Software System Design and Implementation

Motivation & Introduction

Gabriele Keller
(Manuel M. T. Chakravarty)

The University of New South Wales
School of Computer Science and Engineering
Sydney, Australia
Motivation

What this course is all about?
Software affects our lives everywhere

• Mobile phones

• Social networks

• Cars & public transport

• Multimedia devices

• Medical systems

• Obviously, on computers at home and at work
Software is often mission critical.
Modern cars are computer networks on wheels

Next generation will have wifi...
So, why is software still so unreliable?
We have two competing goals

- Software must be of high quality: correct, safe & secure
  - Software defects affect many activities, often deeply
  - Software is increasingly used in safety and security critical applications
  - Financial success often depends on software success

- Software must be developed with low effort: cheap & quickly
  - Many products include increasing amounts of software
  - Products shouldn't get more expensive due to software
  - Product releases shouldn't be delayed due to software
Sometimes we can sacrifice one for the other

- **Computer games** *(development effort is key)*
  
  - Correctness and safety are secondary
  
  - Nobody notices a few pixel with the wrong colour
  
  - Who cares if the game physics isn't quite accurate

- **Flight management system** *(correctness is key)*
  
  - Safety and security is an overriding concern
  
  - Defects are very costly and may harm human life
Usually, we need a balance

- Consider testing for a mass market product
  
  ▪ If we test for a very long time with many testers
    - the product will be expensive and we ship too late, but with few defects
  
  ▪ If we do very little testing
    - the product will be cheaper and ship early, but be riddled with bugs

- In practice, you want to ship when the remaining bugs are unlikely to affect users (sales?) very much
Implications

1. We need to be able to *trade quality for reduced effort*

   ▶ To be broadly applicable, an approach to software design and implementation must support this trade off

2. We ideally want to *increase quality, while reducing effort*

   ▶ We seek novel approaches that solve both problems at once
Produce better software with less effort

• Better software
  – Software that has fewer defects (including security defects)
  – Software that is more usable — we won't talk about usability in this course

• Less effort
  – Shorter development time
  – Fewer programmers
  – Less-specialised programmers
The core theme of this course

How can mathematical tools help to produce better software with less effort?
Why are mathematical tools interesting?

• Mature engineering disciplines are based on mathematical foundations
  ‣ We rarely *guess* the input current to a semiconductor
  ‣ Would you build a bridge and then *test* whether it can hold its load?

• They enable a qualitative change in software development
  ‣ We want to increase quality, while reducing effort
  ‣ High-assurance software requires proof
What mathematical tools?

• Tools to reason about specifications and programs
  ‣ Theorem provers & proof assistants
  ‣ Static analysis tools & type checkers

• Tools to transform and refine programs
  ‣ Meta programming & generative programming
  ‣ Rewriting tools

• Advanced programming languages and environments
What is the basis of these tools?

∀λx. P(x)

Mathematical logic

Discrete mathematics

Calculus

The basis for reasoning about programs
Discrete mathematics

- Study of structures that are fundamentally discrete rather than continuous

- In a discrete space, individual points are isolated from each other — consider, natural numbers versus real numbers

functions in computing

state space of a computer

Photo by Un ragazzo chiamato Bi - http://flic.kr/p/4NFrtx
Mathematical logic

• Foundation of reasoning about computation, languages, and programs
  ‣ Mathematical basis for software design and implementation

• Originally arose out of the study of the foundations of mathematics
  ‣ On what ultimate basis can mathematical statements be called true?
  ‣ The foundations of mathematics were heavily disputed in the early 20th century
A little bit of history

• 1920s: Hilbert's program
  ‣ Base all of mathematics on a finite, complete set of axioms
  ‣ Show that they are consistent and system is decidable

• 1931: Gödel's incompleteness theorems
  ‣ Consistent, computable axiom set, covering arithmetic, can never be complete
  ‣ Such a system can't even prove it's own consistency
• Adapted form of Hilbert's program

  ‣ Instead of formalising all of mathematics, formalise the important parts

• There are many consistent and complete logics

  ‣ Gödel showed that first-order logic is complete
• 1928: Hilbert's Entscheidungsproblem ("decision problem")

  ▸ Is there an algorithm that, given a mathematical statement (in a formal language), will tell us whether the statement is true or false?

