SE 2A04 Fall 1999

Program Construction and the Program Design Language

(Based on a presentation by D. L. Parnas)

Instructor: W. M. Farmer

Revised: October 25, 1999

Components of a Powerful Language

1. Primitive expressions

2. Means of combination
   - Compound expressions are built from simpler ones via constructors
   - Expressions denote the combination of objects

3. Means of abstraction
   - Compound expressions are built from simpler ones via constructors
   - Expressions denote new objects

   Taken from Abelson, Sussman, and Sussman, *Structure and Interpretation of Computer Programs*

Example: Lambda Notation

- Lambda notation (or something equivalent) is used in many languages to express ideas about functions
- Objects: functions and individuals
- Primitive expressions: names for primitive functions and individuals
  - For example, $0 : \mathbf{N}$, $\mathrm{suc} : \mathbf{N} \to \mathbf{N}$, $+ : \mathbf{N} \times \mathbf{N} \to \mathbf{N}$
- Means of combination: function application constructor
  - $\mathrm{suc}(0)$, $+(\mathrm{suc}(0), \mathrm{suc}(0))$
- Means of abstraction: function abstraction constructor
  - $\lambda\, x : \mathbf{N} \,.\, \mathrm{suc}(\mathrm{suc}(0))$, $\lambda\, x : \mathbf{N} \,.\, +(x, x)$

Constructing Programs

- At the machine level, programming is "telling the computer what to do"

- At the human level, programming is constructing bigger programs from smaller ones
  - Previous written programs are building blocks
  - Constructors are used to assemble new programs from the building blocks
  - The new programs become building blocks for someone else

- If we are given a mathematical description of the building blocks, we must be able to produce a mathematical description of the new programs

Program Design Language (PDL)

- Pseudocode language for describing the construction of new programs from primitive programs
- Primitive programs include variable assignment, `abort`, and `skip`
- PDL has four constructors:
  - Sequential composition
  - Conditional execution
  - Selection (of conditional programs)
  - Iteration
- PDL programs are well-structured, easy to understand, and easy to implement in an imperative programming language **down to the primitive programs**

Programs and States

- Programs are intended to start in certain states and, if they terminate, end in certain states
- Programs usually exhibit different behavior with different start states
  - Start states may lead to behavior considered incorrect
- What are the "good" start states for a program?
  - Program runs correctly and eventually terminates (safe state)
  - Program runs correctly but does not terminate (semi-safe state)
- What are the possible end states for a program?

Sequential Composition

- **Construction rule**: If A and B are programs, $A; B$ is a program.
  - Intuitively, $A; B$ means do A and then do B
  - State transition: $s_1 \xrightarrow{A} s_2 \xrightarrow{B} s_3$
  - Note: A and B need not be primitive programs

- This constructor is found in almost every programming language

- Simple examples:
  - Assignment with two variables
  - Swapping two values

Partial Syntax for PDL (1)

Expressed in Backus-Naur Formalism (BNF):

```
<program> ::= <simple program> | <composed program>

<simple program> ::= <primitive program>
                   | (<program>)
                   | ...

<composed program> ::=
   <simple program> ‘;’ <simple program>
   | <composed program> ‘;’ <simple program>

<primitive program> ::= <assignment>
                      | ‘abort’
                      | ‘skip’
                      | ...
```

Are More Constructors Needed?

- With a "rich" set of primitive programs, many programs can be constructed using just ";" alone
  - APL is a programming language in which programs are constructed from powerful primitives using sequential composition

- We need additional constructors to:
  - Limit the conditions under which a program will be executed (conditionals)
  - Select which of several programs to execute (branches)
  - Iterate programs (loops)

Guarded Programs

- A **guard** is a boolean expression
  - Boolean expressions evaluate to **true** or **false**
- **Construction rule**: If g is a guard, and P is a program, then $g \to P$ is a **guarded program**.
  - Intuitively, $g \to P$ means P should be executed only if g evaluates to true
  - If g evaluates to true, $g \to P$ is equivalent to $g; P$
  - If g evaluates to false, $g \to P$ is equivalent to g
- "Guarded programs" are considered to be distinct from "programs"

Proper Use of Guards

- Guards should not be executed in "trap" states:
  - States in which the guard g does not terminate
  - States in which the guard g terminates in a state in which the program P does not terminate

- Best: Guards should not cause side-effects

Selection of Guarded Programs

- **Construction rule**: If $A_1, A_2, \ldots, A_n$ are guarded programs, then $(A_1|A_2|\cdots|A_n)$ is a program.

