COMP4075/G54RFP: Lecture 1

Administrative Details and Introduction

Henrik Nilsson

University of Nottingham, UK

Finding People and Information (1)

- Henrik Nilsson
  Room A08, Computer Science Building
  e-mail: `nhn@cs.nott.ac.uk`

Finding People and Information (1)

- Henrik Nilsson
  Room A08, Computer Science Building
  e-mail: `nhn@cs.nott.ac.uk`
- Main module web page:
  `www.cs.nott.ac.uk/~psznhn/G54RFP`
  `www.cs.nott.ac.uk/~psznhn/COMP4075`

Finding People and Information (1)

- Henrik Nilsson
  Room A08, Computer Science Building
  e-mail: `nhn@cs.nott.ac.uk`
- Main module web page:
  `www.cs.nott.ac.uk/~psznhn/G54RFP`
  `www.cs.nott.ac.uk/~psznhn/COMP4075`
- Moodle (COMP4075):
  `moodle.nottingham.ac.uk/`
  `course/view.php?id=94617`

Finding People and Information (2)

- Direct questions concerning lectures and coursework to the ***Moodle COMP4075/G54RFP Forum***.

Finding People and Information (2)

- Direct questions concerning lectures and coursework to the ***Moodle COMP4075/G54RFP Forum***.

  Anyone can ask and answer questions, but you must not post exact solutions to the coursework.

Aims and Motivation (1)

Aims and Motivation (1)

- Why did you opt to take this module?

Aims and Motivation (1)

- Why did you opt to take this module?
- What are good reasons to take this module?

Aims and Motivation (2)

- To introduces tools, techniques, and theory needed for programming real-world applications functionally.

Aims and Motivation (2)

- To introduces tools, techniques, and theory needed for programming real-world applications functionally.
- Particular emphasis on the inherent benefits of functional programming and strong typing for:
  - reuse
  - maintenance
  - concurrency
  - distribution
  - scalability
  - high availability

Aims and Motivation (3)

- Such aspects have:
  - contributed to the popularity of functional programming for demanding applications e.g. in the finance industry
  - have had a significant impact on the design of many languages and frameworks such as Java, C#, and Rust, MapReduce, React

Aims and Motivation (4)

We will use Haskell as medium of instruction, but:

- What is covered has broad applicability.
- Guest lectures and coursework provide opportunities to branch out beyond Haskell.

Content

The module will cover a range of topics, some more foundational, some applied, such as:

- Lazy functional programming
- Purely functional data structures
- Key libraries
- Functional design patterns
- Concurrency
- Web programming
- GUIs

Guest Lectures

- In the process of organising 3 to 4 guest lectures and/or tutorials.
- Time frame: November–December.
- To allow lecturers to travel on the day, these will likely take place in the afternoon slot or the lab slot. Possibly in an ad hoc slot if a lecturer cannot make the Friday.

Literature (1)

No main reference. The following two will be useful, though, both freely available online:

- *Haskell*, Wikibooks
- *Real World Haskell*, by Bryan O'Sullivan, John Goerzen, and Don Stewart

We will also use tutorials, research papers, videos, etc. References given on module web page or as we go along.

Lecture Notes

- Come prepared to take notes.

Lecture Notes

- Come prepared to take notes.
- All ***electronic*** slides, program code, and other supporting material in ***electronic*** form used during the lectures, will be made available on the course web page.

Lecture Notes

- Come prepared to take notes.
- All ***electronic*** slides, program code, and other supporting material in ***electronic*** form used during the lectures, will be made available on the course web page.
- ***However!*** The electronic record of the lectures is neither guaranteed to be complete nor self-contained!

Assessment

- 50 % unseen written examination (1.5 h),
  50 % coursework

Assessment

- 50 % unseen written examination (1.5 h),
  50 % coursework
- Coursework, 2 parts:
  - Part I: Basics; 15 h
  - Part II: Advanced topics and applications; 35 h

Assessment

- 50 % unseen written examination (1.5 h), 50 % coursework
- Coursework, 2 parts:
  - Part I: Basics; 15 h
  - Part II: Advanced topics and applications; 35 h
- Coursework support from 18 October.

Assessment

- 50 % unseen written examination (1.5 h),
  50 % coursework
- Coursework, 2 parts:
  - Part I: Basics; 15 h
  - Part II: Advanced topics and applications; 35 h
- Coursework support from 18 October.
- Use time until then to get up to speed on Haskell.

Coursework Timeline

Preliminary timeline (TBC):

- Part I:
  - Release: Wednesday 16 October
  - Deadline: Wednesday 6 November
- Part II:
  - Release: Wednesday 6 November
  - Deadline: Wednesday 11 December

Coursework Timeline

Preliminary timeline (TBC):

- Part I:
  - Release: Wednesday 16 October
  - Deadline: Wednesday 6 November
- Part II:
  - Release: Wednesday 6 November
  - Deadline: Wednesday 11 December

Start early! It is ***not*** possible to do this coursework at the last minute.

COMP4095: Optional Project (1)

- 10 credits (100 hours)

COMP4095: Optional Project (1)

- 10 credits (100 hours)
- Opportunity to learn in depth about aspects of functional programming at scale.

COMP4095: Optional Project (1)

- 10 credits (100 hours)
- Opportunity to learn in depth about aspects of functional programming at scale.
- Project must be clearly related to what is covered in COMP4075/G54RFP, but "functional" interpreted in a broad sense.

COMP4095: Optional Project (1)

- 10 credits (100 hours)
- Opportunity to learn in depth about aspects of functional programming at scale.
- Project must be clearly related to what is covered in COMP4075/G54RFP, but "functional" interpreted in a broad sense.
- Project defined through a "pitch" that must be discussed and agreed. Needs to clarify:
  - The relevance of the project to COMP4075
  - Size appropriate for 10 credits

COMP4095: Optional Project (2)

Preliminary timeline (TBC):

- Release of project criteria: Wednesday 13 November
- Pitch deadline: Wednesday 4 December (but earlier is better)
- Submission deadline (code and report): 15 January

Imperative vs. Declarative (1)

- ***Imperative Languages***:
  - Implicit state.
  - Computation essentially a sequence of side-effecting actions.
  - Examples: Procedural and OO languages

Imperative vs. Declarative (1)

- ***Imperative Languages***:
  - Implicit state.
  - Computation essentially a sequence of side-effecting actions.
  - Examples: Procedural and OO languages
- ***Declarative Languages*** (Lloyd 1994):
  - ***No*** implicit state.
  - A program can be regarded as a theory.
  - Computation can be seen as deduction from this theory.
  - Examples: Logic and Functional Languages.

Imperative vs. Declarative (2)

Another perspective:

- ***Algorithm = Logic + Control***

Imperative vs. Declarative (2)

Another perspective:

- ***Algorithm = Logic + Control***
- Declarative programming emphasises the logic (“what”) rather than the control (“how”).

Imperative vs. Declarative (2)

Another perspective:

- ***Algorithm = Logic + Control***
- Declarative programming emphasises the logic (“what”) rather than the control (“how”).
- Strategy needed for providing the “how”:
  - Resolution (logic programming languages)
  - Lazy evaluation (some functional and logic programming languages)
  - (Lazy) narrowing: (functional logic programming languages)

No Control?

Declarative languages for practical use tend to be only ***weakly declarative***; i.e., not totally free of control aspects. For example:

No Control?

Declarative languages for practical use tend to be only ***weakly declarative***; i.e., not totally free of control aspects. For example:

- Equations in functional languages are directed.

No Control?

Declarative languages for practical use tend to be only ***weakly declarative***; i.e., not totally free of control aspects. For example:

- Equations in functional languages are directed.
- Order of patterns often matters for pattern matching.

No Control?

Declarative languages for practical use tend to be only ***weakly declarative***; i.e., not totally free of control aspects. For example:

- Equations in functional languages are directed.
- Order of patterns often matters for pattern matching.
- Constructs for taking control over the order of evaluation. (E.g. `cut` in Prolog, `seq` in Haskell.)

Relinquishing Control

Theme of this and next lecture: ***relinquishing control by exploiting lazy evaluation***.

