Programming Paradigms and Languages

Christian Schulte
Johan Montelius

Software and Computer Systems
School of Information and Communication Technology
KTH – Royal Institute of Technology
Stockholm, Sweden
Overview

- Programming paradigms and features
  - functional programming
  - concurrent programming

- Programming languages
  - language description
    - syntax and semantics
  - language execution
    - compilers and execution environments

- Other courses
Functional Programming
Characteristics

- Computation amounts to recursive evaluation of expressions
  - expressions defined by functions
  - typically, functions are first-class citizens: generic functions

- Pure functional programming
  - value of expressions only depends on values of subexpressions
  - no side effects
  - functions can be understood in isolation
Parametric Polymorphism

- **First-class functions support parametric polymorphism**
  - functions can compute for different types of arguments
  - length of lists:
    - parametric wrt type of list elements
  - map over list:
    - parametric wrt function provided
    - function type fits type of list elements

- **Other forms of polymorphism**
  - ad-hoc polymorphism:
    - overloading
  - object-based polymorphism:
    - programs can compute for different types of objects
    - implemented by late binding (Java, virtual in C++)
Applications

- Anything that has to do with symbolic processing.
- When correctness and time to market is more important than resource utilization.
- Check the following webpage
  - Functional Programming in the Real World
FP Inspired: MapReduce

- Users specify
  - map function that maps key/value pair to intermediate key/value pair
  - reduce (fold) function merges intermediate values associated with the same intermediate key

- Programs written in this functional style are automatically parallelized and executed on a large cluster of commodity

- MapReduce runs on large cluster of commodity machines and is highly scalable: many terabytes of data on thousands of machines
Typing Disciplines

- Different approaches to typing possible
  - static versus dynamic typing: check types at compile time (static) or at runtime (dynamic)
  - weak versus strong typing: types can be changed and are possibly changed automatically

- Examples:
  - Erlang: strong, dynamic
  - Haskell: strong, static
  - Java: strong, static
  - C++: weak, static
Static Typing in FP

- One goal of functional programming is simplicity and conciseness
  - functions as simple mathematical model
  - data types with pattern matching for conciseness

- Declaring types puts burden on programmer
  - in contrast to simplicity of FP

- Idea in FP: type inference
  - automatically infer types from program
  - check that inferred types are used consistently
  - infer most general types that works with polymorphism
Type Inference

- Infer type of expression by how it is used or defined

\[
\text{len}([]) \rightarrow 0;
\]

\[
\text{len}([\_|Xr]) \rightarrow \text{len}(Xr) + 1.
\]

- 1 has type int
- \text{len}(Xr) must have type int (due to +)
- \text{len}/1 must have return type int
- \text{len}/1 must have argument type list
- has (polymorphic) type: \(\forall \alpha. \text{list}([\alpha]) \rightarrow \text{int}\)
Type Inference

- Infer type of expression by how it is used or defined

map(F,[]) -> [];

map(F,[X|Xr]) -> [F(X)|map(F,Xr)].

- F has type $\alpha \rightarrow \beta$
- F(X) has type $\beta$
- X has type $\alpha$
- map/2 has type $\forall \alpha. \forall \beta. (\alpha \rightarrow \beta) \times \text{list}(\alpha) \rightarrow \text{list}(\beta)$

- Hindley-Milner (Damas-Milner) type system
  - support automatic type inference of polymorphic types
Problem in FP: functions can have side effects (IO, message sending)

Leads to inconvenient restrictions
- guards can be not user-defined
- no referential transparency

Other solutions
- make side effects manifest in types (monads)
- declare that functions are side effect free (const in C++)
Functional Programming Languages

- Dynamically typed languages
  - Erlang, Scheme, LISP

- Statically typed with type inference
  - Standard ML (OCaml)
  - Haskell
  - F#

- Widely used in education, ...
Logic Programming

- Logic programming: relations instead of functions
  
  ```prolog
  app([], Ys, Ys).
  app([X|Xr], Ys, [X|Zr]) :- app(Xr, Ys, Zr).
  ```
  - uses relational representation of append function

- Can be used in several directions
  - app([1,2], [3,4], Zs)
    
    Zs = [1,2,3,4]
  
  - app(Xs, Ys, [1,2])
    
    Xs = [], Ys = [1,2]; Xs = [1], Ys = [2]; Xs = [1,2], Ys = []
  
