

## Reasoning about Programs

Sophia Drossopoulou and Mark Wheelhouse

Department of Computing  
Imperial College London

## With warm thanks to

Samson Abramski, Steve Vickers, Susan Eisenbach, Ian Hodgkinson, Krysia Borda, Francesca Toni, Jeremy Bradley Robert Craven, Alex Summers, Tony Field, Tristan Allwood, Alastair Donaldson, Steffen van Bakel, William Heavens, Peter Müller

and also to

Junqiang Qian, Galina Peycheva, James Prince, Giacomo Guerri, Heiki Riisikamp, Jason Yu, Sebastian Delecta, Alan Dukai, Jinsung Hajin, Ayman, Jaime Rodriguez, Rosita Rodrigues, Oliver Norton, Summer Jones, Jonson Harry Goff-White, Alan Du, Sebastian Delekta, Kabeer Vohra, Harry Roscoe, Jonathan King, Giulio Jiang, Franklin Schrans, Nana Asiedu-Ampem

and also to

Rachel Mekhtieva, Mark Aduolma, Virain Nikhil Gupta, Ryan Chung, Joseph Katsioloudes, Christo Lolov, Jenny Lea, Alberto Spina, Panayiotis Panayiotou, Tarun Sabbineni, Ashley Davies, Shayan Khaksar, Sarah Bakas, Andrei Brabetea, Szymon Zmyslony, Samuel Ogunmola, Ruhi Choudhury, Ashley Davies, Mohammad Zubair Chowdhury, Riku Murair, Max Smith.

and also to

Alexandra Clara Gila, Amar Lakhani, Anson Miu, Ben Dixon, Boris Barath, Daniel, David Buterez, David Kurniadi Angdinata, Ethan LUO Yicheng, Hang Li Li, Ian Yeo, Inara Ramji, Isaac Hutt, Jin Sun Park, Julian Chow, Karam Ali Chaudhry, Lloyd Arthur James Ollerhead, Louis Carteron, Michael Tsang, Mircea-Stefan Mohora, Miroslav Dimitrov, Naman Wahi, Nathaniel Oshunniyi, Niklas Vangerow, Ningzhi Wang, Pablo Gorostiaga, Paul Andrei Gramatovici, Qiang Feng, Remi Kaan Uzel, Subhash Nalluru, Thomas Yung.

## Course Outline

- ① **Part I: Reasoning About Haskell Programs**
  - Mathematical Induction
  - Strong Induction
  - Two more cautionary tales
  - Summary
  - Structural induction over Haskell data types
  - Induction over any recursively defined structures
- ② **Part II: Reasoning about Java Programs**
  - Program Specifications
  - Conditional Branches
  - Method Calls
  - Recursion
  - Iteration
- ③ **Part III: Reasoning Case Studies**
  - Binary Search
  - Dutch Flag Problem

# Introduction and Motivation

Slide 4

Introduction and Motivation

Introduction and Motivation

Drossopoulou & Wheelhouse (DoC) Reasoning about Programs 4 / 393

Slide 5

Introduction and Motivation

Why Reason About Programs?

**Lessons from History:**

- Therac-25
- Patriot Missiles
- Ariane 5
- USS Yorktown
- Zune Screen of Death

**Activity:** which was the worst?

**Possible Solutions:** Testing vs. Proof

Drossopoulou & Wheelhouse (DoC) Reasoning about Programs 5 / 393

## Lessons from History

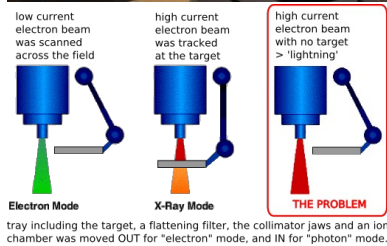
*“To err is human, but to really foul things up you need a computer”*

- Paul R. Ehrlich

We all make mistakes, that is part of being human. However, what I believe Ehrlich is hinting at is that when a programmer makes a mistake, that error can end up being run billions and billions of times. This has the potential to cause some very extreme and undesirable side-effects.

We look at a few of the most famous (embarrassing?) cases over the next few pages.

## Lessons form History: Therac-25



**Year:** 1985

### Summary:

Canada's Therac-25 radiation therapy machine malfunctions delivering lethal doses to patients.

### Cause:

Failure to verify machine configuration before activating radiation emitter.

### Cost:

3 dead, 3 seriously injured.

**System Setup:** Therac-25 was a radiation therapy machine. It inherited legacy code from earlier models (Therac-6 to Therac-20). It had two settings: 1) a low-powered mode for treatment; 2) a high-powered mode for taking x-rays. When run in high-powered mode a lead sheet needed to be put between the patient and the ray emitter to diffuse the radiation.

**What happened:** Due to a system configuration error, 6 people received overdoses of between 15,000 and 20,000 rads. The typical treatment dose should have been between 20-50 rads.


**Reasons for Failure:** There were many errors in the design of the system, how it was tested, the code itself and the user interface. In particular:

- Testing was carried out exclusively at the system level - there was no attempt to test modular components.
- The user interface erroneously reported no/low dosage received without verifying with the actual output.
- The system was very poorly documented (particularly the meaning of error messages).
- The software made no attempt to validate the machine's settings.