- Evaluation orders
- Strict vs. Non-strict semantics
- Lazy evaluation
- Applications of lazy evaluation:
  - Programming with infinite structures
  - Circular programming
  - Dynamic programming
  - Attribute grammars

Evaluation Orders (1)

Consider:

```
sqr x = x * x
dbl x = x + x
main = sqr (dbl (2 + 3))
```

Roughly, any expression that can be evaluated or ***reduced*** by using the equations as rewrite rules is called a ***reducible expression*** or ***redex.***

Assuming arithmetic, the redexes of the body of `main` are: 2 + 3
dbl (2 + 3)
sqr (dbl (2 + 3))

Evaluation Orders (2)

Thus, in general, many possible reduction orders. Innermost, leftmost redex first is called ***Applicative Order Reduction*** (AOR). Recall:

```
sqr x = x * x
dbl x = x + x
main = sqr (dbl (2 + 3))
```

Starting from `main`:

$\underline{\texttt{main}} \Rightarrow \texttt{sqr (dbl (}\underline{\texttt{2 + 3}}\texttt{))} \Rightarrow \texttt{sqr (}\underline{\texttt{dbl 5}}\texttt{)}$
$\Rightarrow \texttt{sqr (}\underline{\texttt{5 + 5}}\texttt{)} \Rightarrow \underline{\texttt{sqr 10}} \Rightarrow \underline{\texttt{10 * 10}} \Rightarrow \texttt{100}$

This is just ***Call-By-Value***.

Evaluation Orders (3)

Outermost, leftmost redex first is called ***Normal Order Reduction*** (NOR):

```
main ⇒ sqr (dbl (2 + 3))
⇒ dbl (2 + 3) * dbl (2 + 3)
⇒ ((2 + 3) + (2 + 3)) * dbl (2 + 3)
⇒ (5 + (2 + 3)) * dbl (2 + 3)
⇒ (5 + 5) * dbl (2 + 3) ⇒ 10 * dbl (2 + 3)
⇒ ... ⇒ 10 * 10 ⇒ 100
```

(Applications of arithmetic operations only considered redexes once arguments are numbers.)
Demand-driven evaluation or ***Call-By-Need***

Why Normal Order Reduction? (1)

NOR seems rather inefficient. Any use?

- Best possible termination properties.

  A pure functional languages is just the λ-calculus in disguise. Two central theorems:

  - Church-Rosser Theorem I:
    No term has more than one normal form.
  - Church-Rosser Theorem II:
    If a term has a normal form, then NOR will find it.

Why Normal Order Reduction? (2)

- More expressive power; e.g.:
  - “Infinite” data structures
  - Circular programming
- More declarative code as control aspects (order of evaluation) left implicit.

Exercise 1

Consider:

```
f x = 1
g x = g x
main = f (g 0)
```

Attempt to evaluate `main` using both AOR and NOR. Which order is the more efficient in this case? (Count the number of reduction steps to normal form.)

Strict vs. Non-strict Semantics (1)

- $\perp$, or "bottom", the ***undefined value***, representing ***errors*** and ***non-termination***.
- A function f is ***strict*** iff:

$$f \perp = \perp$$

For example, $+$ is strict in both its arguments:

$$\begin{aligned}(0/0) + 1 &= \perp + 1 = \perp \\ 1 + (0/0) &= 1 + \perp = \perp\end{aligned}$$

Strict vs. Non-strict Semantics (2)

Again, consider:

```
f x = 1
g x = g x
```

What is the value of `f (0/0)`? Or of `f (g 0)`?

- AOR: $\texttt{f}\ (\underline{\texttt{0/0}}) \Rightarrow \perp$; $\texttt{f}\ (\underline{\texttt{g 0}}) \Rightarrow \perp$
  Conceptually, $\texttt{f} \perp = \perp$; i.e., `f` is strict.
- NOR: $\underline{\texttt{f (0/0)}} \Rightarrow 1$; $\underline{\texttt{f (g 0)}} \Rightarrow 1$
  Conceptually, $\texttt{f} \perp = 1$; i.e., `f` is non-strict.

Thus, NOR results in non-strict semantics.

Lazy Evaluation (1)

Lazy evaluation is a ***technique for implementing NOR*** more efficiently:

Lazy Evaluation (1)

Lazy evaluation is a ***technique for implementing NOR*** more efficiently:

- A redex is evaluated ***only if needed***.

Lazy Evaluation (1)

Lazy evaluation is a ***technique for implementing NOR*** more efficiently:

- A redex is evaluated ***only if needed***.
- ***Sharing*** employed to avoid duplicating redexes.

Lazy Evaluation (1)

Lazy evaluation is a ***technique for implementing NOR*** more efficiently:

- A redex is evaluated ***only if needed***.
- ***Sharing*** employed to avoid duplicating redexes.
- Once evaluated, a redex is ***updated*** with the result to avoid evaluating it more than once.

Lazy Evaluation (1)

Lazy evaluation is a ***technique for implementing NOR*** more efficiently:

- A redex is evaluated ***only if needed***.
- ***Sharing*** employed to avoid duplicating redexes.
- Once evaluated, a redex is ***updated*** with the result to avoid evaluating it more than once.

As a result, under lazy evaluation, any one redex is evaluated at most once.

Lazy Evaluation (2)

Recall:

```
sqr x = x * x
dbl x = x + x
main =
   sqr (dbl (2+3))
```

<u>`sqr (dbl (2 + 3))`</u>

Lazy Evaluation (2)

Recall:

```
sqr x = x * x
dbl x = x + x
main =
   sqr (dbl (2+3))
```

$\underline{\texttt{sqr (dbl (2 + 3))}}$

$\Rightarrow$ $\boxed{\underline{\texttt{dbl (2 + 3)}}}$ * $(\bullet)$

Lazy Evaluation (2)

Recall:

```
sqr x = x * x
dbl x = x + x
main =
    sqr (dbl (2+3))
```

sqr (dbl (2 + 3))

⇒ dbl (2 + 3) * (●)

⇒ ((2 + 3) + (●)) * (●)

Lazy Evaluation (2)

Recall:

```
sqr x = x * x
dbl x = x + x
main =
    sqr (dbl (2+3))
```

```
sqr (dbl (2 + 3))
⇒ dbl (2 + 3) * (●)
⇒ ((2 + 3) + (●)) * (●)
⇒ (5 + (●)) * (●)
```

Lazy Evaluation (2)

Recall:

```
sqr x = x * x
dbl x = x + x
main =
   sqr (dbl (2+3))
```

```
sqr (dbl (2 + 3))
⇒ dbl (2 + 3) * (•)
⇒ ((2 + 3) + (•)) * (•)
⇒ (5 + (•)) * (•)
⇒ 10 * (•)
```

Lazy Evaluation (2)

Recall:

```
sqr x = x * x
dbl x = x + x
main =
   sqr (dbl (2+3))
```

```
sqr (dbl (2 + 3))
⇒ dbl (2 + 3) * (•)
⇒ ((2 + 3) + (•)) * (•)
⇒ (5 + (•)) * (•)
⇒ 10 * (•)
⇒ 100
```

Lazy Evaluation (3)

"Evaluated at most once" needs to be interpreted with care: it referes to individual redex ***instances***.

Lazy Evaluation (3)

“Evaluated at most once” needs to be interpreted with care: it referes to individual redex ***instances***.

For example:

- `(1 + 2) * (1 + 2)`

  `1 + 2` evaluated twice as ***not the same*** redex.

Lazy Evaluation (3)

"Evaluated at most once" needs to be interpreted with care: it referes to individual redex ***instances***.

For example:

- `(1 + 2) * (1 + 2)`

  `1 + 2` evaluated twice as ***not the same*** redex.

- `f x = x + y where y = 6 * 7`

  `6 * 7` evaluated whenever `f` is called.

Lazy Evaluation (3)

"Evaluated at most once" needs to be interpreted with care: it referes to individual redex ***instances***.

For example:

- `(1 + 2) * (1 + 2)`

  `1 + 2` evaluated twice as ***not the same*** redex.

- `f x = x + y where y = 6 * 7`

  `6 * 7` evaluated whenever `f` is called.

A good compiler will rearrange such computations to avoid duplication of effort, but this has nothing to do with laziness.

Lazy Evaluation (4)

Memoization means caching function results to avoid re-computing them. Also distinct from laziness.