• Church (1936) & Turing's (1937) Church-Turing Theorem

  ▸ Showed that such an algorithm is impossible

• Church-Turing Thesis

  ▸ Recursive functions, Turing machines & lambda calculus are equivalent
Lambda calculus

• Minimal calculus that can express all computable functions
  ‣ Only variables, function abstraction, and function application

• The basis for many theorem provers

• The foundation of all functional programming languages
  ‣ Extend the lambda calculus with explicit data structures
  ‣ Add syntactic sugar to make it more convenient
Course contents

- Logical program properties
  - Basic logic & formal properties
  - Property-based testing
- Types help to design programs
  - Program properties as types
  - Types guide the design
  - Encapsulating properties
- Types help to implement programs
  - Types imply programs
  - Types control effects
  - Types prevent defects
- Effect control helps with parallelism
  - Effects interfere with concurrency & parallelism
Haskell

• A practical, strongly-typed functional programming language
  ▸ Widely used in research, industry & education
  ▸ Mature, highly optimising compiler with interactive environment
  ▸ Over thousands of open-source libraries and tools

• Named after the logician Haskell B. Curry

http://haskell.org/
Why Haskell?

• Functional languages are based on the lambda calculus
  ‣ Semantics of programs is fairly precisely defined
    ‣ This simplifies formal reasoning about these programs
  
• Functional languages can dramatically increase productivity
  ‣ Factor of four has been cited for Erlang versus C++

• Haskell has a very sophisticated type system

• Haskell has controlled effects
Software life cycle, high assurance & types
Stages in software development

- Requirements
- Specification
- Design
- Implementation
- Validation
- Maintenance
Waterfall model versus agile methods
Waterfall model

What are the properties?
Agile methods

What are the properties?

Specification → Design

Requirements

Design → Implementation

Validation

Implementation → Maintenance

Maintenance

Specification
Different projects require different methods

- Implementing an AES (Advanced Encryption Standard) component
  - Requirements and specification is not going to change significantly
  - Predictability and correctness are paramount

- Implementing a social media website
  - Requirements are initially very vague
  - Web users are well accustomed to half-baked features and a little downtime
Again, we need to be flexible

- We need to be able to **trade quality for reduced effort**

- We need to be able to **trade predictability for agility**
The scope of this course

- Requirements
- Specification
- Design
- Implementation
- Validation
- Maintenance
Logical program properties

• Our tool of choice for specifications

• They are flexible

  1. They can directly be used for testing

  2. They can directly be used for formal verification

• They are fundamentally connected to types
Type-driven development

• We will use types for all four stages of software development

  1. Specification — types can encode arbitrary properties

  2. Design — types structure code

  3. Implementation — types guide and sometimes imply implementations

  4. Validation — types can be automatically checked
Types provide flexibility

- **Singleton types** are perfectly precise
  \[ n : \text{SInt}(n) \]

- **Bit-size types** track an important implementation constraint
  \[ n : \text{BInt}(w) \]

- **Types as we know them**
  \[ n : \text{Int} \]

- **Dynamic types**
  \[ n : \text{Dynamic} \]
By making types more precise...

• We refine the specification

• The type checker requires us to justify our implementation in more detail

We gain quality, but also have to spend more effort
By making types less precise...

- We simplify experimentation
- We will have to perform more testing, or accept defects

We avoid fixing too many details of the specification
When specifications are types…

• Programs become proofs

• The object of the proof and the proof cannot diverge: compiling the program checks the proof

Proofs as programs help programmers
Lambda calculus in a nutshell
Background

• Very simple, but Turing-complete (Church-Turing thesis)
  ▸ Pure lambda calculus has three constructs and two rewrite rules
  ▸ It's compositional (i.e., it's highly modular)

• Today there are many different flavours
  ▸ With and without types, with many extensions, and so on

• It was introduced by Alonzo Church (in the 1930s)

• Lambda abstractions in C#, C++1x, Objective-C (as blocks), Swift & FP languages
• The pure $\lambda$-calculus has three syntactic forms

  ▸ **Variables:** $x, y, z, \ldots$

  ▸ **Lambda abstraction:** $\lambda x . M$

  ▸ **Function application:** $MN$
Reduction rules

• Alpha conversion:
  • We may change the name of any bound variable.
  • For example, we may convert $\lambda x.x$ to $\lambda y.y$.
  • A bound variable is one named in an enclosing lambda abstraction.
  • For example, here $x$ is bound, but $y$ is free: $\lambda x. xy$.
Reduction rules

- **Beta reduction:**
  - We can reduce a term of the form $(\lambda x. M) N$ to $M[N/x]$
  - Where $M[N/x]$ means to replace any $x$ in $M$ by $N$
  - The latter process is called substitution

- For example, we have

\[(\lambda x. x)y \rightarrow y\]
\[(\lambda x. \lambda y. x)(\lambda z. z)vw \rightarrow (\lambda y. \lambda z. z)vw \rightarrow (\lambda z. z)w \rightarrow w\]