Ultimately, this boils down to a failure to validate/check the user-input and system set-up before beginning treatment.

Introduction and Motivation

Lessons from History: Patriot Missile System



Year: 1991

Summary:  
Patriot Missile system in Saudi Arabia fails to intercept an incoming Scud missile, which hits an army barracks.

Cause:  
Software rounding error leads to miscalculation of interception ranges.

Cost:  
28 dead, over 100 injured.

Drossopoulou & Wheelhouse (DoC) Reasoning about Programs 8 / 393

**System Setup:** The Patriot Missile System is a missile defence system that saw widespread use during the 1st Gulf War. The system consisted of an anti-missile missile launcher unit and a paired radar unit.

**What happened:** The system failed to determine a correct intercept course for an incoming missile and the target (a military barracks) was destroyed as a result.

**Reasons for Failure:** There was a small error in the internal clock representation used in calculating the range to a target. This error was on the order of  $0.000000095s$  (or  $9.5 \times 10^{-8}s$  or 95 nanoseconds), but this code was used every 1/10th of a second. After 100 hours of operation, the error had build up to 0.34s, in which time a Scud missile travels 1/2km. This meant that by the time the system had worked out it could intercept the missile, it had already hit its target.

Ultimately, this boils down to a failure to follow the system documentation (specification), which called for a system reboot roughly every 30 hours.

[United States General Accounting Office, <http://fas.org/spp/starwars/gao/im92026.htm>, February 1992]

## Lessons from History: Ariane 5



**Year:** 1996

**Summary:**

Flagship unmanned rocket explodes shortly after launch on its maiden flight.

**Cause:**

64-bit number copied into 16-bit buffer. Resulting overflow crashes guidance system and backup system (running the same software).

**Cost:**

\$500 million + 4 scientific satellites

**System Setup:** Ariane 5 was the European Space Agency's pioneering unmanned rocket.


**What happened:** Shortly after take-off on its maiden flight, the rocket exploded destroying both itself and all of its payload.

**Reasons for Failure:** In the internals of the guidance computer a variable tracking the sideways velocity of the rocket was moved from a 64-bit int to a 16-bit int. The resulting overflow crashed the whole guidance system. The rocket switched to a back-up guidance system, that was running exactly the same code. With no functioning guidance system, the rocket veered off course and the rocket's safety system then triggered an emergency self-destruct. To add to the embarrassment, the software which contained the overflow wasn't actually needed during flight and could have been disabled during take-off.

Ultimately, this boils down to a failure to type-check the guidance system code (16-bit and 64-bit ints are not the same type).

Introduction and Motivation

## Lessons from History: USS Yorktown



Year: 1997

Summary:

US Aegis missile cruiser shuts down for nearly 3hrs during manoeuvres, has to be towed back to port.

Cause:

Crew member enters 0 into a database field causing “divide-by-zero” error crashing all machines on the network.

Cost:

\$1 billion ship put at risk, major software upgrade scrapped.

Drossopoulou & Wheelhouse (DoC)
Reasoning about Programs
10 / 393

**System Setup:** The U.S. Navy’s Aegis Missile Cruisers are high-tech ships that specialise in missile interception. They are a significant part of the U.S. Military’s first-line of defence against missile attack (conventional or nuclear).

**What happened:** One of the crew incorrectly entered a 0 into a text-box in the navigation system and the entire ship’s computer system crashed. Reports leaked by the crew state that the ship was “dead in the water” for almost 3 hours and had to be towed back to port. Interestingly, the official U.S. Navy report on the incident claims that the outage was only short (of the order of a few minutes) and that the ship was able to get back to port under its own power. Regardless, a planned multi-million \$ software upgrade to the Aegis missile cruiser fleet was scrapped.

**Reasons for Failure:** The erroneous data entry into the navigation system triggered a “divide-by-zero” error in the software which crashed the entire ship’s computer system. Reports leaked from the crew claim that the new software upgrade was “full of bugs” and that controls on the ship “regularly blue-screened”. This suggests that the new software had not been very extensively tested, particularly on badly-formed user inputs.

Ultimately, this boils down to a failure to validate/check user-input.

[Gregory Slabodkin, GCN, <https://gcn.com/Articles/1998/07/13/Software-glitches-leave-Navy-Smart-Ship-dead-in-the-water.aspx>, July 1998]



## Lessons from History: Zune Screen of Death



**Year:** 2008

**Summary:**

At midnight on December 31 every Zune 30 player (and some Toshiba media players) froze. Event dubbed the Zunepocalypse.

**Cause:**

An error in date calculation leads to an infinite loop in the code, causing the player to freeze indefinitely.

**Cost:**

Massive hit to consumer confidence.

**System Setup:** Zune was a brand of digital media products and services marketed by Microsoft. Zune 30 was an early portable media device competing against the first i-Pods.

**What happened:** At midnight on 31st December 2008 every Zune 30 player in the world (and some Toshiba media players) froze. The official solution published by Microsoft was to completely power down the device and wait until 1st January 2009.