Exercise 2

Evaluate `main` using AOR, NOR, and lazy evaluation:

```
f x y z  =  x * z
g x      =  f (x * x) (x * 2) x
main     =  g (1 + 2)
```

(Only consider an applications of an arithmetic operator a redex once the arguments are numbers.)

How many reduction steps in each case?

Exercise 2

Evaluate `main` using AOR, NOR, and lazy evaluation:

```
f x y z  =  x * z
g x      =  f (x * x) (x * 2) x
main     =  g (1 + 2)
```

(Only consider an applications of an arithmetic operator a redex once the arguments are numbers.)

How many reduction steps in each case?

Answer: 7, 8, 6 respectively

Reading

- John W. Lloyd. Practical advantages of declarative programming. In *Joint Conference on Declarative Programming, GULP-PRODE'94*, 1994.
- John Hughes. Why Functional Programming Matters. *The Computer Journal*, 32(2):98–197, April 1989.

COMP4075/G54RFP: Lecture 2 & 3

Pure Functional Programming: Exploiting Laziness

Henrik Nilsson

University of Nottingham, UK

Recap: Lazy Evaluation (1)

Lazy evaluation is a ***technique for implementing NOR*** more efficiently:

Recap: Lazy Evaluation (1)

Lazy evaluation is a ***technique for implementing NOR*** more efficiently:

- A redex is evaluated ***only if needed***.

Recap: Lazy Evaluation (1)

Lazy evaluation is a ***technique for implementing NOR*** more efficiently:

- A redex is evaluated ***only if needed***.
- ***Sharing*** employed to avoid duplicating redexes.

Recap: Lazy Evaluation (1)

Lazy evaluation is a ***technique for implementing NOR*** more efficiently:

- A redex is evaluated ***only if needed***.
- ***Sharing*** employed to avoid duplicating redexes.
- Once evaluated, a redex is ***updated*** with the result to avoid evaluating it more than once.

Recap: Lazy Evaluation (1)

Lazy evaluation is a ***technique for implementing NOR*** more efficiently:

- A redex is evaluated ***only if needed***.
- ***Sharing*** employed to avoid duplicating redexes.
- Once evaluated, a redex is ***updated*** with the result to avoid evaluating it more than once.

As a result, under lazy evaluation, any one redex is evaluated at most once.

Recap: Lazy Evaluation (2)

Recall:

```
sqr x = x * x
dbl x = x + x
main =
   sqr (dbl (2+3))
```

`sqr (dbl (2 + 3))`

Recap: Lazy Evaluation (2)

Recall:

```
sqr x = x * x
dbl x = x + x
main =
   sqr (dbl (2+3))
```

```
sqr (dbl (2 + 3))

⇒ dbl (2 + 3)  *  (•)
```

Recap: Lazy Evaluation (2)

Recall:

```
sqr x = x * x
dbl x = x + x
main =
    sqr (dbl (2+3))
```

```
sqr (dbl (2 + 3))
⇒ dbl (2 + 3) * (•)
⇒ ((2 + 3) + (•)) * (•)
```

Recap: Lazy Evaluation (2)

Recall:

```
sqr x = x * x
dbl x = x + x
main =
   sqr (dbl (2+3))
```

```
  sqr (dbl (2 + 3))
⇒ dbl (2 + 3) * (•)
⇒ ((2 + 3) + (•)) * (•)
⇒ (5 + (•)) * (•)
```

Recap: Lazy Evaluation (2)

Recall:

```
sqr x = x * x
dbl x = x + x
main =
   sqr (dbl (2+3))
```

```
sqr (dbl (2 + 3))
⇒ dbl (2 + 3)  *  (•)
⇒ ((2 + 3) + (•))  *  (•)
⇒ (5 + (•))  *  (•)
⇒ 10  *  (•)
```

Recap: Lazy Evaluation (2)

Recall:

```
sqr x = x * x
dbl x = x + x
main =
    sqr (dbl (2+3))
```

sqr (dbl (2 + 3))

⇒ dbl (2 + 3) * (•)

⇒ ((2 + 3) + (•)) * (•)

⇒ (5 + (•)) * (•)

⇒ 10 * (•)

⇒ 100

Infinite Data Structures (1)

```
take 0 _      = []
take n []     = []
take n (x:xs) = x : take (n-1) xs

from n = n : from (n+1)

nats = from 0

main = take 5 nats
```

Infinite Data Structures (2)

`main`

`nats`

Infinite Data Structures (2)

`main` $\Rightarrow^1$ `take 5 (•)`

`nats`

Infinite Data Structures (2)

`main` $\Rightarrow^1$ `take 5 (•)`

`nats` $\Rightarrow^2$ `from 0`

Infinite Data Structures (2)

$$\underline{\texttt{main}} \Rightarrow^{1} \underline{\texttt{take 5 }(\bullet)}$$

$$\boxed{\underline{\texttt{nats}}} \Rightarrow^{2} \underline{\texttt{from 0}} \Rightarrow^{3} 0 : \boxed{\underline{\texttt{from 1}}}$$

Infinite Data Structures (2)

$$\underline{\texttt{main}} \Rightarrow^{1} \underline{\texttt{take 5 }(\bullet)} \Rightarrow^{4} \texttt{0:}\underline{\texttt{take 4 }(\bullet)}$$

$$\boxed{\underline{\texttt{nats}}} \Rightarrow^{2} \underline{\texttt{from 0}} \Rightarrow^{3} \texttt{0:}\boxed{\underline{\texttt{from 1}}}$$

Infinite Data Structures (2)

$\texttt{main} \Rightarrow^{1} \texttt{take 5 (}\bullet\texttt{)} \Rightarrow^{4} \texttt{0:take 4 (}\bullet\texttt{)}$

$\boxed{\texttt{nats}} \Rightarrow^{2} \texttt{from 0} \Rightarrow^{3} \texttt{0:}\boxed{\texttt{from 1}}$

$\Rightarrow^{5} \texttt{0:1:}\boxed{\texttt{from 2}}$

Infinite Data Structures (2)

main $\Rightarrow^1$ take 5 (•) $\Rightarrow^4$ 0:take 4 (•)

$\Rightarrow^6$ 0:1:take 3 (•)

nats $\Rightarrow^2$ from 0 $\Rightarrow^3$ 0:from 1

$\Rightarrow^5$ 0:1:from 2

Infinite Data Structures (2)

main $\Rightarrow^1$ take 5 (•) $\Rightarrow^4$ 0:take 4 (•)

$\Rightarrow^6$ 0:1:take 3 (•)

nats $\Rightarrow^2$ from 0 $\Rightarrow^3$ 0:from 1

$\Rightarrow^5$ 0:1:from 2 $\Rightarrow^7$ ...

Infinite Data Structures (2)

$\underline{\texttt{main}} \Rightarrow^{1} \underline{\texttt{take 5 }(\bullet)} \Rightarrow^{4} \texttt{0:}\underline{\texttt{take 4 }(\bullet)}$

$\Rightarrow^{6} \texttt{0:1:}\underline{\texttt{take 3 }(\bullet)} \Rightarrow^{8} \ldots$

$\boxed{\underline{\texttt{nats}}} \Rightarrow^{2} \underline{\texttt{from 0}} \Rightarrow^{3} \texttt{0:}\boxed{\underline{\texttt{from 1}}}$

$\Rightarrow^{5} \texttt{0:1:}\boxed{\underline{\texttt{from 2}}} \Rightarrow^{7} \ldots$

Infinite Data Structures (2)

$\underline{\texttt{main}} \Rightarrow^{1} \underline{\texttt{take 5 }(\bullet)} \Rightarrow^{4} \texttt{0:}\underline{\texttt{take 4 }(\bullet)}$

$\Rightarrow^{6} \texttt{0:1:}\underline{\texttt{take 3 }(\bullet)} \Rightarrow^{8} \ldots$

$\boxed{\underline{\texttt{nats}}} \Rightarrow^{2} \underline{\texttt{from 0}} \Rightarrow^{3} \texttt{0:}\boxed{\underline{\texttt{from 1}}}$

$\Rightarrow^{5} \texttt{0:1:}\boxed{\underline{\texttt{from 2}}} \Rightarrow^{7} \ldots \Rightarrow \texttt{0:1:2:3:4:}\boxed{\underline{\texttt{from 5}}}$

