Declarative Programming Techniques
Declarativeness, iterative computation (VRH 3.1-3.2)

Carlos Varela
RPI
October 10, 2006
Adapted with permission from:
Seif Haridi
KTH
Peter Van Roy
UCL

Overview
• What is declarativeness?
  – Classification,
  – Advantages for large and small programs
• Control Abstractions
  – Iterative programs

Declarative operations (1)
• An operation is declarative if whenever it is called with the same arguments, it returns the same results independent of any other computation state
• A declarative operation is:
  – Independent (depends only on its arguments, nothing else)
  – Stateless (no internal state is remembered between calls)
  – Deterministic (call with same operations always give same results)
• Declarative operations can be composed together to yield other declarative components
  – All basic operations of the declarative model are declarative and combining them always gives declarative components

Declarative operations (2)

Why declarative components (1)
• There are two reasons why they are important:
  • (Programming in the large) A declarative component can be written, tested, and proved correct independent of other components and of its own past history.
    – The complexity (reasoning complexity) of a program composed of declarative components is the sum of the complexity of the components
    – In general the reasoning complexity of programs that are composed of nondeclarative components explodes because of the intimate interaction between components
  • (Programming in the small) Programs written in the declarative model are much easier to reason about than programs written in more expressive models (e.g., an object-oriented model).
    – Simple algebraic and logical reasoning techniques can be used

Why declarative components (2)
• Since declarative components are mathematical functions, algebraic reasoning is possible i.e. substituting equals for equals
  • The declarative model of chapter 2 guarantees that all programs written are declarative
  • Declarative components can be written in models that allow stateful data types, but there is no guarantee
    Given $f(a) = a^2$
    We can replace $f(a)$ in any other equation
    $b = 7f(a)^2$ becomes $b = 7a^4$
Classification of declarative programming

- The word *declarative* means many things to many people. Let’s try to eliminate the confusion.
- The basic intuition is to program by defining the *what* without explaining the *how*.

Descriptive language

\[
\begin{align*}
\langle \text{skip} \rangle & \quad \text{empty statement} \\
\langle x \rangle = \langle y \rangle & \quad \text{variable-variable binding} \\
\langle x \rangle = \langle \text{record} \rangle & \quad \text{variable-value binding} \\
\langle \text{local} \rangle \langle x \rangle \langle \text{in} \rangle \langle s_1 \rangle & \quad \text{sequential composition} \\
\langle \langle s_1 \rangle \rangle \langle s_2 \rangle & \quad \text{declaration} \\
\langle \text{proc} \rangle \langle x \rangle \langle \{ y_1, \ldots, y_n \} \rangle \langle s_1 \rangle & \quad \text{procedure introduction} \\
\langle \text{if} \rangle \langle x \rangle \langle \text{then} \rangle \langle s_1 \rangle \langle \text{else} \rangle \langle s_2 \rangle & \quad \text{conditional} \\
\langle \text{case} \rangle \langle x \rangle \langle \text{of} \rangle \langle \text{pattern} \rangle \langle \text{then} \rangle \langle s_1 \rangle \langle \text{else} \rangle \langle s_2 \rangle & \quad \text{pattern matching}
\end{align*}
\]

Other descriptive languages include HTML and XML.

Kernel language

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

\[
\begin{align*}
\langle \text{skip} \rangle & \quad \text{empty statement} \\
\langle x \rangle = \langle y \rangle & \quad \text{variable-variable binding} \\
\langle x \rangle = \langle v \rangle & \quad \text{variable-value binding} \\
\langle \text{local} \rangle \langle x \rangle \langle \text{in} \rangle \langle s_1 \rangle & \quad \text{sequential composition} \\
\langle \langle s_1 \rangle \rangle \langle s_2 \rangle & \quad \text{declaration} \\
\langle \text{proc} \rangle \langle x \rangle \langle \{ y_1, \ldots, y_n \} \rangle \langle s_1 \rangle & \quad \text{procedure introduction} \\
\langle \text{if} \rangle \langle x \rangle \langle \text{then} \rangle \langle s_1 \rangle \langle \text{else} \rangle \langle s_2 \rangle & \quad \text{conditional} \\
\langle \text{case} \rangle \langle x \rangle \langle \text{of} \rangle \langle \text{pattern} \rangle \langle \text{then} \rangle \langle s_1 \rangle \langle \text{else} \rangle \langle s_2 \rangle & \quad \text{pattern matching}
\end{align*}
\]

Why the KL is declarative

- All basic operations are declarative
- Given the components (sub-statements) are declarative,
  - sequential composition
  - local statement
  - procedure definition
  - procedure call
  - if statement
  - case statement
  are all declarative (independent, stateless, deterministic).