**Reasons for Failure:** Part of the date calculation in Zune 30 software used the following code to calculate the current year from the number of days since 1st January 1980:

```

1  year = 1980;
2  while (days > 365) {
3      if (IsLeapYear(year)) {
4          if (days > 366) {
5              days -= 366;
6              year += 1;
7          }
8      } else {
9          days -= 365;
10         year += 1;
11     }
12 }
```

This code enters an infinite loop when the current year is a leap year (such as 2008) and the number of days remaining is exactly 366 (as it will be on 31st Dec. in such a year).

Ultimately, this boils down to a failure to fully consider the termination conditions of the loops in the code.

## Activity - Which Disaster was the Worst?

Think about the five computing disasters we have discussed so far.

**Without talking to anyone else** please vote (via Mentimeter) on which **you** think was the worst disaster.

Now, with the people sitting around you, discuss and order these disasters from what you collectively consider to be the least to worst cases.

You will have 5 - 10 minutes to complete this task and then we will vote again.

Be prepared to give a brief explanation of why you chose to order the disasters in the way that you did.

## Have we Learned the Lesson?

Have you heard of...



- Heartbleed (2011-2014)
- TimSort bug (Feb 2015)
- Windows 9 (July 2015)

For the interested, here are a few links to some further reading on the more recent

Computing Disasters discussed above.

### The Heartbleed Bug:

In a nutshell, an OpenSSL vulnerability allows users to ask a server for more data than they should be able to read. This results in leaking passwords, secret keys and other confidential information.

[<http://heartbleed.com/>]

### The TimSort algorithm and bug:

In a nutshell, an efficient Java.util sorting algorithm (widely used on Java Android platforms) can sometimes throw an Array-Out-Of-Bounds exception.

[<http://www.envisage-project.eu/proving-android-java-and-python-sorting-algorithm-is-broken-and-how-to-fix-it/>]

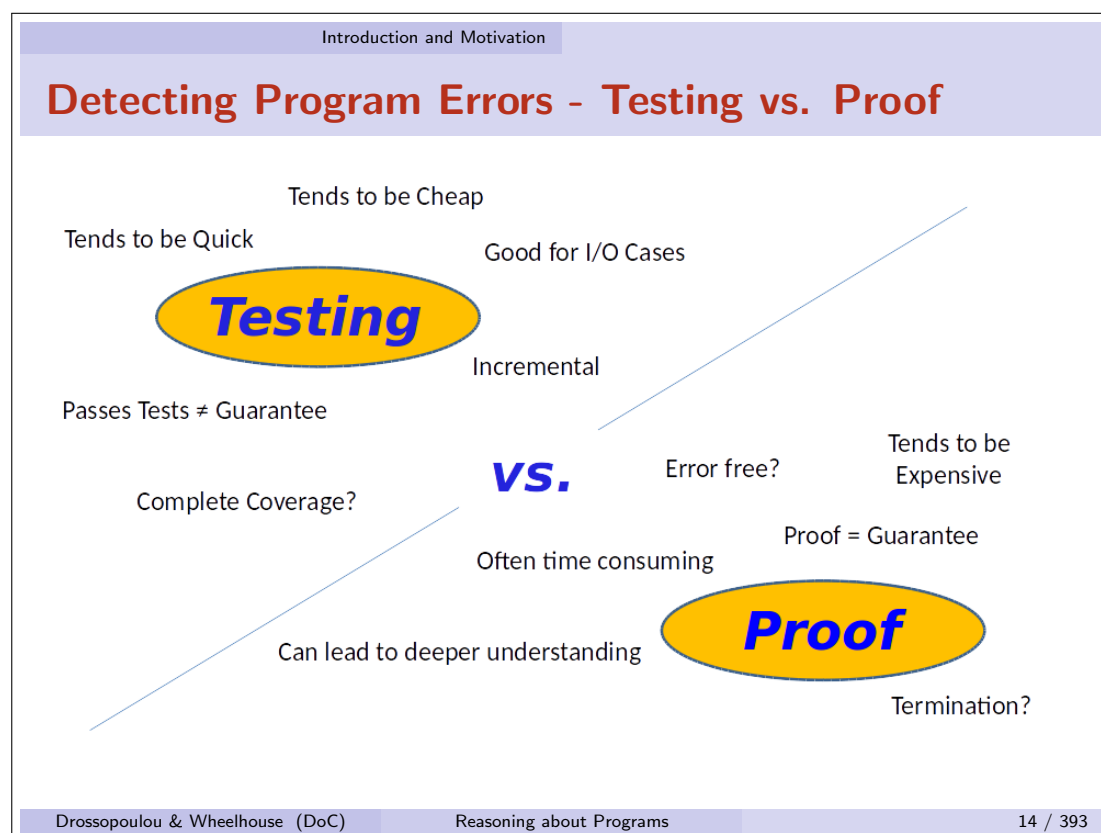
### Why Windows 10 and not 9?:

In a nutshell, empirical evidence suggests that a large amount of 3rd party Windows software/libraries begins with a test like `if(version.startswith("windows 9"))` to check if the Operating System is Windows 95 or 98. Of course, this would pick up Windows 9 as well, leading to buggy behaviour.

[<http://www.extremetech.com/computing/191279-why-is-it-called-windows-10-not-windows-9>]

[<https://searchcode.com/?q=if%28version%2Cstartswith%28%22windows+9%22%29>]

Slide 14



Introduction and Motivation

**In this course we focus on proofs**

Drossopoulou & Wheelhouse (DoC) Reasoning about Programs 15 / 393

Note that your programming courses and later year courses in Software Engineering will cover testing in depth. This includes principles such as Unit Testing, Integration Testing and Systems Testing.