Infinite Data Structures (2)

$\underline{\texttt{main}} \Rightarrow^{1} \underline{\texttt{take 5 }(\bullet)} \Rightarrow^{4} \texttt{0:}\underline{\texttt{take 4 }(\bullet)}$

$\Rightarrow^{6} \texttt{0:1:}\underline{\texttt{take 3 }(\bullet)} \Rightarrow^{8} \texttt{...}$

$\Rightarrow \texttt{0:1:2:3:4:}\underline{\texttt{take 0 }(\bullet)}$

$\boxed{\underline{\texttt{nats}}} \Rightarrow^{2} \underline{\texttt{from 0}} \Rightarrow^{3} \texttt{0:}\boxed{\underline{\texttt{from 1}}}$

$\Rightarrow^{5} \texttt{0:1:}\boxed{\underline{\texttt{from 2}}} \Rightarrow^{7} \texttt{...} \Rightarrow \texttt{0:1:2:3:4:}\boxed{\underline{\texttt{from 5}}}$

Infinite Data Structures (2)

$\underline{\texttt{main}} \Rightarrow^1 \underline{\texttt{take 5 }(\bullet)} \Rightarrow^4 \texttt{0:}\underline{\texttt{take 4 }(\bullet)}$

$\Rightarrow^6 \texttt{0:1:}\underline{\texttt{take 3 }(\bullet)} \Rightarrow^8 \ldots$

$\Rightarrow \texttt{0:1:2:3:4:}\underline{\texttt{take 0 }(\bullet)} \Rightarrow \texttt{[0,1,2,3,4]}$

$\boxed{\underline{\texttt{nats}}} \Rightarrow^2 \underline{\texttt{from 0}} \Rightarrow^3 \texttt{0:}\boxed{\underline{\texttt{from 1}}}$

$\Rightarrow^5 \texttt{0:1:}\boxed{\underline{\texttt{from 2}}} \Rightarrow^7 \ldots \Rightarrow \texttt{0:1:2:3:4:}\boxed{\underline{\texttt{from 5}}}$

Circular Data Structures (1)

```
take 0 _      = []
take n []     = []
take n (x:xs) = x : take (n-1) xs

ones = 1 : ones

main = take 5 ones
```

Circular Data Structures (2)

main

$\boxed{\text{ones}}$

Circular Data Structures (2)

`main` $\Rightarrow^1$ `take 5 (•)`

`ones`

Circular Data Structures (2)

`main` $\Rightarrow^1$ `take 5 (•)`

`ones` $\Rightarrow^2$ `1 : •`

Circular Data Structures (2)

$\underline{\texttt{main}} \Rightarrow^{1} \underline{\texttt{take 5 }(\bullet)} \Rightarrow^{3} 1:\underline{\texttt{take 4 }(\bullet)}$

$\boxed{\underline{\texttt{ones}}} \Rightarrow^{2} \boxed{1 : \bullet}$

Circular Data Structures (2)

main $\Rightarrow^1$ take 5 (•) $\Rightarrow^3$ 1:take 4 (•)

$\Rightarrow^4$ 1:1:take 3 (•)

$\boxed{\texttt{ones}}$ $\Rightarrow^2$ $\boxed{\texttt{1 : •}}$

Circular Data Structures (2)

$\underline{\texttt{main}} \Rightarrow^{1} \underline{\texttt{take 5 }(\bullet)} \Rightarrow^{3} \texttt{1:}\underline{\texttt{take 4 }(\bullet)}$

$\Rightarrow^{4} \texttt{1:1:}\underline{\texttt{take 3 }(\bullet)} \Rightarrow^{5} \ldots$

$\boxed{\underline{\texttt{ones}}} \Rightarrow^{2} \boxed{\texttt{1 : }\bullet}$

Circular Data Structures (2)

$\underline{\texttt{main}} \Rightarrow^{1} \underline{\texttt{take 5 }(\bullet)} \Rightarrow^{3} \texttt{1:}\underline{\texttt{take 4 }(\bullet)}$

$\Rightarrow^{4} \texttt{1:1:}\underline{\texttt{take 3 }(\bullet)} \Rightarrow^{5} \ldots$

$\Rightarrow \texttt{1:1:1:1:1:}\underline{\texttt{take 0 }(\bullet)}$

$\boxed{\underline{\texttt{ones}}} \Rightarrow^{2} \boxed{\texttt{1 : }\bullet}$

Circular Data Structures (2)

main $\Rightarrow^1$ take 5 (•) $\Rightarrow^3$ 1:take 4 (•)

$\Rightarrow^4$ 1:1:take 3 (•) $\Rightarrow^5$ ...

$\Rightarrow$ 1:1:1:1:1:take 0 (•) $\Rightarrow$ [1,1,1,1,1]

ones $\Rightarrow^2$ 1 : •

Exercise

Given the following tree type

```
data Tree = Empty
          | Node Tree Int Tree
```

define:

- An infinite tree where every node is labelled by `1`.
- An infinite tree where every node is labelled by its depth from the root node.

Exercise: Solution

```
treeOnes = Node treeOnes 1 treeOnes

treeFrom n = Node (treeFrom (n + 1))
                  n
                  (treeFrom (n + 1))

treeDepths = treeFrom 0
```

Circular Programming (1)

A non-empty tree type:

```
data Tree = Leaf Int | Node Tree Tree
```

Circular Programming (1)

A non-empty tree type:

```
data Tree = Leaf Int | Node Tree Tree
```

Suppose we would like to write a function that replaces each leaf integer in a given tree with the ***smallest*** integer in that tree.

Circular Programming (1)

A non-empty tree type:

```
data Tree = Leaf Int | Node Tree Tree
```

Suppose we would like to write a function that replaces each leaf integer in a given tree with the ***smallest*** integer in that tree.

How many passes over the tree are needed?

Circular Programming (1)

A non-empty tree type:

```
data Tree = Leaf Int | Node Tree Tree
```

Suppose we would like to write a function that replaces each leaf integer in a given tree with the ***smallest*** integer in that tree.

How many passes over the tree are needed?

One!

Circular Programming (2)

Write a function that replaces all leaf integers by a given integer, and returns the new tree along with the smallest integer of the given tree:

```
fmr :: Int -> Tree -> (Tree, Int)
fmr m (Leaf i) = (Leaf m, i)
fmr m (Node tl tr) =
    (Node tl' tr', min ml mr)
    where
        (tl', ml) = fmr m tl
        (tr', mr) = fmr m tr
```

Circular Programming (3)

For a given tree `t`, the desired tree is now obtained as the ***solution*** to the equation:

```
(t', m) = fmr m t
```

Thus:

```
findMinReplace t = t'
    where
        (t', m) = fmr m t
```

Intuitively, this works because `fmr` can compute its result without needing to know the ***value*** of `m`.

A Simple Spreadsheet Evaluator (1)

	a	b	c
1	c3 + c2		
2	a3 * b2	2	a2 + b2
3	7		a2 + a3

s

$\Rightarrow$

	a	b	c
1	37		
2	14	2	16
3	7		21

s'

```
s' = array (bounds s)
           [ (r, evalCell s' (s ! r))
           | r <- indices s ]
```

The evaluated sheet is again simply the ***solution*** to the stated equation. No need to worry about evaluation order. ***Any caveats?***

