COMP 3031 Final Presentation

By LEUNG, Hang Kam

SID: 20425828

Email: hkleungai@connect.ust.hk

Course Overview

Prerequisites Under The Hood

- Solid programming background — Preferably in C or C++ or even Java.
  - Patience and bravery for embracing new languages within 1-2 weeks.
- Some knowledge in data structure and discrete mathematics.
  - Tree, Stack, Heap, etc.
- Minimal knowledge in commonly used algorithm.
  - Somewhat between COMP2711(H) and COMP3711(H).
  - e.g. In this semester we are asked to deal with graph-search in assignment.
- Be ready for linux-based working environment — Vim / Nanos, CLI tools, etc.

Main Topics

- Comparative studies of programming constructs.
  - Non-imperative paradigms — functional, logic, concurrent.
  - Each one of them give rise to one programming assignment.
- Program parsing — translation and interpretation.
  - Also appears in programming assignment.
- Storage allocation and run-time organization.
  - Related to the topic of concurrent programming.
- Procedual Activation.

Functional Programming

Main Features Of Functional Programming

- Function as first class citizens.
  - As an argument for other functions.
  - As a return value of some other functions.
  - As a variable, indistinguishable from any other types of native variables.
- Program computes by constructing, composing and applying functions.
- Native recursion for control-flow instead of looping.
- Free from (most) side-effects — No I/O, file read/write available.

High Order Functions With SML

- Lambda — A non-static function definition that takes input(s) with non-void return.
- High Order Functions that built upon lambdas give **immutability of data**.
  - Map — Compute input array element by the given lambda to give an output.
  - Filter — Inject array elements that satisfies the given lambda to an output.
  - Reduce — Compute the input array linearly by the given lambda.
- These are taught together with SML
  - It offers `list` (array), `tuple` (mixed-type array), `record` (JSON).
  - Also offers features like static scoping and strong typing.

Program Parsing

Context-Free Grammars

- Any language description contains Syntax and Semantics.
- CFG $\rightarrow$ tokens $+$ non-terminals $+$ production $+$ starting non-terminal.
  - Together they describe and specifics different **strings**.
  - Often be written in *Backus-Naur Form*.
- Parse Tree $\rightarrow$ token as leaf $+$ nonterminal as node $+$ starting nonterminal as root
  - Strings are formed by reading the terminals at leaves from left to right.
- Syntactic Ambiguity $-$ A string can be formed by more than one parse trees.
  - Undesirable for program parsing. (COMP3721 has more discussions on it.)

Regular Grammars

- A subset of CFGs with only left-linear or right-linear productions.
- Involve operations like parenthesis, counters, concatenation, and disjunction.
- Commonly used for pattern matching in editors, word processors, CLI, etc
  - Unix (vi, emacs, perl, grep)
  - Microsoft Word v6+

Expressions

- Expressions appear as a function applying on arguments of sub-expressions.
- Binary expressions often take an **infix form**. e.g. $a\ f\ b$ instead of $f(a, b)$.
  - Precedence — Priority among different operators
  - Associativity — Priority among the same operator in multiple occurences.
- Also expressions can take prefix form, $f\ a\ g\ b\ c$ instead of $f(a, g(b, c))$.
  - Favour left-to-right linear scan.
- Or they can take postfix form, $a\ b\ c\ g\ f$ instead of $f(a, g(b, c))$.
  - Can be easily evaluated with a stack data structure.

Flex && Bison

- Flex — Fast Lexical Analyzer
  - Reads a scanner description and outputs a C program with `yylex()` routine.
  - Typical `.lex` file contains RE patterns and the corresponding C actions.
  - `yylex()` scans input from left to right to look for strings with RE patterns.
- Bison — A program that generates a parser for a given CFG.
  - Reads a CFG description and output a C program with `yyparse()` routine.
  - Like flex, a `.y` file contains grammar Rules and the corresponding actions.
- Flex + Bison $\rightarrow$ a complete set of parser and scanner.

Logic Programming

Relations

- A program in a logic programming language partially specifies a set of relations.
- Relations treat arguments and results uniformly.
  - For n-relations, we can supply any $n - 1$ args and retrieve the desired value.
- A table completely specifies a relation.
  - A n-tuple *belongs to* a relation iff it constitutes to any one row of that table.
- We may treat relation as **predicate**.
  - Instead of saying $\vec{x} \in R$, we may say whether $R(\vec{x})$ is true.
- Sometimes it is easier to introduce recursiveness to a relation definition.

Framework

- **Query** $\Rightarrow \left[\textbf{Knowledge Base} \rightarrow \textbf{Theorem Prover} \right] \Rightarrow$ result.
  - Each Query asks whether some tuple belongs to a relation.
  - Knowledge Base forms with a predefined set of facts and rules.
  - Theorem Prover exhausts the Knowledge Base with the given Query.
- Hence a logic program is written in a set of true relations,
  with incoming query (from user) as driver to the program.

Prolog

- Contains data-type like atom, (anonymous) variables, and numbers.
- Relation names are referred as **functor**
  - We call `<functor> (<terms>)` a **structure** or a compound terms.
- Relations can be formed with either *facts* or *rules*.
- List and tail recursion are still available in Prolog.
  - Equipped with operations like `append()`, `member()`, `split()`, etc.
  - But NO MORE iterators like loop or control like if-else.
- Advanced concept like Substitutions and Unification, Search Trees, Cut.

Concurrent Programming

Parallel Tasks

- Program tasks with minimial dependency are able to executed in parallel.
  - Like array's elementwise computation, matrix operation, etc.
  - Helpful for reducing *spreading out* the cost from one machine to many others.
- Parallelism achievement replies on several factors.
  - Synchronization — determine the percedence of one task over another.
    - Cooperation synchronization, and Competition synchronization.
  - Scheduling — determine the current best choice of executable task.
- Producer-Consumer Problem — Balancing for service providers and users.

Statement-level Concurrency

- Instruct the compiler to map a program onto a multiprocessor machine, by
  - specifications about the number of processors,
  - statements about how data distribution over the processors memories,
  - statements about the alignment of data.
- CUDA and HPF (high-performance Fortran) are good examples.
  - `BLOCK` — Divide and distribute evenly by the number of processors.
  - `CYCLIC` — Ciculate the task distribution by the number of processors.

CUDA Programming

- Suitable for general-purpose computing applications that exhibit data parallelism
- GPU utilization $-$ Host $\Rightarrow \left[\text{Device} \rightarrow \text{GPU} \rightarrow \text{Device} \right] \Rightarrow$ Host.
  - Single Instruction $-$ Every thread run with a same set of program
  - Multiple Data $-$ Each thread runs on different element.
- Some performance issue often worth to be explored.
  - Warp Divergence $-$ Unserialized execution on parallel group of threads.
  - Shared Memory Usage $-$ Prefer on-chip shared memory over global memory.
  - Coalesced Memory Access $-$ Prevent running out of threads.

The End