A good software project should make use of *both* testing and reasoning techniques to ensure you have a robust and reliable solution.

# Course Overview

Slide 16

Course Overview

Course Overview

Drossopoulou & Wheelhouse (DoC) Reasoning about Programs 16 / 393

Slide 17

Course Overview

Course Aims and Rationale

- Develop a better understanding of programming
- Prove certainty of a program's behaviour
- Investigate formal program documentation
- Program verification/analysis tools are becoming ever more powerful/pervasive

**In this course we...**

- write program specifications
- use mathematical techniques to verify the correctness of functions, methods and loops

We do not consider issues to do with aliasing, concurrency or the verification of large programs. Many of these topics will be covered in courses in later years.

Drossopoulou & Wheelhouse (DoC) Reasoning about Programs 17 / 393

We consider that this course, along with others on the *Fundamental Computer Science Theory* slide, to be what separate the “*Computer Scientist*” from the “*Hacker*”. We are teaching you to take a more principled and considered approach to programming, rather than taking a random walk through the run-debug-run cycle.

Course Overview

## Course Structure

**Introduction and Motivation**

**Part I - Reasoning About Haskell Programs**

- Function specifications
- Proof by Induction

**Part II - Reasoning About Java Programs**

- Method Definitions and Calls
- Loops, Variants and Invariants

**Part III - Reasoning Case Studies**

Drossopoulou & Wheelhouse (DoC)
Reasoning about Programs
18 / 393

There is some overlap between Part I of this course and the end of the Discrete Structures course. This is somewhat inevitable due to an ongoing reorganisation of the taught material in year 1 and also because JMC students do not take the Discrete Structures course.

However, in this course we will cover Proof by Induction from a new perspective - starting from first principles.

- We will show how a recursive definition implicitly introduces an inductive principle.
- We will show how the inductive principle introduces a proof schema.
- We will show how the proof schema can be used to prove a property of an inductively defined set (or relation, or function).

By revisiting the topic of Induction we allow any students who might not have followed all of the last part of the Discrete Structures course a chance to catch up. We will also be covering some new, more subtle, aspects of Induction for those students who are already confident with the topic from Discrete Structures.

In particular, we will pay extra attention to:

- the correct use of quantifiers;
- when the Induction Hypothesis is applicable;
- the discovery of auxiliary lemmas to help with proofs;
- and cases where we need to strengthen properties in order to prove weaker ones.

Slide 19

Course Overview

## Fundamental Computer Science Theory

Many of the techniques you learn in this course will be revisited in:

- Models of Computation
- Models of Concurrent Computation
- Type Systems for Programming Languages
- Advanced Issues in Object Oriented Programming
- Program Analysis
- Software Reliability
- Separation Logic

Drossopoulou & Wheelhouse (DoC) Reasoning about Programs 19 / 393

## Program Verification - A Brief History

- 1969 Tony Hoare: An Axiomatic Semantics for Computer Programs [Communications of the ACM]
- 1970s Axiomatic Definition for Pascal
- 1980s Program Verification for concurrent programs
- 2000s Program Verification for the heap/aliasing

There now exist many tools that support program verification.

Program Verification is an active research area at many top institutions:

Microsoft Research, Imperial College, Cambridge University, Oxford University, Chalmers University, Koblenz University, Max Planck Institute, Queen Mary, Westfield College, and many more...

Program Verification is now seeing active use at some top companies:

Facebook, Amazon and Microsoft (amongst others)

There is also a Program Verification Grand Challenge.

Some recent noteworthy mentions/projects involving program verification:

- **Ironclad:** [<https://www.microsoft.com/en-us/research/project/ironclad/>]  
Making use of the Dafny verification tool to build a secure compiler for app execution.
- **BlueScreens:** [<http://www.zdnet.com/article/why-the-blue-screen-of-death-no-longer>]  
Microsoft has used a number of techniques, including automated program verification, to drastically cut down on the frequency of “Blue Screens of Death”.



Course Overview

## Course Logistics

**Course Timetable:**  
 Tuesdays: 09:00 - 10:00 (weeks 2 - 10)  
 Thursdays: 09:00 - 11:00 (weeks 2 - 10)

**Integrated Lectures/Tutorials:**  
 Bring a pen, paper and notes with you and attempt all exercises!

**Assessed by Final Exam:**  
**When:** Summer Term (April/May)  
**Format:** 2 Questions covering whole course  
**Length:** 1hr 15mins

Drossopoulou & Wheelhouse (DoC) Reasoning about Programs 21 / 393

Course Overview

## Additional Course Support

**Enhanced Notes:**  
 Summary handouts posted at the start of section.  
 Full notes posted at the end of each section.  
 You should make your own notes during lectures.

**Tutorial Sheets:**  
 One tutorial sheet per week (*working through key parts in class*)  
 One PMT sheet per week (*to discuss during PMT sessions*)  
 Assessed PMT exercises in weeks 4, 5, 7 and 8  
*(marked by PMT-UTA: formative not summative)*

**Piazza Discussion Forum:**  
<https://piazza.com/imperial.ac.uk/spring2018/141>

Drossopoulou & Wheelhouse (DoC) Reasoning about Programs 22 / 393

# What is a Proof?

Slide 23

What is a Proof?