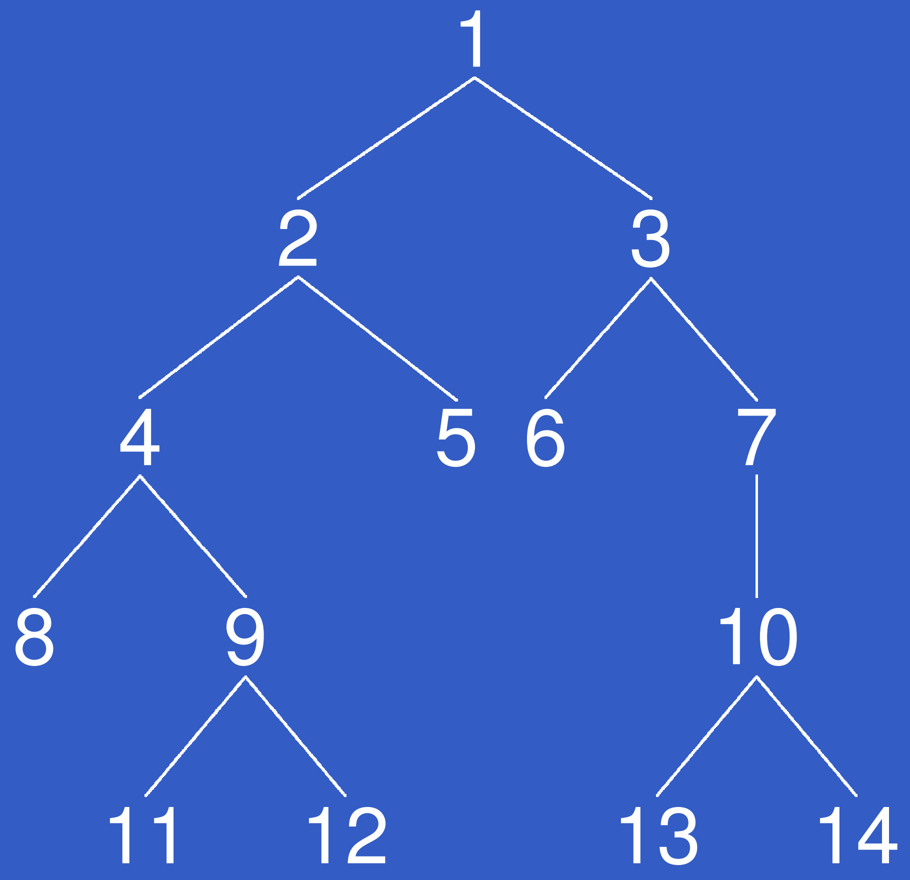
A Simple Spreadsheet Evaluator (2)

As it is quite instructive, let us develop this evaluator together. Some definitions to get us started:

```
type CellRef = (Char, Int)

type Sheet a = Array CellRef a

data BinOp = Add | Sub | Mul | Div

data Exp = Lit Double
         | Ref CellRef
         | App BinOp Exp Exp
```

Breadth-first Numbering (1)

Consider the problem of numbering a possibly infinitely deep tree in breadth-first order:



Breadth-first Numbering (2)

The following algorithm is due to G. Jones and J. Gibbons (1992), but the presentation differs.

Consider the following tree type:

```
data Tree a = Empty
            | Node (Tree a) a (Tree a)
```

Define:

$\mathrm{width}\ t\ i$	The width of a tree t at level i (0 origin).
$\mathrm{label}\ t\ i\ j$	The jth label at level i of a tree t (0 origin).

Breadth-first Numbering (3)

The following system of equations defines breadth-first numbering:

$$
\begin{aligned}
\mathrm{label}\; t\; 0\; 0 &= 1 & (1)\\
\mathrm{label}\; t\; (i+1)\; 0 &= \mathrm{label}\; t\; i\; 0 + \mathrm{width}\; t\; i & (2)\\
\mathrm{label}\; t\; i\; (j+1) &= \mathrm{label}\; t\; i\; j + 1 & (3)
\end{aligned}
$$

Note that $\mathrm{label}\; t\; i\; 0$ is defined for ***all*** levels i (as long as the widths of all tree levels are finite).

Breadth-first Numbering (4)

The code that follows sets up the defining system of equations:

Breadth-first Numbering (4)

The code that follows sets up the defining system of equations:

- ***Streams*** (infinite lists) of labels are used as a ***mediating data structure*** to allow equations to be set up between adjacent nodes within levels and between the last node at one level and the first node at the next.

Breadth-first Numbering (4)

The code that follows sets up the defining system of equations:

- ***Streams*** (infinite lists) of labels are used as a ***mediating data structure*** to allow equations to be set up between adjacent nodes within levels and between the last node at one level and the first node at the next.
- Idea: the tree numbering function for a subtree takes a stream of labels for the ***first node*** at each level, and returns a stream of labels for the ***node after the last node*** at each level.

Breadth-first Numbering (5)

- As there manifestly are ***no cyclic dependences*** among the equations, we can entrust the details of solving them to the lazy evaluation machinery in the safe knowledge that a solution will be found.

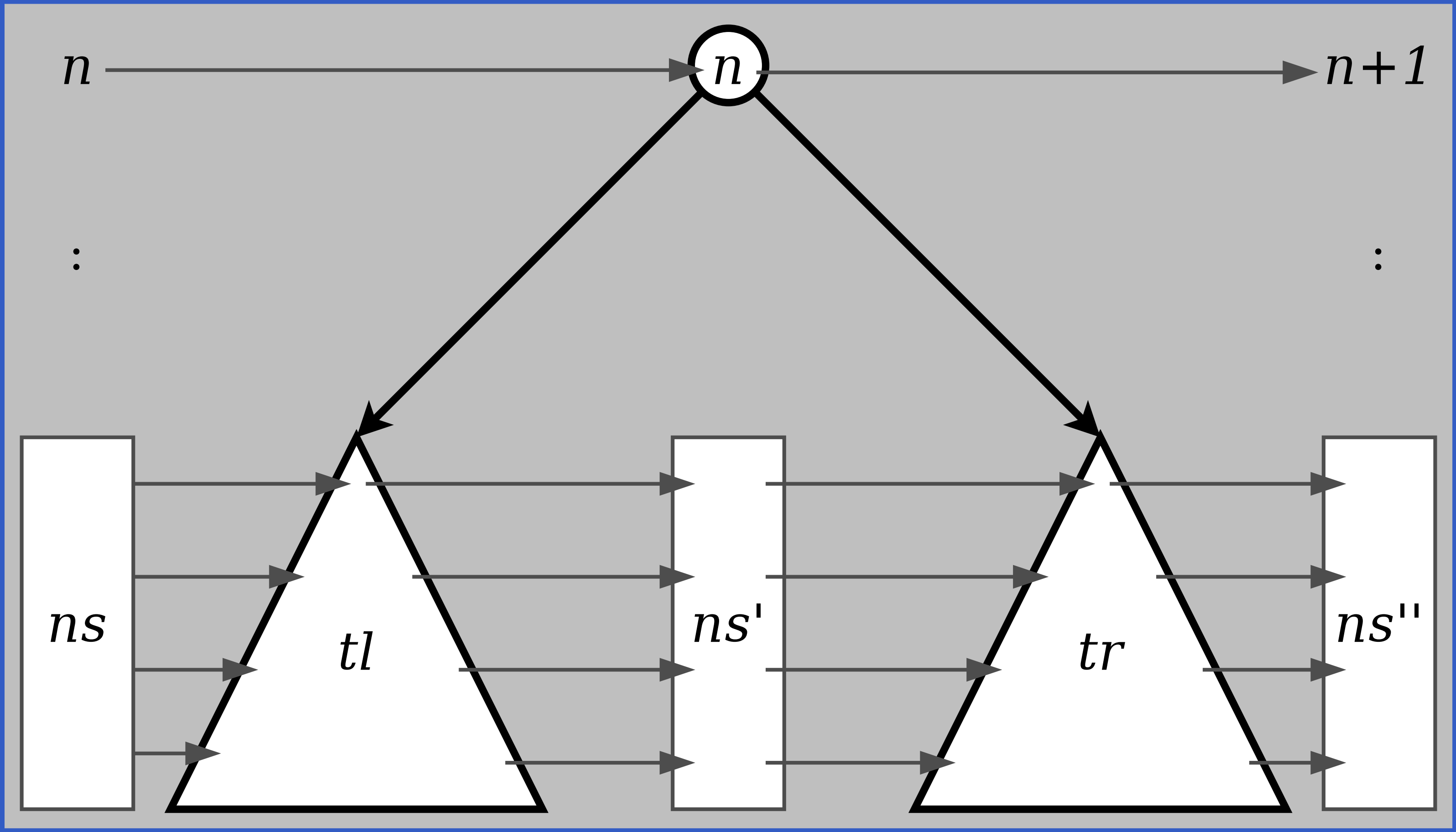
Breadth-first Numbering (6)

Eqns (1) & (2)