  - uses search and unification

- Implemented by, for example, Prolog
Logic Programming

- Logic programming: relations instead of functions
  
  ```prolog
app([], Ys, Ys).
app([X|Xr], Ys, [X|Zr]) :- app(Xr, Ys, Zr).
```

  - uses relational representation of append function

- Can be used in several directions
  
  ```prolog
  app([1,2], [3,4], Zs)
  Zs=[1,2,3,4]
  ```
  
  ```prolog
  app(Xs, Ys, [1,2])
  Xs=[], Ys=[1,2]; Xs=[1], Ys=[2]; Xs=[1,2], Ys=[]
  ```

  - uses search and unification

- Implemented by, for example, Prolog
Concurrent Programming
Models

- **Share nothing between threads**
  - data structures are private to a thread
  - threads communicate by passing data structures as messages
  - example: Erlang

- **Share data structures between threads**
  - heap-allocated objects are shared
  - threads communicate by access to shared data structures
  - examples: Java, pthreads for C++
Concurrency in Java

- Create threads by
  - create an object that implements a runnable interface (`java.lang.Runnable`), in particular a `run()` method
  - create a new thread (`java.lang.Thread`) that executes `run()` method of runnable
  - threads are heavy weight, much more expensive than Erlang's processes
Synchronization in Java

- Java guarantees that most primitive operations are atomic
- Methods (and code blocks) can be synchronized
  - each object has a lock
  - at most one thread can hold the lock
  - threads will be blocked if lock is already taken
- Threads waiting on lock can be notified
  - supports variants: wait, notify, notifyAll
Concurrency Complexity

- Concurrency in Erlang is natural
  - no explicit synchronization required
  - all synchronization is automatically tied to receive

- Concurrency in Java requires careful design
  - which methods must be synchronized
  - what is interaction between several threads using the same object

- Orthogonal: how to guarantee liveness and safety
Distributed Programming

- Distribution presupposes concurrency
  - concurrent activities execute in parallel

- Sharing with distribution
  - difficult to achieve fully
  - Java (RMI) uses approximation

- Message passing is straightforward for distribution
  - semantic gap smaller
Language Description
Describing Languages

- Syntax of programming language
  - syntactical form of programs, statements, expressions, ...

- Semantics of programming language
  - describes the meaning of a program
  - what is the result
  - how to prove a program correct
  - how is it computed
Syntax

- Syntax description on two levels
  - lexical: how words are formed
  - syntactical: how sentences are formed from words

- Lexical
  - words (lexemes) are grouped into tokens
  - typical token types: identifiers, number, ...

- Syntax
  - phrasal structure of a program, expression, ...
  - syntactical structure represented by a tree
Lexical Structure

- Described by: regular expressions
  - convenient method to describe tokens

- Serve as description for programmer and compiler
  - lexer in compiler creates sequence of words
Regular Expressions

- **Symbol** \( a \)
  - denotes language just containing string \( a \)

- **Alternation** \( M|N \)
  - where \( M \) and \( N \) are regular expressions
  - string in language of \( M|N \), if string in language of \( M \) or in language of \( N \)

- **Concatenation** \( M\cdot N \)
  - where \( M \) and \( N \) are regular expressions
  - string in language of \( M\cdot N \), if concatenation \( \alpha\beta \) of strings \( \alpha \) and \( \beta \) such that \( \alpha \) in language of \( M \) and \( \beta \) in language of \( N \)
Regular Expressions

- **Epsilon** $\varepsilon$
  - denotes language just containing the empty string

- **Repetition** $M^*$
  - where $M$ is regular expression
  - called Kleene closure
  - string in language of $M^*$, if concatenation of zero or more strings in language of $M$
Regular Expression Examples

- $a|b$  \{"a","b"\}
- $(a|b)\cdot a$  \{"aa","ba"\}
- $(a\cdot b)|\varepsilon$  \{"ab",""\}
- $((a|b)\cdot a)^*$  \{"","aa","ba",
"aaaa","aaba",
"baaa","baba",
...\}
Syntactical Structure

- Described by context-free grammar (CFG)
  - describes how a correct program can be derived according to grammar rules
  - syntactical structure defined by derivation tree (parse tree)
  - related approaches: BNF and EBNF

- CFG serves as description for programmer and compiler
  - compiler uses parser to construct parse trees
Context-free Grammar

- Describes language
- Has productions of form
  \[ \text{symbol symbol symbol symbol ... symbol} \]
  - zero or more symbols on rhs
- Symbols are either
  - terminal token from alphabet
  - nonterminal appears on lhs of production
  - no token ever on lhs
- One nonterminal distinguished as start symbol
Example: Simple Programs

