DP

Property-based testing

Program validation

Why it matters …

Go to `fscheck.fsx`
`(up to BACK)`

Installing FsCheck under Linux

- If you use .Net Core SDK, just call
  - #r "nuget: FsCheck";;
- Install nuget with apt
  - Type **sudo nuget update -self ; nuget install fscheck**
  - Copy the dll where dotnet wants it or in your working dir
  - Current version: 2.16.6
- Documentation about FsCheck: https://fscheck.github.io/FsCheck/
- A pretty good blog about PBT with FsCkeck:

  http://fsharpforfunandprofit.com/posts/property-based-testing

Outline of rest of lecture

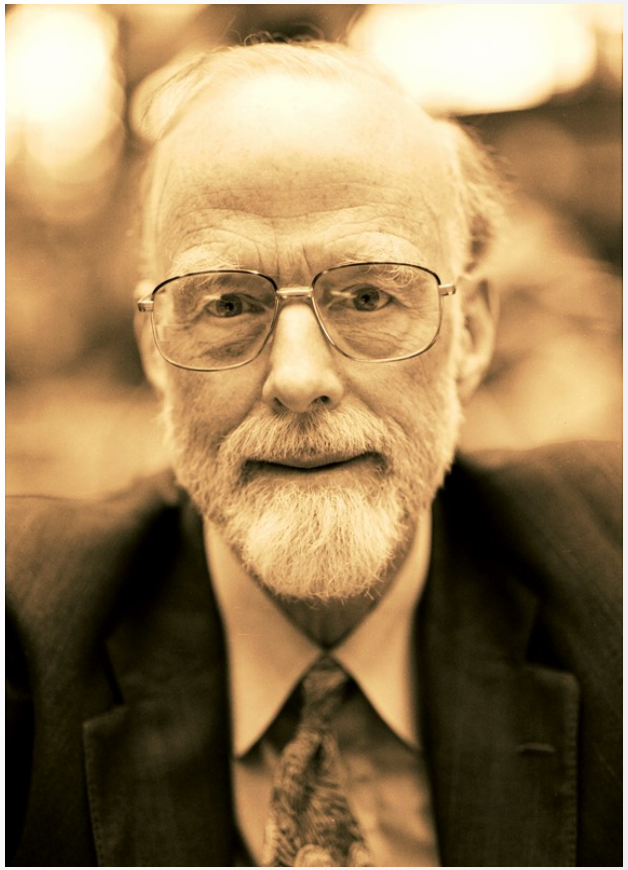
- Background of PBT within formal methods
- Intro to PBT with FsCheck:
  - basic examples
  - conditional properties

Why software validation?

I conclude there are two ways of constructing a software design.

One way is to make it *so simple there are obviously no deficiencies*, and the other way is to make it *so complicated that there are no obvious deficiencies.*

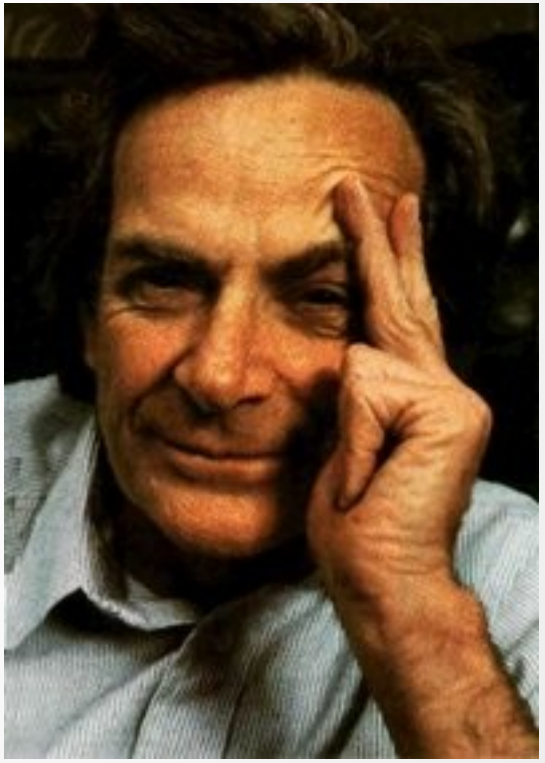
– Tony Hoare [Turing Award Lecture, 1980]



Why automated analysis?

The first principle is that you must not fool yourself, and you are the easiest person to fool.