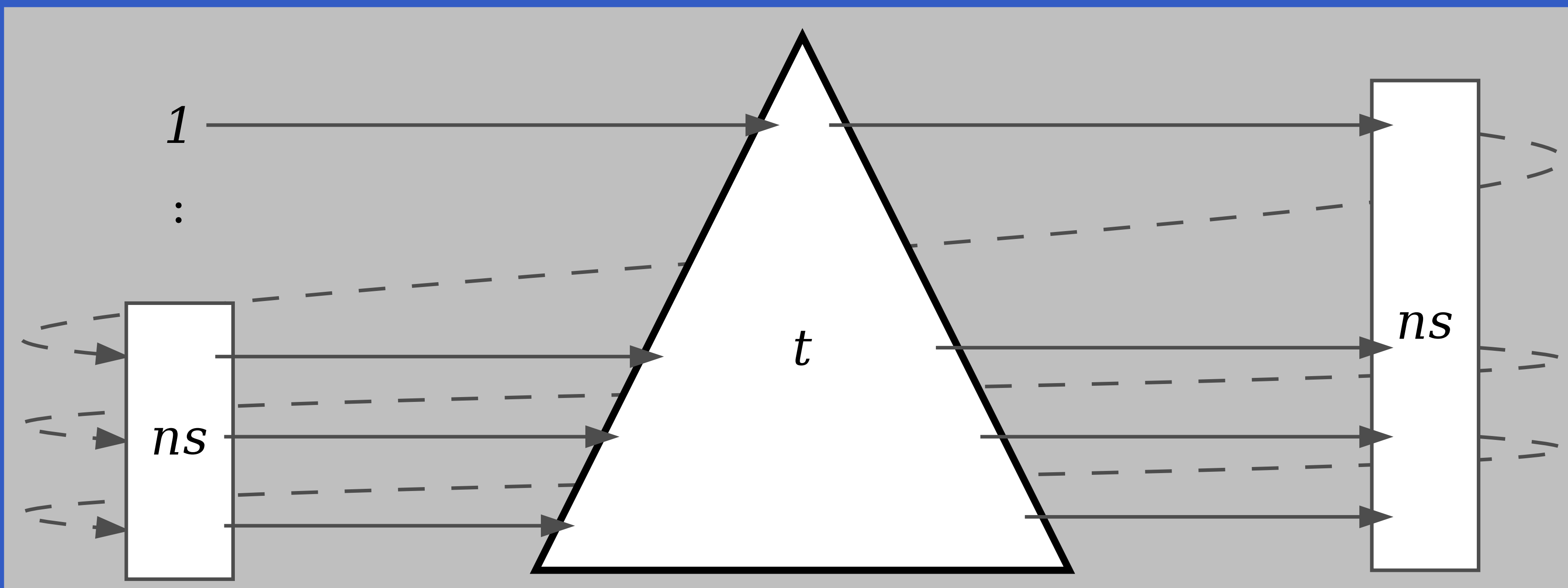
```
bfn :: Tree a -> Tree Integer
bfn t = t'
    where
        (ns, t') = bfnAux (1 : ns) t

bfnAux :: [Integer] -> Tree a
               -> ([Integer], Tree Integer)
bfnAux ns       Empty            = (ns, Empty)
bfnAux (n : ns) (Node tl _ tr) = ((n + 1) : ns'',
                                    Node tl' n tr')
    where
        (ns',  tl') = bfnAux ns tl
        (ns'', tr') = bfnAux ns' tr
```

Eqn (3)

Breadth-first Numbering (7)



Breadth-first Numbering (8)



Dynamic Programming

Dynamic Programming:

- Create a ***table*** of all subproblems that ever will have to be solved.
- Fill in table without regard to whether the solution to that particular subproblem will be needed.
- Combine solutions to form overall solution.

Dynamic Programming

Dynamic Programming:

- Create a ***table*** of all subproblems that ever will have to be solved.
- Fill in table without regard to whether the solution to that particular subproblem will be needed.
- Combine solutions to form overall solution.

Lazy Evaluation is perfect match: no need to worry about finding a suitable evaluation order.

Dynamic Programming

Dynamic Programming:

- Create a ***table*** of all subproblems that ever will have to be solved.
- Fill in table without regard to whether the solution to that particular subproblem will be needed.
- Combine solutions to form overall solution.

Lazy Evaluation is perfect match: no need to worry about finding a suitable evaluation order.

In effect, using laziness to implement limited form of ***memoization***.

The Triangulation Problem (1)

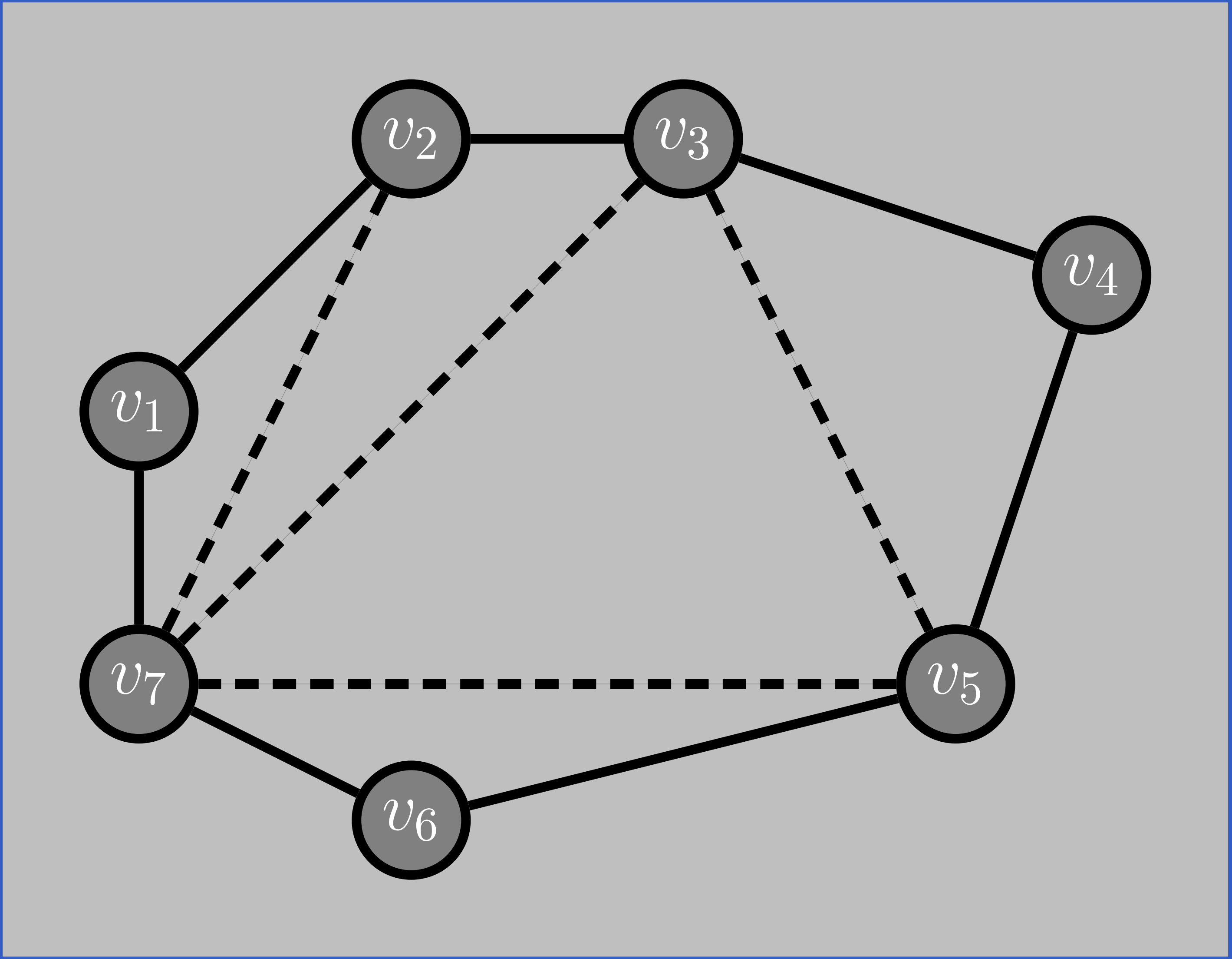
Select a set of ***chords*** that divides a convex polygon into triangles such that:

- no two chords cross each other
- the sum of their length is minimal.