- Intuitively, $(A_1|A_2|\cdots|A_n)$ means:
  - Select one of $A_1, A_2, \ldots, A_n$ whose guard is true and execute its program
  - But if all the guards are false, `abort` is executed

- Note: If more than one guard is true, the behavior of $(A_1|A_2|\cdots|A_n)$ is nondeterministic

Partial Syntax for PDL (2)

```
<program> ::= <simple program> | <composed program>

<simple program> ::= <primitive program>
                   | (<program>)
                   | (<guarded program list>)
                   | ...

<composed program> ::= ...

<guard> ::= <boolean expression>

<guarded program> ::= <guard> '->' <simple program>

<guarded program list> ::=
   <guarded program>
   | <guarded program list> '|' <guarded program>

<primitive program> ::= ...
```

Divide and Conquer Programming

- **Divide and conquer** is one of the principal ways of mastering complexity in programs
  - Never try to understand or write a whole program at once

- Guarded program lists facilitate divide and conquer

- To check a guarded program list do:
  1. Make sure that at least one guard is true in every state
  2. Make sure each guarded program will behave correctly when the guard is true

Iteration

- To iterate a program means to repeat the program one or more times
  - Part of the repeated program determines whether the program should be executed again

- **Construction rule**: If P is a program, then

  `it` P `ti`

  is a program.
  - `it` P `ti` is called a **loop**
  - P is called the **body** of the loop

- `it` P `ti` means P will be executed and then, by a decision made within P, either P will be executed again or execution will stop

Go and Stop Primitives

- Two new primitives are introduced to determine whether iteration should continue or stop: `go`, `stop`
  - If `go` is executed in `it` P `ti`, the iteration of P continues
  - If `stop` is executed in `it` P `ti`, iteration stops
  - If both are executed, only the last execution counts
  - If neither are executed, iteration stops and `abort` is executed

- Note: The execution of `stop` in P does not cause the execution of P to stop

- Normally, P should execute either `go` or `stop` during each iteration

Other Iteration Constructs

- `while` B `do` P: $\quad$ `it` $(B \to (P; \texttt{go}) \mid \neg B \to \texttt{stop})$ `ti`
- `until` B `do` P: $\quad$ `it` $(\neg B \to (P; \texttt{go}) \mid B \to \texttt{stop})$ `ti`
- `repeat` P `while` B: $\quad$ `it` $P; (B \to \texttt{go} \mid \neg B \to \texttt{stop})$ `ti`
- `repeat` P `until` B: $\quad$ `it` $P; (\neg B \to \texttt{go} \mid B \to \texttt{stop})$ `ti`
- `for` $I \Leftarrow A$ `step` S `until` C `do` P:

  $I \Leftarrow A;$
  `it` $I \leq C \to (P; I \Leftarrow I + S; \texttt{go}) \mid I > C \to \texttt{stop})$ `ti`

Full Syntax for PDL

```
<program> ::= <simple program> | <composed program>

<simple program> ::= <primitive program>
                   | (<program>)
                   | (<guarded program list>)
                   | it <program> ti

<composed program> ::=
   <simple program> ‘;’ <simple program>
   | <composed program> ‘;’ <simple program>

<guard> ::= <boolean expression>

<guarded program> ::= <guard> ‘->’ <simple program>
```

Full Syntax for PDL (cont.)

```
<guarded program list> ::=
   <guarded program>
   | <guarded program list> '|' <guarded program>

<primitive program> ::= <assignment>
                        | 'abort'
                        | 'skip'
                        | 'go'
                        | 'stop'
                        | ...
```

Termination Example

- Problem: $(\mathsf{Even}(`x) \Rightarrow x' = 0) \wedge (\mathsf{Odd}(`x) \Rightarrow x' = 1)$
- Solution:

$x \Leftarrow \mathtt{abs}(x);$
$\mathtt{it}$
$\quad((x = 0 \vee x = 1 \rightarrow \mathtt{stop}) \mid$
$\quad\ (x > 1 \rightarrow (x \Leftarrow x - 2; \mathtt{go})))$
$\mathtt{ti}$

- Does the loop always terminate?
- Does the solution work for all values of x?

Checking Termination

- How to check whether a loop will terminate:
  - Find a quantity that decreases whenever `go` is executed in the loop's body
  - Show that the quantity is always greater than or equal to some minimum value
- Many useful programs do not always terminate
- It is essential to know when a program will terminate

Euclid's GCD Algorithm: Problem

- The GCD of two positive integers is the **greatest common divisor** of the two integers
- Problem: $(`x > 0) \wedge (`y > 0) \wedge (x' = y' = \mathrm{GCD}(`x, `y))$
- Some mathematical facts:
  - If $x > 0$, $y > 0$, and $x > y$, then $\mathrm{GCD}(x - y, y) = \mathrm{GCD}(x, y)$
  - If $x > 0$, then $\mathrm{GCD}(x, x) = x$

Euclid's GCD Algorithm: Solution

- Solution:

$$
\begin{array}{l}
((x > 0 \wedge y > 0 \rightarrow \\
\quad \texttt{it} \\
\quad\quad ((x > y \rightarrow (x \Leftarrow x - y; \texttt{go})) \mid \\
\quad\quad\ (y > x \rightarrow (y \Leftarrow y - x; \texttt{go})) \mid \\
\quad\quad\ (x = y \rightarrow \texttt{stop})) \\
\quad \texttt{ti}) \mid \\
\ (x \leq 0 \vee y \leq 0 \rightarrow \texttt{abort}))
\end{array}
$$

- The loop terminates because:
  - Max(x, y) decreases whenever `go` is executed
  - Max(x, y) is always $\geq$ GCD(x, y)