FP

Lez 6

Property-based testing

Admin

- Primo compitino previsto per **11 Aprile**
- **31 Marzo** si terrà un laboratorio **valutato** a cui **dovete** essere presenti per partecipare al compitino
- Gli esercizi assegnati durante le lezioni possono essere caricati su upload.di.unimi.it e verranno corretti a campione
- Vi possiamo *garantire* che, in generale, chi non fa gli esercizi avrà problemi al compitino.

TESTING

Why it matters …

Go to `fscheck16.fsx`

Installing FsCheck and docs

- Installing FsCheck under Visual Studio
  - Create a console application project on Desktop contrl-N
  - Open Nuget packet manager (Tools >)
  - Type: **Install-Package FsCheck**
  - Copy **FsCheck.dll** deep from the folder to your dir
- Under Linux here
- Documentation about FsCheck
- A pretty good blog about PBT with FsCkeck

Outline of lecture

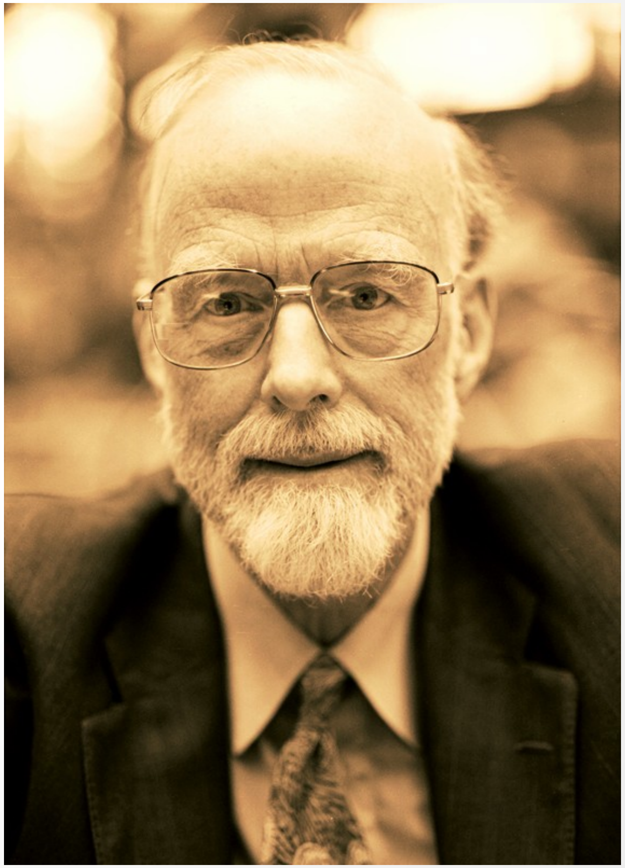
- Background of PBT vs. formal methods
- Intro to PBT with FsCheck:
  - basic examples
  - shrinking
  - model-based testing
  - Conditional properties
    - Lazy annotations
  - weak and strong specifications

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.*

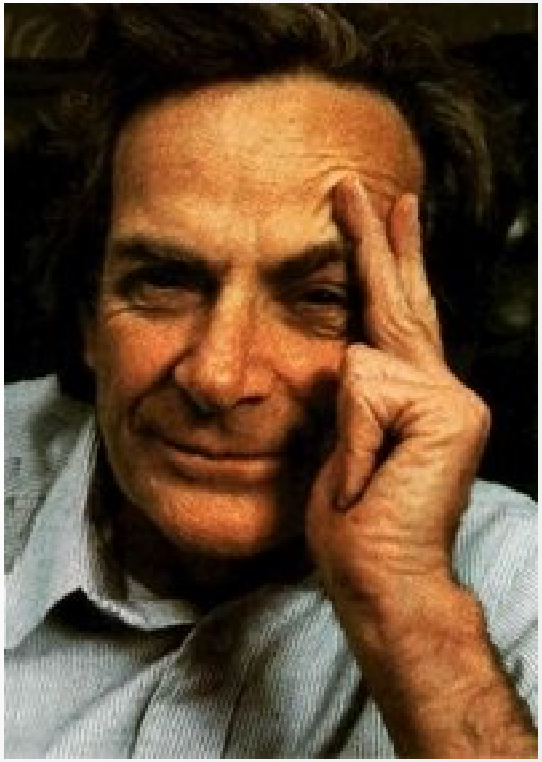
– Tony Hoare [Turing Award Lecture, 1980]



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

- **Lightweight formal methods**: specifying *critical* properties of a system and focus on finding errors *quickly*, rather (or before) than proving correctness.
  - “*Spec ’n Check*” is the mantra

  Up to …

- **Full correctness:** Specify *all* properties of interest of an entire system and perform a *complete proof* of correctness

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]

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 $ (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) we're testing
  - **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**: property-based testing - PBT

PBT: Quickcheck

- **Quickcheck** was introduced by Claessen & Hughes (2000)
- A tool for testing Haskell programs automatically.
- The programmer provides a specification of the program, in the form of ***properties*** that functions should satisfy
- QuickCheck then tests that the properties hold in a large number of ***randomly generated cases***.

PBT

Quickcheck is now available for many PLs, including imperative ones, such as *Java, C(++), JavaScript, Objective-C, Perl, Erlang, Python, Ruby, Scala …*

- **Quickcheck** is based on *random testing*
- There are alteratives such as (**Lazy)Smallcheck**, based on *exhaustive testing* and *symbolic execution*, but just for FP right now
- Now integrated in **proof assistants** such as *Isabelle* and *Coq*

Commercial uses of PBT

- Mostly within QuviQ, Hughes' start-up commercializing Quickcheck for *Erlang*
  - See paper “Quickcheck for fun and profit”
- Some success stories:
  - Ericsson’s 4G radio base stations
  - Database reliability at Basho
  - Mission-critical gateway at Motorola
  - AUTOSAR Basic Software
  - Google’s LevelDB database …

Quickcheck's design decisions

- A lightweight tool – originally 300 lines of Haskell code, then extended to deal with the monadic fragment
- Spec are written via a DSL in the very 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 xs → [1;0], for which rev xs ≠ xs
- Quickcheck generates *pseudo-random* values up to size *k (EndSize)* and stops when
  - a counterexample is found, or
  - the maximum size of test values has been reached (*MaxTest*), or
  - a default timeout expires (*MaxFail*)

Conditional laws

- More interesting are *conditional laws*:
  - `ordered xs` $\Longrightarrow$ `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