1: $S \implies S ; S$
2: $S \implies \text{id} := E$
3: $S \implies \text{print} (L)$
4: $E \implies \text{id}$
5: $E \implies \text{num}$
6: $E \implies E + E$
7: $L \implies E$
8: $L \implies L , E$
Example Grammar

- **Terminal symbols**
  - id  num  print  +  ,
  - (  )  :=  ;

- **Nonterminals:** S, E, L

- **Sentence in language**
  id:=num; id:=id + (id:=num + num, id)

with possible source text

a := 7;

b := c + (d := 5 + 6, d)
Derivations

- Showing that sentence is in language: perform derivation
  - start with start symbol
  - repeat: replace nonterminal by one of its rhs

- Many different derivations possible
  - leftmost  replace leftmost nonterminal
  - rightmost  replace rightmost nonterminal
Derivation Example

S
Derivation Example

\[ s \]
\[ s ; s \]
Derivation Example

\[ S \]
\[ S ; S \]
\[ S ; \text{id} := E \]
Derivation Example

\[ S \]

\[ S \; S \]

\[ S \; \text{id} := E \]

\[ \text{id} := E; \; \text{id} := E \]
Derivation Example

\[ S \]
\[ S ; S \]
\[ S ; \text{id := } E \]
\[ \text{id := } E ; \text{id := } E \]
\[ \text{id := num;} \text{id := } E \]
Derivation Example

\[
S
\]

\[
S ; S
\]

\[
S ; \text{id} := E
\]

\[
\text{id} := E; \text{id} := E
\]

\[
\text{id} := \text{num}; \text{id} := E
\]

\[
\text{id} := \text{num}; \text{id} := E + E
\]
Derivation Example

\[ S \]
\[ S ; S \]
\[ S ; \text{id} := E \]
\[ \text{id} := E; \text{id} := E \]
\[ \text{id} := \text{num}; \text{id} := E \]
\[ \text{id} := \text{num}; \text{id} := E + E \]
\[ \ldots \]
\[ \text{id} := \text{num}; \text{id} := \text{id} + \text{id} \]
Parse Tree

- Parse tree created
  - connect each symbol in derivation to symbol from which it was derived

- Different derivations can have same parse tree

- But also: same sentence can have different parse trees...
  - that's bad: what does the sentence mean
Example sentence  
\text{id} := \text{id} + \text{id} + \text{id}

Derivation  
\text{S} \rightarrow \text{id} := \text{E}  
\rightarrow \text{id} := \text{E} + \text{E}  
\rightarrow \text{id} := \text{E} + \text{id}  
\rightarrow \text{id} := \text{E} + \text{E} + \text{id}  
\rightarrow \text{id} := \text{id} + \text{E} + \text{id}  
\rightarrow \text{id} := \text{id} + \text{id} + \text{id}
Example sentence

```latex
id := id + id + id
```

Derivation

```latex
S \rightarrow id := E \\
\rightarrow id := E + E \\
\rightarrow id := E + id \\
\rightarrow id := E + E + id \\
\rightarrow id := E + id + id \\
\rightarrow id := id + id + id
```
Example sentence: \( \text{id} := \text{id} + \text{id} + \text{id} \)

Derivation:

\[
S \rightarrow \text{id} := E \\
\rightarrow \text{id} := \text{id} + E + E \\
\rightarrow \text{id} := \text{id} + \text{id} + E + E \\
\rightarrow \text{id} := \text{id} + \text{id} + \text{id} + \text{id}
\]
Ambiguous Grammars

- Ambiguous grammar
  - can derive sentence with two different parse trees
  - our example grammar is ambiguous
  - can be rewritten to unambiguous grammar: every sentence has exactly one meaning

- Problematic for compiling
  - compilers derive meaning from parse trees
Language Semantics

- Model and explain
  - what is computed: axiomatic and denotational
  - how it is computed: operational

- Example: MiniErlang semantics
  - is a model of how Erlang computes
  - designed to explain computation
  - can serve as a blueprint of an implementation
  - defines the language
Semantics

- **Axiomatic semantics**
  - meaning of statements defined by assertions
  - useful for proving (verifying) that programs are correct wrt specification
  - example: Hoare logic, weakest preconditions

- **Denotational semantics**
  - define meaning of a program
  - useful for deep analysis
Language Execution
Compilation and Execution

- How to execute program written in some high-level programming language

- Two aspects
  - compilation
    - transform into language good for execution
  - execution
    - execute program
Compiler

- Compiler translates program from one programming language into another
  - language compiled from source language
  - language compiled to target language

- Source language: for programming
  - examples: Java, C, C++, Oz, ...

