Programming Languages
Lecture 3: Functional Languages

Benjamin J. Keller

Department of Computer Science
Virginia Tech
keller@cs.vt.edu
Lecture Outline

- Motivation for FP
- Commands vs. Expressions
- History of Functional Languages
- ML
Motivation

- John Backus, 1978 Turing Award Lecture
- Imperative programming languages too restrictive
- Abstractions of von Neumann architecture
- Antiquated way of thinking (from '50's)
Von Neumann Bottleneck

• Computer has
  – CPU with accumulator and registers
  – Memory
  – Bus between memory and CPU (von Neumann bottleneck)

• Execution of machine statement
  – fetch — move instruction from memory to CPU
  – decode — break into parts
  – execute — interpret
Example

- Execute instruction \texttt{ADD 162}

  1. Fetch instruction from memory
  2. Decode into operation (\texttt{ADD}) and address (162)
  3. Fetch contents from address 162
  4. Add contents to accumulator

- Simple statements require many transfers through bus
Imperative Languages

- Program can be viewed as control statements guiding execution of assignment statements.

- Assignments are accesses and stores to memory

- Variable refers to memory location where contents can change

- Value of $x+1$ not same throughout program
Imperative Languages

- Order of execution important (hard to perform in parallel)

- Changing values makes reasoning about variables difficult

- Hard to reason about programs
Mathematical Perspective

• Use of variables in mathematics

• Variables are static

• *referential transparency* — can replace an expression anywhere that it occurs by its value without changing result of program

• Key idea: compute result once and then reuse

• Good for parallelism

• Imperative languages not referentially transparent (x+1)
Advantages of Functional Programming

- Referentially transparent — easier to reason about, easier to parallelize

- Order of execution need not be specified — evaluate expressions when necessary

- Higher-level — shorter, more understandable programs

- Flexibility in combining old programs to form new ones

- “Lazy” evaluation allows computing with infinite data objects
Other Reasons for FP

- Useful in AI programming

- Useful in developing executable specifications and rapid prototyping

- Closely related to topics in theoretical CS (recursive functions, denotational semantics).
Commands/Imperative Languages

- Support for variables — represent memory locations for storing updatable values

- Assignment operation — computation depends on changes to values stored in variables

- Repetition — flow of control guided by loops and conditional statements
Imperative Languages

- Based on commands (statements)

- Meaning of command is operation which modifies the current contents of memory, based on current contents of memory and explicitly provided data

- Results of one command communicated to next command through changes to memory

- Highly dependent upon computer architecture
Expressions

- Return a value, depending on state of computation

- Examples
  - Literals: 3, true, "a string", 42.323
  - Aggregates: arrays, records, sets, lists, .... Ex. {1,3,5}
  - Function calls: f(a,b), a+b*(c-d),
    (if x>0 then sin else cos)([[pi]])
  - Conditional expressions:
    if x <> 0 then a/x else 1,
    case (only in functional languages)
  - Named constants and variables: pi, x
Expressions

- Mathematical expressions better behaved than commands

- Meaning of a *(pure)* expression is operation that returns a value based on current contents of memory and explicit values
Referential Transparency

- System is referentially transparent if in fixed context the meaning of the whole system can be determined by meaning of its parts.

- Independent of surrounding expression

- Once expression is evaluated in a particular context its value in that context will not change

- Mathematical expressions referentially transparent

- Context: $a = 3, b = 4, c = 7, x = 2$

- Evaluating $(2ax + b)(2ax + c)$ only requires evaluating $2ax$ once
Ref. Trans. Examples

- Can determine meaning of $f(g(x))$ by knowing independent meaning of $f$, $g$ and $x$

- If know that $g'$ is the same as $g$, then know $f(g(x))$ is the same as $f(g'(x))$

- Equivalences important for program transformations used in optimization
Side Effects

- Side effect — expression does more than return value

- Example $f(x)$ returns a value but also increments $x$ by 1

- Lose referential transparency if side effects allowed

- Can’t count on $f(x) + f(x)$ being the same as $2*f(x)$

- Easier to prove a program correct if referentially transparent
Imperative Languages and Ref. Transparency

• Lose referential transparency with imperative languages

• Consider $x : x + y; y := 2 \times x$; and $y := 2 \times x; x : x + y$;

• Rationale:
  – Each command changes underlying state of computation
  – Evaluation depends on state
  – Ordering critical
Issues with Expressions

• Order of evaluation

  – Ex. short-circuited evaluation of boolean expressions

  – if $i \geq 0$ and $A[i] \neq 99$ then ...

  – What if int $A[100]$ and $i = -1$?

• Side effects

• Treating expressions and commands identically (Algol 68, C)

  – Artificial and loses referential transparency

  – $x = (y = x + 1) + y + (x++)$

  – Compare $2*(x++)$ and $(x++)+(x++)$
Pure Functional Languages

• Program is application of function to data

• Pure expressions — no side effects

• Expressions and functions are *first class* (used as data)

• No traditional notion of memory or assignment

• Promote reasoning about programs

• Support parallel implementation
History of Functional Languages

• Theoretical foundations:
  
  – Gödel’s general recursive functions
  
  – Use of lambda calculus by Church and Kleene as model of computable functions
  
  – Church’s thesis

• LISP — John McCarthy (1958-60). Originally used for symbolic differentiation with linked lists. Many dialects. Finally, Common LISP and Scheme.
History (cont)

- *Denotational semantics* — meaning of programs as functions (1960’s)

- Backus’ Turing award lecture, 1978. Language called FP (now FL).


- Other languages SASL, KRC, Miranda (David Turner), Haskell. Use lazy evaluation.
Schools of Functional Languages

• LISP/Scheme  
  (dynamic typing, imperative)

• Strict (eager evaluation) — ML, Hope  
  (static typing, imperative, polymorphic functions, type inference)

• Lazy (evaluation) — Miranda, Haskell  
  (static typing, polymorphic functions, type inference)