Formal Methods and Functional Programming

Week 1

February 21

General information: Exercise Sheets

- one exercise session per week
- one exercise sheet per week - due Sunday by 11:59 PM
  - All assignments (code and text) need to be submitted via *codeboard*.
  - Register for your exercise groups at: codeboard.ethz.ch/fmfp17_register/
  - Make sure you are logged-in on codeboard.io before submitting if you want to save your solution.
  - Please compile before you run the program.
  - Feedback on your solution given by tutors can be viewed on *codeboard*.

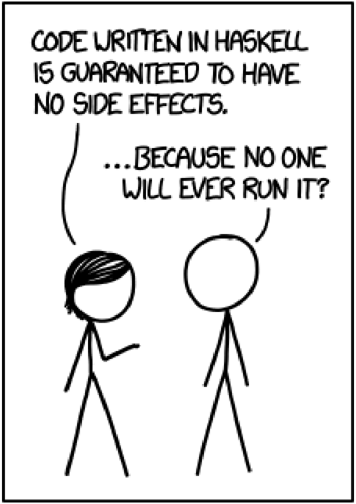
Structure of the exercise session:

- feedback on last exercise sheet hand-ins
  ask questions if solutions should be explained
- review and practice new material from lecture
  bring your laptop/tablet/...
- give *preview* information about new exercise sheet

Haskell introduction

- installation
- pick text editor of choice
  Setting up Haskell-mode for emacs: `www.inf.ed.ac.uk/teaching/courses/inf1/fp/emacs.pdf`
- offline workflow demonstrated:
  1. write/modify haskell source in text file
  2. load in ghci
  3. test your function definitions
  4. repeat from 1
- debugging: typecheck + runtime
  - mistakes demo
- web-based IDE: codeboard.ethz.ch (demostrated)

Haskell introduction



Input and Output

- How would we write a the following program in Haskell?

```
void f(String out) {
    String inp1 = Console.readLine();
    String inp2 = Console.readLine();
    if (inp2.equals(inp1)) System.out.println(out); }
```

- Assume there would be functions

```
getLine :: String      putStrLn :: String -> ()
```

`()` is the unit type

Input and Output

- How would we write a the following program in Haskell?

```
void f(String out) {
    String inp1 = Console.readLine();
    String inp2 = Console.readLine();
    if (inp2.equals(inp1)) System.out.println(out); }
```

- Assume there would be functions

```
getLine :: String     putStrLn :: String -> ()
```

`()` is the unit type

```
f :: String -> ()
f out = let
          inp1 = getLine
          inp2 = getLine
  in if inp2 == inp1
     then putStrLn out
     else ()
```

- What does `inp2 == inp1` evaluate to?
- In which order are arguments evaluated?
- These functions cause side-effects!

`IO`: Input and Output

- Cannot use normal functions `putStrLn` and `getLine`
- Tag types with `IO` to capture side effects

  ```
  getLine :: IO String
  putStrLn :: String -> IO ()
  ```
- Syntax for `IO` type:
  - `do` block sequences side effects
  - `<-` extracts values from `IO`
  - `return` tags values with `IO`

```
f :: String -> ()
f out = let
      inp1 = getLine
      inp2 = getLine
  in if inp2 == inp1
      then putStrLn out
      else ()
```

```
f :: String -> IO ()
f out = do
inp1 <- getLine
inp2 <- getLine
if inp2 == inp1
  then putStrLn out
  else return ()
```

main

- `main :: IO ()` is the entry function for Haskell programs

```haskell
main :: IO ()
main = do
  putStrLn "Enter your name:"
  name <- getLine
  putStrLn ("Hello, " ++ name ++ "!")
```

- compile with GHC

```
> ghc hello.hs
> ./hello
Enter your name:
David
Hello, David!
```

- run within GHCi

```
> ghci
? :load hello.hs
? main
Enter your name:
David
Hello, David!
```

No escape from `IO`

- `IO` tag sticks to values
  Can compute with `IO` values only in `do` blocks
  - Clumsier than with pure expressions
  - Results are tagged, too.
- Stay out of `IO` as long as possible
- Separate computations from user interface