Iterative computation

- An iterative computation is a one whose execution stack is bounded by a constant, independent of the length of the computation
- Iterative computation starts with an initial state \( S_0 \), and transforms the state in a number of steps until a final state \( S_{\text{final}} \) is reached:

\[
S_0 \rightarrow S_1 \rightarrow \ldots \rightarrow S_{\text{final}}
\]
The general scheme

fun {Iterate $S_i$}
if {IsDone $S_i$} then $S_i$
else $S_{i+1} = \{\text{Transform } S_i\}$
{Iterate $S_{i+1}$}
end
end

- IsDone and Transform are problem dependent

The computation model

- STACK : $[R = \{\text{Iterate } S_i\}]$
- STACK : $[S_i = \{\text{Transform } S_i\}]$
  \label{r}\
  R = \{\text{Iterate } S_i\}$
end
- STACK : $[R = \{\text{Iterate } S_{i+1}\}]$
- STACK : $[S_{i+1} = \{\text{Transform } S_i\}]$
  \label{r}\
  R = \{\text{Iterate } S_{i+1}\}$
end

Newton’s method for the square root of a positive real number

- Given a real number $x$, start with a guess $g$, and improve this guess iteratively until it is accurate enough
- The improved guess $g'$ is the average of $g$ and $x/g$:
  \[ g' = \frac{g + x/g}{2} \]
  \[ \varepsilon = g - \sqrt{x} \]
  \[ \varepsilon' = g' - \sqrt{x} \]
  For $g'$ to be a better guess than $g$: $\varepsilon' < \varepsilon$
  \[ \varepsilon = g' - \sqrt{x} = \frac{(g + x/g)}{2} - \sqrt{x} = \frac{\varepsilon^2}{2g} \]
  i.e. $\varepsilon^2 / 2g < \varepsilon$, $\varepsilon / 2g < 1$
  i.e. $\varepsilon < 2g$, $g = \sqrt{x} < 2g$, $0 < g + \sqrt{x}$

The program version 1

fun (SqrtIter Guess X)
if (GoodEnough Guess X) then Guess
else
  Guess1 = (Improve Guess X)
in (SqrtIter Guess1 X)
end
end

- Compare to the general scheme:
  - The state is the pair Guess and X
  - IsDone is implemented by the procedure GoodEnough
  - Transform is implemented by the procedure Improve

SqrtIter

fun (SqrtIter Guess X)
if (GoodEnough Guess X) then Guess
else
  Guess1 = (Improve Guess X)
in (SqrtIter Guess1 X)
end
end

Newton’s method for the square root of a positive real number

- Given a real number $x$, start with a guess $g$, and improve this guess iteratively until it is accurate enough
- The improved guess $g'$ is the average of $g$ and $x/g$:
- Accurate enough is defined as:
  \[ |x - g^2| / x < 0.00001 \]
Using local procedures

- The main procedure Sqrt uses the helper procedures SqrtIter, GoodEnough, Improve, and Abs
- SqrtIter is only needed inside Sqrt
- GoodEnough and Improve are only needed inside SqrtIter
- Abs (absolute value) is a general utility
- The general idea is that helper procedures should not be visible globally, but only locally

Sqrt version 2

```plaintext
local
fun {Sqrt X}
    fun {SqrtIter Guess X}
        if {GoodEnough Guess}
            Guess
        else
            {SqrtIter {Improve Guess} X}
        end
    end
    end
    Guess = 1.0
    in
    {SqrtIter Guess X}
end
end
```

Sqrt version 3

- Define GoodEnough and Improve inside SqrtIter

```plaintext
local
fun {SqrtIter Guess X}
    fun {Improve}
        (Guess + X/Guess)/2.0
    end
    fun {GoodEnough}
        {Abs X - Guess*Guess}/X < 0.000001
    end
    if {GoodEnough} then
        Guess
    else
        {SqrtIter {Improve} Guess X}
    end
end
end
fun {Sqrt X}
    Guess = 1.0
    in
    {SqrtIter Guess X}
end
end
```

Sqrt final version

```plaintext
fun {Sqrt X}
    fun {Improve Guess}
        (Guess + X/Guess)/2.0
    end
    fun {GoodEnough Guess}
        {Abs X - Guess*Guess}/X < 0.000001
    end
    fun {SqrtIter Guess}
        if {GoodEnough} then
            Guess = 1.0
        else
            {SqrtIter {Improve} Guess}
        end
    end
    end
end
```

From a general scheme to a control abstraction (1)

```plaintext
fun {Iterate S}
    if {IsDone S} then
        S
    else
        S = {Transform S}
        {Iterate S}
    end
end
```

The final version is a compromise between abstraction and efficiency

- IsDone and Transform are problem dependent
From a general scheme to a control abstraction (2)

fun {Iterate S IsDone Transform}
if {IsDone S} then S
else S1 in
S1 = {Transform S}
{Iterate S1 IsDone Transform}
end
end

fun {Iterate S i}
if {IsDone S i} then S i
else S i + 1 in
S i + 1 = {Transform S i}
{Iterate S i + 1}
end
end

Sqrt using the Iterate abstraction

fun {Sqrt X}
fun {Iterate S i}
end
end

fun {Improve Guess}
(Guess + X/Guess)/2.0
end

fun {GoodEnough Guess}
(Abs X - Guess*Guess)/X < 0.000001
end

Guess = 1.0
in
{Iterate Guess GoodEnough Improve}
end

Sqrt using the control abstraction

fun {Sqrt X}
{Iterate 1.0
end
fun {G} (Abs X - G*G)/X < 0.000001 end
fun {G} (G + X/G)/2.0 end
}
end

Iterate could become a linguistic abstraction

Exercises

43. Modify the Pascal function to use local functions for AddList, ShiftLeft, ShiftRight. Think about the abstraction and efficiency tradeoffs.
44. VRH Exercise 3.10.2 (page 230)
45. *VRH Exercise 3.10.3 (page 230)
46. *Develop a control abstraction for iterating over a list of elements.