What is a Proof?

Drossopoulou & Wheelhouse (DoC) Reasoning about Programs 23 / 393

Slide 24

What is a Proof?

**Sample Problem - Supervillains**


Assume the following facts are true:

- (1) A person is happy if all of their children are rich.
- (2) Someone is a supervillain if at least one of their parents is a supervillain.
- (3) All supervillains are rich.

Show that:

All supervillains are happy.

(state any additional assumptions that you make)



Drossopoulou & Wheelhouse (DoC) Reasoning about Programs 24 / 393

## What is a Proof? - Written in English?

Consider the following argument:

*All of a supervillain's children must also be supervillains.  
Now, all supervillains are rich, which means that all of a  
supervillain's children must be rich.  
Therefore any supervillain is happy.*

- Can you see any issues with this argument?
- Is it convincing?

## Supervillains - Flaw in My Argument?



**supervillain(Gru)**  
**rich(Gru)**  
**¬happy(Gru)**

is Gru a person?



**supervillain(Edith)**  
**rich(Edith)**



**supervillain(Margo)**  
**rich(Margo)**



**supervillain(Agnes)**  
**rich(Agnes)**

## Supervillains - The Missing Assumption

Assume the following facts about supervillains are true:

- (1) A person is happy if all of their children are rich.
- (2) Someone is a supervillain if at least one of their parents is a supervillain.
- (3) All supervillains are rich.
- (4) A supervillain is also a person.

Show that:

All supervillains are happy.

Note that another solution to the problem would be to generalise statement (1) to

“*Someone* is happy if all of their children are rich”.

However, this does slightly contradict the assumption that all of the given facts are supposed to be true.

## Supervillains - Formalise the Argument

**Given:**

$$(1) \forall x. [ \text{person}(x) \wedge \forall y. [ \text{childof}(x, y) \longrightarrow \text{rich}(y) ] \longrightarrow \text{happy}(x) ]$$

$$(2) \forall x. [ \exists y. [ \text{childof}(y, x) \wedge \text{supervillain}(y) ] \longrightarrow \text{supervillain}(x) ]$$

$$(3) \forall x. [ \text{supervillain}(x) \longrightarrow \text{rich}(x) ]$$

$$(4) \forall x. [ \text{supervillain}(x) \longrightarrow \text{person}(x) ]$$

**To show:**

$$(\alpha) \forall x. [ \text{supervillain}(x) \longrightarrow \text{happy}(x) ]$$

## Aside - Binding Convention Reminder

(strongest)  $\forall x, \exists x, \neg, \wedge, \vee, \longrightarrow, \longleftrightarrow$  (weakest)

e.g.

$$\forall x. P(x) \wedge Q(x) \longrightarrow R(x) \vee S(x) \equiv ((\forall x. P(x)) \wedge Q(x)) \longrightarrow (R(x) \vee S(x))$$

We usually use [ and ] to explicitly delimit the bounds of quantifiers.

e.g.

$$\forall x. [ P(x) \wedge Q(x) \longrightarrow R(x) \vee S(x) ] \equiv \forall x. [ (P(x) \wedge Q(x)) \longrightarrow (R(x) \vee S(x)) ]$$

## What is a Proof? - Natural Deduction?

(1) $\forall x. [ \text{person}(x) \wedge \forall y. [ \text{childof}(x, y) \rightarrow \text{rich}(y) ] \rightarrow \text{happy}(x) ]$	(given)
(2) $\forall x. [ \exists y. [ \text{childof}(y, x) \wedge \text{supervillain}(y) ] \rightarrow \text{supervillain}(x) ]$	(given)
(3) $\forall x. [ \text{supervillain}(x) \rightarrow \text{rich}(x) ]$	(given)
(4) $\forall x. [ \text{supervillain}(x) \rightarrow \text{person}(x) ]$	(given)
(5) <i>Gru</i>	$\forall I$ const
(6) <i>supervillain(Gru)</i>	(ass)
(7) <i>person(Gru)</i>	$\forall \rightarrow E(6, 4)$
(8) <i>Agnes</i>	$\forall I$ const
(9) <i>childof(Gru, Agnes)</i>	(ass)
(10) <i>childof(Gru, Agnes) <math>\wedge</math> supervillain(Gru)</i>	$\wedge I(9, 6)$
(11) $\exists y. [ \text{childof}(y, \text{Agnes}) \wedge \text{supervillain}(y) ]$	$\exists I(10)$
(12) <i>supervillain(Agnes)</i>	$\forall \rightarrow E(2, 11)$
(13) <i>rich(Agnes)</i>	$\forall \rightarrow E(3, 12)$
(14) <i>childof(Gru, Agnes) <math>\rightarrow</math> rich(Agnes)</i>	$\rightarrow I(9, 13)$
(15) $\forall y. [ \text{childof}(Gru, y) \rightarrow \text{rich}(y) ]$	$\forall I(8, 14)$
(16) <i>person(Gru) <math>\wedge</math> <math>\forall y. [ \text{childof}(Gru, y) \rightarrow \text{rich}(y) ]</math></i>	$\wedge I(7, 15)$
(17) <i>happy(Gru)</i>	$\forall \rightarrow E(1, 16)$
(18) <i>supervillain(Gru) <math>\rightarrow</math> happy(Gru)</i>	$\rightarrow I(6, 17)$
(19) $\forall x. [ \text{supervillain}(x) \rightarrow \text{happy}(x) ]$	$\forall I(5, 18)$