Side effects

```
main :: IO ()
main = do
  n <- getLine
  m <- getLine
  let x = gcd (read n) (read m)
  putStrLn (show x)
```

Pure

```
gcd :: Int -> Int -> Int
gcd x y
| x == y    = x
| x > y     = gcd (x-y) y
| otherwise = gcd x (y-x)
```

Converting to/from `String`

- `show` converts values to `String`s

```
? show 23
"23"

? show True
"True"

? show (17, 'a')
"(17,'a')"

? show (17 + 42)
"59"
```

Converting to/from String

- show converts values to Strings

```
? show 23
"23"

? show True
"True"

? show (17, 'a')
"(17,'a')"

? show (17 + 42)
"59"
```

- read converts Strings to values
  always specify desired type

```
? read "23" :: Integer
23

? read "23" :: Double
23.0

? read "(17, 'a')" :: (Int, Char)
(17,'a')

? (read "17" :: Int)+(read "42"::Int)
59

? read "17+42" :: Int
Exception: Prelude.read: no parse
```

Motivation message derivations

- Assume we have a network protocol which enables Alice and Bob to talk to each other.
- They talk about sensitive things, so they protect the messages using cryptography
- Charlie owns a router somewhere in the middle of the network and he'd like to learn (at least some part of) what Alice and Bob are talking about
- Can he combine the crypto messages he sees in some clever way to get to the secret stuff?
- Alternatively: what messages can he derive from the messages he sees?
- We'd like to reason about this formally

Crypto Messages

Let a set $\mathbf{A}$ of atomic messages be given. $\mathcal{L}_{\mathrm{M}}$, the language of messages, is the smallest set where:

- $M \in \mathcal{L}_{\mathrm{M}}$ if $M \in \mathbf{A}$
- $\langle A, B \rangle \in \mathcal{L}_{\mathrm{M}}$ if $A, B \in \mathcal{L}_{\mathrm{M}}$ (pairing)
- $\{M\}_K \in \mathcal{L}_{\mathrm{M}}$ if $M, K \in \mathcal{L}_{\mathrm{M}}$ (encryption)

Message Derivations

For a sequence of messages $M_1, \ldots, M_k$, we call $M_1, \ldots, M_k \vdash M$ a *sequent*.
Informally, this corresponds to the assertion:
M can be derived from the messages $M_1, \ldots, M_k$.

Derivation rules:

$$\frac{}{\Gamma, M \vdash M}\ \textsc{Ax} \qquad \frac{\Gamma \vdash A \qquad \Gamma \vdash B}{\Gamma \vdash \langle A, B \rangle}\ \textsc{Pair-I}$$

$$\frac{\Gamma \vdash \langle A, B \rangle}{\Gamma \vdash A}\ \textsc{Pair-EL} \qquad \frac{\Gamma \vdash \langle A, B \rangle}{\Gamma \vdash B}\ \textsc{Pair-ER}$$

$$\frac{\Gamma \vdash M \qquad \Gamma \vdash K}{\Gamma \vdash \{M\}_K}\ \textsc{Enc-I} \qquad \frac{\Gamma \vdash \{M\}_K \qquad \Gamma \vdash K}{\Gamma \vdash M}\ \textsc{Enc-E}$$

Derivations

A *derivation* is a tree.
Consider the sequence of messages $\Gamma = \langle k_1, k_2 \rangle, \{\{s\}_{k_1}\}_{k_2}$,
then the following tree is a derivation of the sequent $\Gamma \vdash s$.

Exercises I

- Derive the sequent $k_1, \{k_2\}_{k_1}, \{s\}_{k_1} \vdash \{s\}_{k_2}$.
- Derive the sequent $\langle a, \langle b, c\rangle\rangle, \{s\}_{\langle\langle a,b\rangle,c\rangle} \vdash s$.

