G6021 Comparative Programming

Part 6 - Summary

Overview

Programming paradigms:

- Functional
- Object oriented
- Logic programming
- Imperative

Emphasis on functional programming in Haskell for the labs, however, the exam will be more balanced.

Main Topics

- Types: subtypes, polymorphism, overloading
- Semantics: Operational, denotational
- Foundations: λ-calculus for functional, Turing Machines for imperative, resolution for logic, ς-calculus for object-oriented. λ-calculus concepts explain many concepts found in modern programming languages. Unification for logic programming.
- Implementations: we've implemented them (functional, imperative, logic) in the labs, so you should have some ideas about memory usage, etc. Referentially transparent: always gives the same answer.
- Declarative: What. Imperative: How.
- Ability to critically compare.

Revision

A programming language may enforce a particular style of programming, called a *programming paradigm*.

- **Imperative Languages**: Programs are decomposed into *computation steps* and routines are used to modularise the program. C is an example.
- **Functional Languages**: Based on the mathematical theory of functions; *declarative*: focus on *what* should be computed not *how*. Computation by *evaluation*. Haskell.
- **Logic Languages**: Describing problem by formulas in logic; *declarative*. Computation by *resolution*. Prolog.
- **Object-oriented Languages**: Creating hierarchies of objects. Computation by (dynamic) *object update*. Java.

imperative (Turing Machines)
everything is a command

1	19	1	2	1	1	12	1	state

`x:=x+1;` ↓ transition

1	19	1	2	2	1	12	1	state

functional (λ-calculus)
everything is a function

`(\x->x+1)5` expression

β ↓ evaluation

`5+1` expression

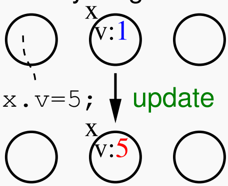
logic (predicate logic)
everything is a formula

`...,a,b,c,b->d,...` theory

resolution ↓ inference

`...,a,b,d,c,b->d,...` theory

object-oriented (ς-calculus)
everything is an object



objects

update

objects

Implementation

- Imperative languages: Programs mapped to memory manipulation instructions.
- Declarative Languages: implemented through abstract machines, and thus do not map directly to memory manipulations.
  - functional languages, e.g. Haskell: Lambda calculus
  - logic programming, e.g. Prolog: resolution

Which are easier to implement? Efficiency?

Types

- Functional: types very important in languages like Haskell. Every program, sub-program, has a type, and this is preserved under reduction.
- Object oriented: subtypes. Types used to structure data
- Imperative: where are types used here?
- Prolog: types not really used (arity, mode)

Main issues:

- Polymorphism: parametric, ad hoc (what is the difference?)
- Type checking/inference (what is the difference?)

Programming

Programming with languages like Haskell and Prolog:

- Easier to program?
- Learning curve?
- Data types - efficiency?

I assume you have done all the lab exercises....

- Ability to write a simple function.
- Lists, Trees, pattern matching
- Higher-order functions: write and use: map, fold, etc.
- Accumulating parameters: tail recursive functions.
- Use some specific features of Haskell: data type declarations, pattern matching, list comprehension, etc.

Types

Type inference (simple examples; $\leq$ 6 nodes). May involve *unification* algorithm on types (based on *disagreement sets*).

Knowing how to give a program a type in Haskell.

- How to type Haskell functions informally (see exercises)
- Start with what you know, and build up. (Use the types of built-in functions also: head (or hd), tail (or tl), ++ (or app), etc.) Examples:

```
add1 x = x+1
tl [1,2,3]
apply f x = f x
twice f x = f (f x)
(+) 3
(++) [True]
\(x,y,z) -> y
```

- Building derivations: Example: $\vdash \lambda xy.x : A \to B \to A$

Disagreement set and Unification

Checklist:

- Can you unify two types? Draw type tree to help see the structure.
- Example: $(((A \rightarrow A) \rightarrow B) \rightarrow B$ and $C \rightarrow \text{int}$

Accumulating parameters

- Know how to convert a given function to a version that uses an accumulating parameter.
  Example:

  ```
  power x y = x * power x (y-1)


  power x y z = power x (y-1) z*x
  ```

- Writing simple examples in CPS: factorial, reverse of list, etc. (So how to convert a simple function so that it is tail recursive.)

Important topics:

- Reduction: know how to reduce a lambda term.
- Reduction graphs. Show all reduction sequences.
- Strategies: the order in which we reduce redexes
- Normal forms: the answers (when we stop reducing)
- Writing functions as a fixpoint of a functional

Objects

- Dynamic look-up, abstraction, subtyping and inheritance.
- Multiple inheritance: problems with this, and how to overcome them.

Logic Programming

- Declarative, knowledge-based programming: the program just describes the problem.
- Advantages/Disadvantages?
- Termination: order of the clauses important (relation with Haskell pattern matching)
- Evaluate a simple Prolog program (SLD trees; to show understanding of unification and resolution)

Exam

The format of the exam is standard:

- Answer TWO OUT OF THREE questions
- Candidates should answer ONLY TWO questions

Looking forward:

- Use the material (lecture recordings / exercise sheets with answers / papers) on canvas to help with your revision
- Contact me in case of questions / requests
  (will be away from the 13th of December till 1st of January)
- Good luck!


DO YOU HAVE AN IDEA FOR YOUR STORY YET?
NO, I'M WAITING FOR INSPIRATION.
YOU CAN'T JUST TURN ON CREATIVITY LIKE A FAUCET. YOU HAVE TO BE IN THE RIGHT MOOD.
WHAT MOOD IS THAT?
LAST-MINUTE PANIC.