

— Summary —
Core Themes & Unifying Principles

COMP1100 — Introduction to Programming and Algorithms

Clem Baker-Finch

Australian National University

Semester 1, 2006

Types, Types, Types!

- Types *classify program behaviour* according to the *kinds of values* they compute.
- Types are a basic organising principle of Computer Science.

None of this is language specific.

Types are good for:

- *Detecting errors*
- *Documentation*

Since types “classify program behaviour,” reading the type signature of a function sets a framework for understanding its detailed behaviour.
- *Abstraction*

The fundamental activity in *Data Directed Design* is to identify the “types” of things involved in the problem domain.
- *Other stuff you might find out about later, e.g. efficient implementations.*

Data Directed Design

Designing and constructing programs is about identifying and characterising *abstract data types* in which to express the problem or application.

(Data Directed Design steps 2 and 3.)

That word again: **Abstraction**.

Data Structures *Suggest Algorithms*

For example, if we choose to represent an ADT using a *tuple* then we should expect the algorithms on that ADT to be in terms of *selection*.

If we are using a *list* then we should expect the algorithms to use *traversal*.

If we are using an *algebraic data type* then we should expect to have *separate cases* for the alternatives and to *select* the components from each alternative.

Lists

Lists consist of a *variable number* of elements of the *same type*.

So...

Algorithms on lists consist of doing the *same thing* with each of the elements.

Since there is a variable number of elements, that process will occur a *variable number of times*.

We have *repetition!*

Repetition

Lists consist of a number of elements of the *same type*,

so...

List algorithms are structured around *repetition*.

Recursion is a fundamental way of expressing repetition.

Recursion is all we ever need to express *any* repetitive algorithm.

Patterns of Repetition

With experience, we observe the same *patterns* of repetition occur again and again in different algorithms.

We can **abstract** and identify the pattern.

By using the *pattern* (instead of its implementation using recursion) programs are easier to develop and understand.

In Haskell such patterns appear as *higher order functions*.

In Java such patterns appear as (built-in) *loops*.

Reasoning: safety critical systems

In some applications there is a need for formal proof of properties of programs — “correctness.”

(For examples, see **Peter G. Neumann**: *Computer-Related Risks*.)

Reasoning: your understanding

More generally, when we think about programs (e.g. when we are writing them) we are *reasoning* about them, perhaps informally.

There is a very close relationship between *inductive proof*, *recursive data structures* (such as lists) and *recursive algorithms*:

- The clauses of a recursive algorithm reflects the form of the recursive data structure
- The cases of an inductive proof reflect the form of the recursive data structure
- The cases of an inductive proof reflect the clauses of the recursive algorithm

Modularisation

The *problem-domain data types* identified in the *Data Directed Design* process suggest a clean and coherent way to break the overall program into *modules* — conceptually independent components of manageable size and complexity.

This is the fundamental motivation behind *object oriented languages* such as Java.

Classes correspond directly to the *problem-domain data types*.

Interlude



Perspective

We mostly used **Haskell** to *present* the core concepts of the course.

We *reinforced* the concept by expressing them in **Java**.

This was the main reason for looking at Java in this course.

The other reason: to provide a bridge to COMP1110/1510.

Don't forget about Haskell!

Don't stop using it!

Study Focus

- Don't over-emphasise Java just because we have been using it for the last few weeks.
- Most of the course used Haskell and most of the final exam will be in terms of Haskell, or perhaps not language specific.
- Don't be overwhelmed by the intricacies of Java — you have all next semester for that.
- Try to focus on the core concepts.

Aiming for a Particular Result?

(Be realistic.)

Pass or Credit: Focus on the material and concepts in (***) lectures.

For example, in Java concentrate on the *concept* of a class and an object, and on the alternative view of list processing.

Distinction: Material and concepts in (***) and (**) lectures.

High Distinction: Material and concepts in (***), (**), some (*) lectures. . .

. . . and cross your fingers.

Study Advice:

- Prioritise your focus based on your aims.
- Re-work prac exercises.
- Experiment with sample code from lectures.
- Study groups can be much more effective than a solitary effort.
- Study groups can provide extra motivation.

Good Luck!