Derivation rules:

$$\frac{}{\Gamma, M \vdash M}\ \textsc{Ax} \qquad \frac{\Gamma \vdash A \qquad \Gamma \vdash B}{\Gamma \vdash \langle A, B\rangle}\ \textsc{Pair-I}$$

$$\frac{\Gamma \vdash \langle A, B\rangle}{\Gamma \vdash A}\ \textsc{Pair-EL} \qquad \frac{\Gamma \vdash \langle A, B\rangle}{\Gamma \vdash B}\ \textsc{Pair-ER}$$

$$\frac{\Gamma \vdash M \qquad \Gamma \vdash K}{\Gamma \vdash \{M\}_K}\ \textsc{Enc-I} \qquad \frac{\Gamma \vdash \{M\}_K \qquad \Gamma \vdash K}{\Gamma \vdash M}\ \textsc{Enc-E}$$

Knowledge proofs

We now define the language of knowledge formulas $\mathcal{L}_\mathsf{F}$ as the smallest set where:

- M *known* $\in \mathcal{L}_\mathsf{F}$ if $M \in \mathcal{L}_\mathsf{M}$ (knowledge facts)
- $A \to B \in \mathcal{L}_\mathsf{F}$ if $A, B \in \mathcal{L}_\mathsf{F}$ (implication)

We can now write formulas such as
$\langle a, b \rangle$ *known* $\to \{a\}_b$ *known*.

Proof rules

$$\frac{}{\Gamma, A \vdash A}\ \textsc{Ax} \qquad \frac{\Gamma \vdash A\ \mathit{known} \qquad \Gamma \vdash B\ \mathit{known}}{\Gamma \vdash \langle A, B \rangle\ \mathit{known}}\ \textsc{Pair-I}$$

$$\frac{\Gamma \vdash \langle A, B \rangle\ \mathit{known}}{\Gamma \vdash A\ \mathit{known}}\ \textsc{Pair-EL} \qquad \frac{\Gamma \vdash \langle A, B \rangle\ \mathit{known}}{\Gamma \vdash B\ \mathit{known}}\ \textsc{Pair-ER}$$

$$\frac{\Gamma \vdash M\ \mathit{known} \qquad \Gamma \vdash K\ \mathit{known}}{\Gamma \vdash \{M\}_K\ \mathit{known}}\ \textsc{Enc-I}$$

$$\frac{\Gamma \vdash \{M\}_K\ \mathit{known} \qquad \Gamma \vdash K\ \mathit{known}}{\Gamma \vdash M\ \mathit{known}}\ \textsc{Enc-E}$$

$$\frac{\Gamma, A \vdash B}{\Gamma \vdash A \to B}\ \to\text{-I} \qquad \frac{\Gamma \vdash A \to B \qquad \Gamma \vdash A}{\Gamma \vdash B}\ \to\text{-E}$$

Proof

A *proof* of a formula F is a derivation of the sequent $\vdash F$.
Example: $\langle a, b \rangle$ *known* $\rightarrow \{a\}_b$ *known*

Exercises II

- Prove $a\ known \to \langle \{b\}_a, \{s\}_{\{a\}_b} \rangle\ known \to s\ known$.
- Prove $d\ known \to (\{s\}_b\ known \to b\ known) \to \{\langle \{\{s\}_b\}_c, c \rangle\}_d\ known \to s\ known$.

$$\frac{}{\Gamma, A \vdash A}\ \textsc{Ax} \qquad \frac{\Gamma \vdash A\ known \qquad \Gamma \vdash B\ known}{\Gamma \vdash \langle A, B \rangle\ known}\ \textsc{Pair-I}$$

$$\frac{\Gamma \vdash \langle A, B \rangle\ known}{\Gamma \vdash A\ known}\ \textsc{Pair-EL} \qquad \frac{\Gamma \vdash \langle A, B \rangle\ known}{\Gamma \vdash B\ known}\ \textsc{Pair-ER}$$

$$\frac{\Gamma \vdash M\ known \qquad \Gamma \vdash K\ known}{\Gamma \vdash \{M\}_K\ known}\ \textsc{Enc-I}$$

$$\frac{\Gamma \vdash \{M\}_K\ known \qquad \Gamma \vdash K\ known}{\Gamma \vdash M\ known}\ \textsc{Enc-E}$$

$$\frac{\Gamma, A \vdash B}{\Gamma \vdash A \to B}\ \to\text{-I} \qquad \frac{\Gamma \vdash A \to B \qquad \Gamma \vdash A}{\Gamma \vdash B}\ \to\text{-E}$$