We will only consider computing the minimal length.

See Aho, Hopcroft, Ullman (1983) for details.

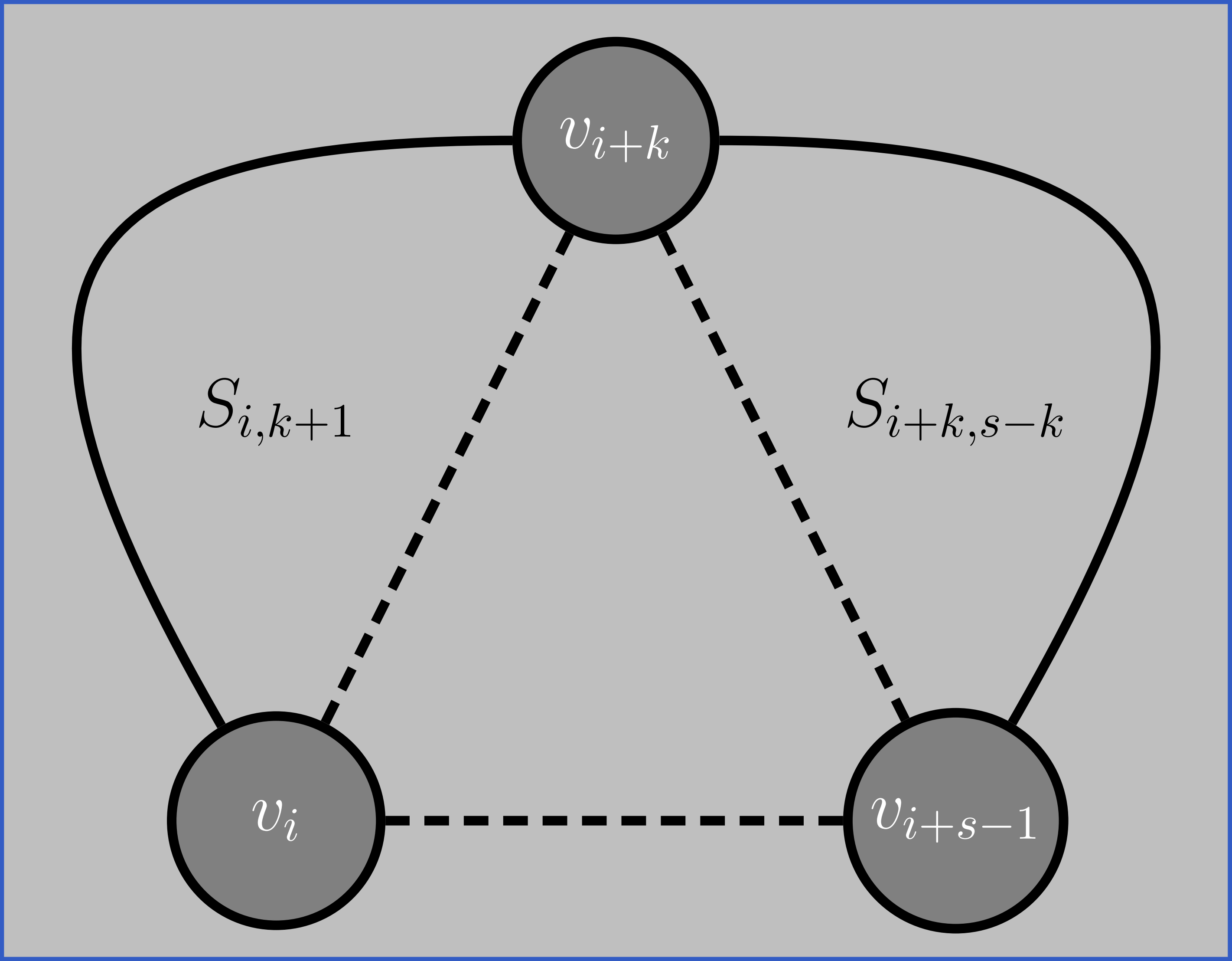
The Triangulation Problem (2)



The Triangulation Problem (3)

- Let S_{is} denote the subproblem of size s starting at vertex v_i of finding the minimum triangulation of the polygon $v_i, v_{i+1}, \ldots, v_{i+s-1}$ (counting modulo the number of vertices).
- Subproblems of size less than 4 are trivial.
- Solving S_{is} is done by solving $S_{i,k+1}$ and $S_{i+k,s-k}$ for all k, $1 \leq k \leq s-2$
- The obvious recursive formulation results in 3^{s-4} (non-trivial) calls.
- But for $n \geq 4$ vertices there are only $n(n-3)$ non-trivial subproblems!

The Triangulation Problem (4)



The Triangulation Problem (5)

- Let C_{is} denote the minimal triangulation cost of S_{is}.
- Let $D(v_p, v_q)$ denote the length of a chord between v_p and v_q (length is 0 for non-chords; i.e. adjacent v_p and v_q).
- For $s \geq 4$:

$$C_{is} = \min_{k \in [1, s-2]} \left\{ \begin{array}{l} C_{i,k+1} + C_{i+k,s-k} \\ +D(v_i, v_{i+k}) + D(v_{i+k}, v_{i+s-1}) \end{array} \right\}$$

- For $s < 4$, $C_{is} = 0$.

The Triangulation Problem (6)

These equations can be transliterated straight into Haskell:

```
triCost :: Polygon -> Double
triCost p = cost!(0,n) where
    cost = array ((0,0), (n-1,n))
                 ([ ((i,s),
                     minimum [ cost!(i, k+1)
                             + cost!((i+k) `mod` n, s-k)
                             + dist p i ((i+k) `mod` n)
                             + dist p ((i+k) `mod` n)
                                      ((i+s-1) `mod` n)
                            | k <- [1..s-2] ])
                  | i <- [0..n-1], s <- [4..n] ] ++
                  [ ((i,s), 0.0)
                  | i <- [0..n-1], s <- [0..3] ])
    n = snd (bounds b) + 1
```

Attribute Grammars (1)

Lazy evaluation is also very useful for evaluation of ***Attribute Grammars***:

- The attribution function is defined recursively over the tree:
  - takes inherited attributes as extra arguments;
  - returns a tuple of all synthesised attributes.
- As long as there exists ***some*** possible attribution order, lazy evaluation will take care of the attribute evaluation.

Attribute Grammars (2)

- The earlier examples on Circular Programming and Breadth-first Numbering can be seen as instances of this idea.

Reading

- John W. Lloyd. Practical advantages of declarative programming. In *Joint Conference on Declarative Programming, GULP-PRODE'94*, 1994.
- John Hughes. Why Functional Programming Matters. *The Computer Journal*, 32(2):98–197, April 1989.
- Thomas Johnsson. Attribute Grammars as a Functional Programming Paradigm. In *Functional Programming Languages and Computer Architecture, FPCA'87*, 1987

Reading

- Geraint Jones and Jeremy Gibbons.
  Linear-time breadth-first tree algorithms: An exercise in the arithmetic of folds and zips.
  Technical Report TR-31-92, Oxford University Computing Laboratory, 1992.
- Alfred Aho, John Hopcroft, Jeffrey Ullman.
  Data Structures and Algorithms.
  Addison-Wesley, 1983.

COMP4075/G54RFP: Lecture 4

Purely Functional Data Structures

Henrik Nilsson

University of Nottingham, UK

Purely Functional Data structures (1)

Purely functional data structures: What? Why?

Purely Functional Data structures (1)

Purely functional data structures: What? Why?

Standard implementations of many data structures rely on imperative update. But:

Purely Functional Data structures (1)

Purely functional data structures: What? Why?

Standard implementations of many data structures rely on imperative update. But:

- In a pure functional setting, we ***need*** pure alternatives.

Purely Functional Data structures (1)

Purely functional data structures: What? Why?

Standard implementations of many data structures rely on imperative update. But:

- In a pure functional setting, we ***need*** pure alternatives.
- In concurrent or distributed settings, side effects are not your friends. Purely functional structures can thus be very helpful!

Purely Functional Data structures (1)

Purely functional data structures: What? Why?

