{local } X \text{ in} \\
  \text{local } Y \text{ in (statement) end} \]

  where \( T \) is a fresh (‘new’) variable identifier

Syntactic sugar (variables)

• Variable initialization
  \[ \text{local } X = \langle \text{value} \rangle \text{ in (statement) end} \]

  Is transformed to
  \[ \text{local } X \text{ in} \\
  X = \langle \text{value} \rangle \\
  \langle \text{statement} \rangle \]

Exercises

42. Using Oz, perform a few basic operations on numbers, records, and booleans (see Appendix B1-B3)
43. Explain the behavior of the declare statement in the interactive environment. Give an example of an interactive Oz session where “declare” and “declare ...
44. VRH Exercise 2.9.1
45. *Describe what an anonymous procedure is, and write one in Oz. When are anonymous procedures useful?