## Stylised Proof - Happy Supervillains

### Given:

- (1)  $\forall x. [ \text{person}(x) \wedge \forall y. [ \text{childof}(x, y) \rightarrow \text{rich}(y) ] \rightarrow \text{happy}(x) ]$
- (2)  $\forall x. [ \exists y. [ \text{childof}(y, x) \wedge \text{supervillain}(y) ] \rightarrow \text{supervillain}(x) ]$
- (3)  $\forall x. [ \text{supervillain}(x) \rightarrow \text{rich}(x) ]$
- (4)  $\forall x. [ \text{supervillain}(x) \rightarrow \text{person}(x) ]$

### To show:

- $$(\alpha) \forall x. [ \text{supervillain}(x) \rightarrow \text{happy}(x) ]$$

### Proof:

Take *Gru* arbitrary,

(ass1) *supervillain(Gru)*

(5) *person(Gru)  $\wedge$   $\forall y. [ \text{childof}(Gru, y) \rightarrow \text{rich}(y) ] \rightarrow \text{happy}(Gru)$*  from (1)

(6) *person(Gru)* from (ass1) and (4)

Take *Edith* arbitrary,

(ass2) *childof(Gru, Edith)*

(7) *supervillain(Edith)* from (ass2), (ass1) and (2)

(8) *rich(Edith)* from (7) and (3)

(9)  $\forall y. [ \text{childof}(Gru, y) \rightarrow \text{rich}(y) ]$  from (ass2), (8) and *Edith* arbitrary

(10) *happy(Gru)* from (5), (6) and (9)

Thus  $(\alpha)$  holds from (ass1), (10) and *Gru* arbitrary.

## Comparing the Proof Styles

### (1) Free-Form Proofs:

- (+) short to develop
- (+) might highlight the intuition
- (−) error prone

### (2) Natural Deduction Proofs:

- (+) total confidence in the proof
- (−) very lengthy
- (−) layout sometimes (often?) difficult
- (−) intuition may be lost in the detail

### (3) Stylised Proofs:

- (+) structure of argument made explicit
- (+) few errors
- (−) errors *are* still possible

## Aims of a Proof

We want to develop proofs which:

- prove only valid statements
- we can easily read (and check) on the day we write them
- we can easily read (and check) several days after we write them
- others can easily read and check the proof years after we wrote them
- (ideally) highlight the intuitions behind the argument

What is a Proof?

## Aims of a Proof

Aim	Free-Form	Natural Deduction	Stylised
prove only valid	X	✓	≈
easily read at time	X	≈	✓
easily read later	X	≈	✓
others can check	X	✓	✓
highlight intuitions	✓	X	≈

From this it should be quite clear why we choose to work with stylised proofs in this course

Drossopoulou & Wheelhouse (DoC) Reasoning about Programs 34 / 393

What is a Proof?

## Stylised Proofs

- Rule 1.** write out and name each given formula
- Rule 2.** write out and name each formula to be shown
- Rule 3.** plan out the proof and name intermediate results
- Rule 4.** justify each step of the proof

Drossopoulou & Wheelhouse (DoC) Reasoning about Programs 35 / 393

As your confidence and experience grow, individual proof steps may become more complex. You will also find that you start to plan the proof as you are writing it.



## Stylised Proofs - Making the Plan

In order to prove  $P$ , we can use any of the following tactics:

- (PC) Contradiction i.e. show that  $\neg P \longrightarrow \text{false}$
- ( $\wedge$ I) If  $P = Q \wedge R$  then we must show both  $Q$  and  $R$
- ( $\vee$ I) If  $P = Q \vee R$  then we must show either  $Q$  or  $R$
- ( $\rightarrow$ I) If  $P = Q \longrightarrow R$  then we can assume  $Q$  and must show  $R$
- ( $\neg$ I) If  $P = \neg Q$  then we can show that  $Q \longrightarrow \text{false}$
- ( $\forall$ I) If  $P = \forall x. Q(x)$  then we can take arbitrary  $c$  and show  $Q(c)$
- ( $\exists$ I) If  $P = \exists x. Q(x)$  then we must find some  $c$  and show  $Q(c)$

[NB: in this context  $=$  stands for “has the form”]

These tactics should be familiar to you from Natural Deduction.