– Richard P. Feynman



The range of formal methods

- The study of **verification** and/or **validation** of software: from
  - **Lightweight formal methods**: specifying *critical* properties of a system and focus on finding errors *quickly*, rather (or before) than proving **correctness**.
    - "*Spec 'n Check*" up to …
  - **Full correctness:** Specify *all* functional properties of interest of an entire system and perform a *complete proof* of correctness

Software testing

Most common approach to SW quality

- Very labour-intensive
  - up to 50% of SW development
- Even after testing, a bug remains on average per 100 lines of code, costing 60 billions dollars (2002)
- Need of *automatic* testing tools
  - To complete tests in shorter time
  - To test better
  - To repeat tests more easily
  - *To generate test cases automatically*

The dominant paradigm

- By far the most widely used style of testing functionality of pieces of code is ***unit testing***.
  - **Invent** a "state of the world".
  - **Run** the unit (function/method) under test
  - **Check** the modified state of the world to see if it looks like it should

The dominant paradigm

```
public class TestAdder {
    public void testSum() {
        Adder adder = new AdderImpl();
        assert(adder.add(1, 1) == 2);
        assert(adder.add(1, 2) == 3);
        assert(adder.add(2, 2) == 4);
        assert(adder.add(0, 0) == 0);
        assert(adder.add(-1, -2) == -3);
        assert(adder.add(-1, 1) == 0);
        assert(adder.add(1234, 988) == 2222);
    }
}
```

The dominant paradigm

Problem: unit testing is only as good as your *patience:*

→ The previous example contains 7 tests.

- Ericsson's ATM switch controlled by 1.5 mil of code + 700.000 lines of UT
- Typically we lose the will to continue inventing new unit tests long before we've exhausted our *search* of the space of possible bugs.
- (One) **Solution**: randomized testing

Randomized testing

Generating random inputs and feeding them to a function to see whether it behaves correctly.

- **Fuzzing:** feed a string of random characters into a program in the hope to uncover failures.
- **Model-based** testing: If a reference implementation is available, then the outputs of the two implementations can be compared.
- **PBT**: check some property of the output

PBT: Quickcheck

- **Quickcheck** was introduced by Claessen & Hughes (2000) for Haskell
- The programmer provides a specification of the program, in the form of ***properties*** that functions should satisfy
  - Think a **contract**, detailing *pre* and *post-conditions*
- QuickCheck then tests that the properties hold in a large number of ***randomly generated cases***.
- Tests are described by
  - a generator (delivering random input)
  - a property (Boolean-valued function)

Uses of PBT

- **Quickcheck** is now available for most PLs, including imperative ones, such as *Java, C(++), JavaScript, Go, Objective-C, Perl, Erlang, Python, Ruby, Scala …*
- Commercially: QuviQ, Hughes' start-up commercializing Quickcheck for *Erlang*
  - See paper “Quickcheck for fun and profit”

Quickcheck's design decisions

- A lightweight tool – originally 300 lines of Haskell code
- Spec are written via a DSL in the module under test
- Adoption of random testing
- Put distribution of test data in the hand of the user
  - API for writing generators and observe distributions
- Emphasis on *shrinking* failing test cases to facilitate debugging

PBT

Back to code

Quickcheck: how

- Checking $\forall \mathbf{x} : \boldsymbol{\tau}.\ C(\mathbf{x})$ means trying to see if there is an assignment $\mathbf{x} \rightarrow \mathbf{a}$ at type $\boldsymbol{\tau}$ such that $\neg C(\mathbf{a})$ holds
  - e.g. checking $\forall xs :$ `int list.` `rev xs = xs` means finding e.g. xs $\rightarrow$ [1;0], for which rev xs $\neq$ xs
- Quickcheck generates *pseudo-random* values up to size *k* and stops when
  - a counterexample is found, or
  - the maximum number of test values has been reached or
  - a default timeout expires

Connecting pre and post conditions

```
ordered xs ⇒ ordered (insert x xs)
```

- Here we generate random lists that may or may not be sorted and then check if insertion preserves ordered-ness
- If a candidate list does not satisfies the condition, it is discarded
  - *Coverage is an issue:* what's the likelihood of randomly generating lists (of length > 1) that are *sorted*?
- Quickcheck gives combinator to *monitor* test data distribution – but in the end one has to write an ad-hoc generator, here yielding only ordered lists
  - LLM are surpisingly good at writing generators …

What’s next?

- Much more on FsCheck in a later lecture
- If you want to try a small exercise at home, see file exCheck.txt
- And a word of caution:

Dijkstra's ghost

“Program testing can at best show the presence of errors, but never their absence” [*Notes On Structured Programming*, 1970]

“None of the program in this monograph, *needless to say*, has been tested on a machine” [Introduction to *A Discipline of Programming*, 1980]