- Target language: for execution
  - examples: assembler (x86, MIPS, ...), JVM code
Execution

- Can be by concrete hardware
  - how to manage memory
  - how to link and load programs
  - take advantage of architectural features

- Can be as abstract machine
  - how to interpret abstract machine code efficiently
  - how to further compile at runtime
Compilation Phases

- Frontend depends on source language
- Backend depends on target architecture
- Factorize dependencies
Frontend: Tasks

- **Lexical analysis**
  - how program is composed into tokens (words)
  - typical token classes: identifier, number, keywords, ...
  - creates token stream

- **Syntax analysis**
  - phrasal structure of program (sentences)
  - grammar rules describing how expressions, statements, etc are formed
  - creates syntax tree

- **Semantic analysis**
  - perform identifier analysis (scope), type checking, ...
  - creates intermediate representation: control flow graph
Intermediate Representation

- Control flow graph
  - nodes are *basic blocks*: simple and abstract instructions, no incoming/outgoing jumps
  - edges represent control flow: jumps, conditional jumps (loops), etc

- Basic blocks
  - typically contain data dependencies: reading of and writing to same location must be in order
  - ease reordering by conversion to SSA (static single assignment) form: new locations assigned only once
Backend: Basic Tasks

- **Optimization**
  - reduce execution time and program size
  - typically independent of target architecture
  - intermediate and complex component: "midend"

- **Instruction selection**
  - which real instructions for abstract operations

- **Register allocation**
  - which variables are kept in which registers?
  - which variables go to memory

- **More generic: memory allocation**
Optimization

- **Common subexpression elimination (CSE)**
  - reuse intermediate results

- **Dead-code elimination**
  - remove code that can never be executed

- **Strength reduction**
  - make operations in loops cheaper: instead of multiplying with \( n \), increment by \( n \) (iterated array access)

- **Constant/value propagation**
  - propagate information on values of variables

- **Code motion**
  - move invariant code out of loops

- Many, many more, ...
Instruction Selection: What

- Clearly depends on instruction set
- For abstract operations emit target machine instructions
  - several operations by one instruction (CISC)
  - one operation by several instructions (RISC)
- Depends much on regularity of instruction set
  - typically simple: RISC architectures
  - can be involved: CISC architectures
    - register classes, two address instructions, memory addressing, ...
Instruction Selection: How

- Criterion: fewest instructions or fewest clock cycles
- Using matching algorithm
  - maximal munch
  - tree grammars
  - dynamic programming
- Simple optimizations: peephole optimization
  - combine, remove, change instructions on simple local rules
Register Allocation

- Find for each temporary value a register if possible: depends on number of registers
- If no register free, put temporary in memory
- Use spill code: free register temporarily
  - move register to memory
  - reload register from memory
- Common technique: register allocation by graph coloring
Register Allocation: Graph Coloring

- Instruction selection assumes infinite supply of temporaries
- Compute liveness information: when is temporary used first and last
- Construct interference graph
  - nodes are temporaries
  - edge between two nodes if live at same time
- Color interference graph: no two connected nodes get same color
  - colors correspond to registers
Additional Techniques

- Use precolored nodes for certain registers
- Handle different cases for registers
  - registers for passing arguments
  - register for return address
  - caller-save registers
  - callee-save registers
- Allows better register utilization
  - more registers available for intermediate results
  - spilling handled in regular and simple way
Impact of Register Allocation

- Graph coloring works well with many (say 32) registers

- Few registers (Pentium 6-8)
  - enormous amount of spilling
  - temporaries even spilled in loops!
  - example: 163,355 instructions [Appel, George, PLDI01]
    - 32 registers  84 spill instructions
    - 8 registers  22,123 spill instructions (about 14%)

- solution: exploit memory operands in CISC instructions
Summary: Basic Tasks

- Basic and simple tasks
  - instruction selection
  - register allocation

- Impact of target architecture
  - number of registers
  - register classes
  - instruction set
  - clock cycles per instruction
Other Courses

- Semantics for Programming Languages
  - DD2454, Dilian Gurov, CSC
- Concurrent Programming
  - ID1217, Vlad Vlassov, ICT
- Compilers And Execution Environments
  - ID2202, Christian Schulte, ICT
- Distributed Programming
  - ID2201, Johan Montelius, ICT
    - with two follow-up courses
Other Courses

- Logic Programming
  - ID2213, Thomas Sjöland, ICT

- Constraint Programming
  - ID2204, Christian Schulte, ICT

- System Modeling And Simulation
  - IV1200, Rassul Ayani, ICT