## Stylised Proofs - Justifying the Steps

We justify the steps in our proofs with the following rules:

- ( $\perp$ E) If *false* holds, then we know that  $P$  holds (for *any*  $P$ )
- ( $\wedge$ E) If  $P \wedge Q$  holds, then we know that  $P$  and  $Q$  both hold
- ( $\vee$ E) If  $P \vee Q$  holds, then we can do case analysis assuming each in turn
- ( $\rightarrow$ E) If  $P \longrightarrow Q$  holds and  $P$  holds, then we know that  $Q$  also holds
- ( $\forall$ E) If  $\forall x. P(x)$  holds, then we know that  $P(c)$  holds for *any*  $c$
- ( $\exists$ E) If  $\exists x. P(x)$  holds, then we know that  $P(c)$  holds for *some*  $c$
- (LEM) We can apply any lemma we are given or have already proven  
e.g.  $\exists x. \neg P(x) \longleftrightarrow \neg \forall x. P(x)$
- (EQV) We can use logical equivalences to rewrite formulae  
e.g.  $\forall x. [ \exists y. [ P(x, y) ] \longrightarrow Q(x) ] \equiv \forall x. \forall y. [ P(x, y) \longrightarrow Q(x) ]$

...as should these justifications.

## Another Example - Happy Dragons

Assume the following facts about dragons are true:

- (1) A dragon is happy if all of its children can fly
- (2) All green dragons can fly
- (3) Something is green if at least one of its parents is green
- (4) All of the children of a dragon are also dragons
- (5) If  $y$  is a child of  $x$ , then  $x$  is a parent of  $y$

Show that:

All green dragons are happy



## Happy Dragons - Formalise the Argument

**Given:**

- (1)  $\forall x. [ \text{dragon}(x) \wedge \forall y. [ \text{childof}(x, y) \longrightarrow \text{fly}(y) ] \longrightarrow \text{happy}(x) ]$
- (2)  $\forall x. [ \text{green}(x) \wedge \text{dragon}(x) \longrightarrow \text{fly}(x) ]$
- (3)  $\forall x. [ \exists y. [ \text{parentof}(x, y) \wedge \text{green}(y) ] \longrightarrow \text{green}(x) ]$
- (4)  $\forall x. [ \forall y. [ \text{childof}(x, y) \wedge \text{dragon}(x) \longrightarrow \text{dragon}(y) ] ]$
- (5)  $\forall x. [ \forall y. [ \text{childof}(x, y) \longrightarrow \text{parentof}(y, x) ] ]$

**To show:**

$$(\alpha) \forall x. [ \text{dragon}(x) \longrightarrow \text{green}(x) \longrightarrow \text{happy}(x) ]$$

## Happy Dragons - The Natural Deduction Proof

(1) $\forall x. [ \text{dragon}(x) \wedge \forall y. [ \text{childof}(x, y) \longrightarrow \text{fly}(y) ] \longrightarrow \text{happy}(x) ]$	(given)
(2) $\forall x. [ \text{green}(x) \wedge \text{dragon}(x) \longrightarrow \text{fly}(x) ]$	(given)
(3) $\forall x. [ \exists y. [ \text{parentof}(x, y) \wedge \text{green}(y) ] \longrightarrow \text{green}(x) ]$	(given)
(4) $\forall x. [ \forall y. [ \text{childof}(x, y) \wedge \text{dragon}(x) \longrightarrow \text{dragon}(y) ] ]$	(given)
(5) $\forall x. [ \forall y. [ \text{childof}(x, y) \longrightarrow \text{parentof}(y, x) ] ]$	(given)
<b>(6) Smaug</b>	
(7) $\text{dragon}(\text{Smaug})$	( $\forall I$ const)
(8) $\text{green}(\text{Smaug})^\dagger$	(ass)
(9) <b>Dragon</b>	( $\forall I$ const)
(10) $\text{childof}(\text{Smaug}, \text{Dragon})$	(ass)
(11) $\text{parentof}(\text{Dragon}, \text{Smaug})$	$\forall \rightarrow E(10, 5)$
(12) $\text{parentof}(\text{Dragon}, \text{Smaug}) \wedge \text{green}(\text{Smaug})$	$\wedge I(11, 8)$
(13) $\exists y. [ \text{parentof}(\text{Dragon}, y) \wedge \text{green}(y) ]$	$\exists I(12)$
(14) $\text{green}(\text{Dragon})$	$\forall \rightarrow E(13, 3)$
(15) $\text{childof}(\text{Smaug}, \text{Dragon}) \wedge \text{dragon}(\text{Smaug})$	$\wedge I(10, 7)$
(16) $\text{dragon}(\text{Dragon})$	$\forall \rightarrow E(15, 4)$
(17) $\text{green}(\text{Dragon}) \wedge \text{dragon}(\text{Dragon})$	$\wedge I(14, 16)$
(18) $\text{fly}(\text{Dragon})$	$\forall \rightarrow E(17, 2)$
(19) $\text{childof}(\text{Smaug}, \text{Dragon}) \longrightarrow \text{fly}(\text{Dragon})$	$\rightarrow I(10, 18)$
(20) $\forall y. [ \text{childof}(\text{Smaug}, y) \longrightarrow \text{fly}(y) ]$	$\rightarrow I(9, 19)$
(21) $\text{dragon}(\text{Smaug}) \wedge \forall y. [ \text{childof}(\text{Smaug}, y) \longrightarrow \text{fly}(y) ]$	$\wedge I(7, 20)$
(22) $\text{happy}(\text{Smaug})$	$\forall \rightarrow E(21, 1)$
(23) $\text{green}(\text{Smaug}) \longrightarrow \text{happy}(\text{Smaug})$	$\rightarrow I(8, 22)$
(24) $\text{dragon}(\text{Smaug}) \longrightarrow \text{green}(\text{Smaug}) \longrightarrow \text{happy}(\text{Smaug})$	$\rightarrow I(7, 23)$
(25) $\forall x. [ \text{dragon}(x) \longrightarrow \text{green}(x) \longrightarrow \text{happy}(x) ]$	$\forall I(6, 24)$

## Happy Dragons - Stylised Proof

**Given:**

- |  |         |
|--|---------|
| (1) $\forall x. [ \text{dragon}(x) \wedge \forall y. [ \text{childof}(x, y) \longrightarrow \text{fly}(y) ] \longrightarrow \text{happy}(x) ]$ | (given) |
| (2) $\forall x. [ \text{green}(x) \wedge \text{dragon}(x) \longrightarrow \text{fly}(x) ]$   | (given) |
| (3) $\forall x. [ \exists y. [ \text{parentof}(x, y) \wedge \text{green}(y) ] \longrightarrow \text{green}(x) ]$                               | (given) |
| (4) $\forall x. [ \forall y. [ \text{childof}(x, y) \wedge \text{dragon}(x) \longrightarrow \text{dragon}(y) ] ]$                              | (given) |
| (5) $\forall x. [ \forall y. [ \text{childof}(x, y) \longrightarrow \text{parentof}(y, x) ] ]$   | (given) |

**To show:**

- ( $\alpha$ )  $\forall x. [ \text{dragon}(x) \longrightarrow \text{green}(x) \longrightarrow \text{happy}(x) ]$

**Proof:**

take an arbitrary dragon *Smaug*

(ass1)  $\text{dragon}(\text{Smaug})$

(ass2)  $\text{green}(\text{Smaug})^\dagger$

need to show  $\text{happy}(\text{Smaug})$

(6)  $\forall x. \forall y. [ \text{parentof}(x, y) \wedge \text{green}(y) \longrightarrow \text{green}(x) ] ]$

follows from (3)

(7)  $\forall x. \forall y. [ \text{childof}(y, x) \wedge \text{green}(y) \longrightarrow \text{green}(x) ] ]$

follows from (6) and (5)

(8)  $\forall x. [ \text{childof}(\text{Smaug}, x) \longrightarrow \text{green}(x) ]$

follows from (7) and (ass2)

(9)  $\forall x. [ \text{childof}(\text{Smaug}, x) \longrightarrow \text{dragon}(x) ]$

follows from (4) and (ass1)

(10)  $\forall x. [ \text{childof}(\text{Smaug}, x) \longrightarrow \text{green}(x) \wedge \text{dragon}(x) ]$

follows from (8) and (9)

(11)  $\forall x. [ \text{childof}(\text{Smaug}, x) \longrightarrow \text{fly}(x) ]$

follows from (10) and (2)

(12)  $\text{happy}(\text{Smaug})$

follows from (1), (ass1) and (11)

Thus ( $\alpha$ ) holds

follows from (ass1), (ass2), (12) and *Smaug* arbitrary

$\dagger$  Note that *Smaug* isn't actually a green dragon. Our proofs are just showing that if he were green, then he would be happy. Maybe then he wouldn't feel the need to go around burning cities to the ground and stealing large piles of gold.

There are two steps in this proof that warrant a little more discussion, namely steps (6) and (10). These logical equivalences might not been immediately obvious, so let's prove them.

**Justifying (6):** This step relies on the following logical entailment:

$$\forall x.[\exists y.[P(x, y)] \longrightarrow Q(x)] \longrightarrow \forall x.\forall y.[P(x, y) \longrightarrow Q(x)]$$

We prove this as follows:

**Given:**

$$(1) \forall x.[\exists y.[P(x, y)] \longrightarrow Q(x)] \quad \text{(given)}$$

**To show:**

$$(\alpha) \forall x.\forall y.[P(x, y) \longrightarrow Q(x)]$$

**Proof:**

take arbitrary  $c_1$  and  $c_2$

need to show  $P(c_1, c_2) \longrightarrow Q(c_1)$

(ass)  $P(c_1, c_2)$

need to show  $Q(c_1)$

(2)  $\exists y.[P(c_1, y) \longrightarrow Q(c_1)]$  follows from (1) letting  $x$  be  $c_1$

(3)  $\exists y.[P(c_1, y)]$  follows from (ass) replacing  $c_2$  with  $y$

(4)  $Q(c_1)$  follows from (2) and (3)

Thus  $(\alpha)$  holds

**Justifying (10):** This step relies on the following logical entailment:

$$(A \longrightarrow B) \wedge (A \longrightarrow C) \longrightarrow (A \longrightarrow B \wedge C)$$

We prove this as follows:

**Given:**

$$(1) A \longrightarrow B \quad \text{(given)}$$

$$(2) A \longrightarrow C \quad \text{(given)}$$

**To show:**

$$(\alpha) A \longrightarrow B \wedge C$$

**Proof:**

(ass)  $A$

need to show  $B \wedge C$

(3)  $B$  follows from (1) and (ass)

(4)  $C$  follows from (2) and (ass)

(5)  $B \wedge C$  follows from (3) and (4)

Thus  $(\alpha)$  holds

(Note: the reverse entailment can be shown in a similar fashion.)