Standard implementations of many data structures rely on imperative update. But:

- In a pure functional setting, we ***need*** pure alternatives.
- In concurrent or distributed settings, side effects are not your friends. Purely functional structures can thus be very helpful!
- Generally interesting to explore different approaches.

Purely Functional Data structures (2)

Key difference:

Purely Functional Data structures (2)

Key difference:

- Imperative data structures are ***ephemeral***: a ***single copy*** gets mutated whenever the structure is updated.

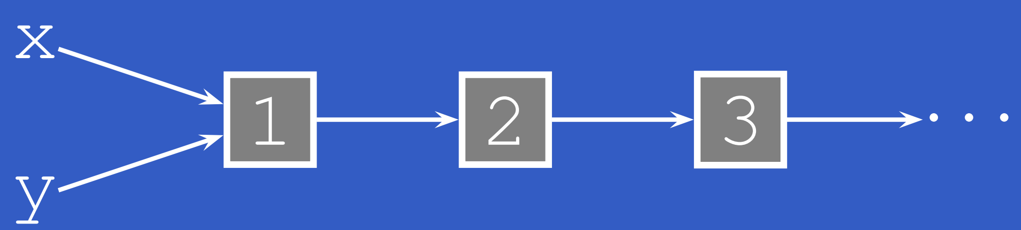
Purely Functional Data structures (2)

Key difference:

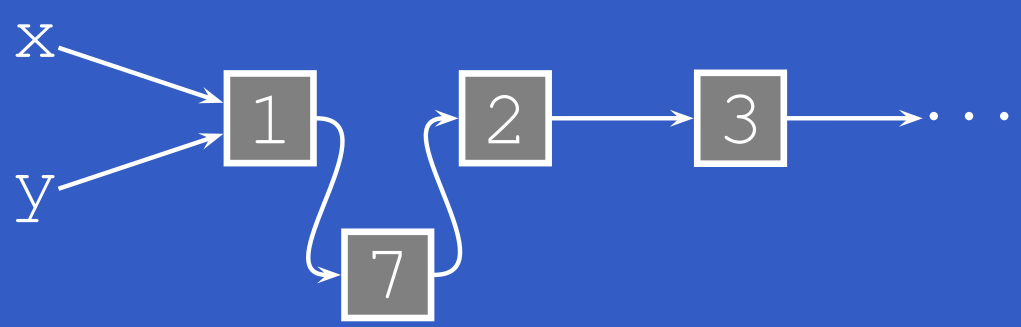
- Imperative data structures are ***ephemeral***: a ***single copy*** gets mutated whenever the structure is updated.
- Purely functional data structures are ***persistent***: a ***new copy*** is created whenever the structure is updated, leaving old copies intact. (Common sub-parts can be shared.)

Purely Functional Data structures (3)

Linked list:

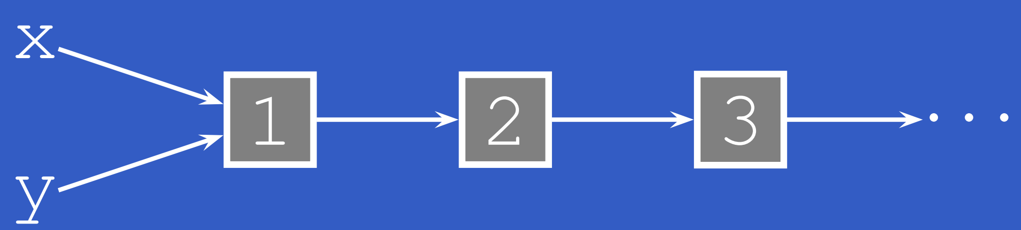


After insert, if ephemeral:

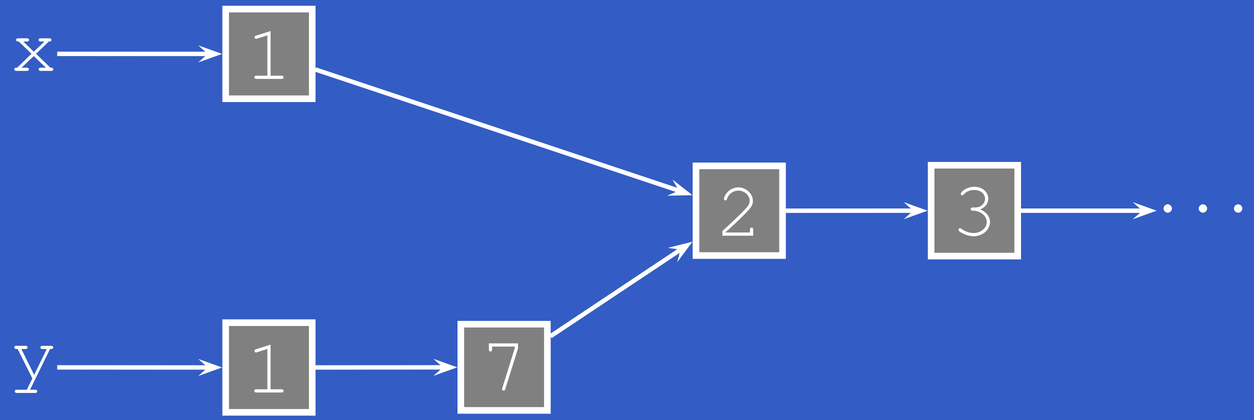


Purely Functional Data structures (4)

Linked list:



After insert, if persistent:



Purely Functional Data structures (5)

This lecture draws from:

> Chris Okasaki. *Purely Functional Data Structures*. Cambridge University Press, 1998.

We will look at some examples of how ***numerical representations*** can be used to derive purely functional data structures.

Numerical Representations (1)

Strong analogy between lists and the usual representation of natural numbers:

```
data List a =                        data Nat =
      Nil                                 Zero
    | Cons a (List a)                   | Succ Nat

tail (Cons _ xs) = xs                pred (Succ n) = n

append Nil          ys = ys          plus Zero n     = n
append (Cons x xs) ys =              plus (Succ m) n =
    Cons x (append xs ys)                Succ (plus m n)
```

Numerical Representations (2)

This analogy can be taken further for designing ***container*** structures because:

- inserting an element resembles incrementing a number
- combining two containers resembles adding two numbers

etc.

Thus, representations of natural numbers with certain properties induce container types with similar properties. Called ***Numerical Representations***.

Random Access Lists

We will consider ***Random Access Lists*** in the following. Signature:

```
data RList a

empty    :: RList a
isEmpty  :: RList a -> Bool
cons     :: a -> RList a -> RList a
head     :: RList a -> a
tail     :: RList a -> RList a
lookup   :: Int -> RList a -> a
update   :: Int -> a -> RList a -> RList a
```

Positional Number Systems (1)

- A number is written as a ***sequence*** of ***digits*** $b_0 b_1 \ldots b_{m-1}$, where $b_i \in D_i$ for a fixed family of digit sets given by the positional system.
- b_0 is the ***least significant*** digit, b_{m-1} the ***most significant*** digit (note the ordering).
- Each digit b_i has a ***weight*** w_i. Thus:

$$\text{value}(b_0 b_1 \ldots b_{m-1}) = \sum_{0}^{m-1} b_i w_i$$

where the fixed sequence of weights w_i is given by the positional system.

Positional Number Systems (2)

- A number is written written in ***base*** B if $w_i = B^i$ and $D_i = \{0, \ldots, B-1\}$.
- The sequence w_i is usually, but not necessarily, increasing.
- A number system is ***redundant*** if there is more than one way to represent some numbers (disallowing trailing zeroes).
- A representation of a positional number system can be ***dense***, meaning including zeroes, or ***sparse***, eliding zeroes.

Exercise 1: Positional Number Systems

Suppose $w_i = 2^i$ and $D_i = \{0, 1, 2\}$. Give three different ways to represent 17.

Exercise 1: Solution

- 10001, since $\mathrm{value}(10001) = 1 \cdot 2^0 + 1 \cdot 2^4$
- 1002, since $\mathrm{value}(1002) = 1 \cdot 2^0 + 2 \cdot 2^3$
- 1021, since $\mathrm{value}(1021) = 1 \cdot 2^0 + 2 \cdot 2^2 + 1 \cdot 2^3$
- 1211, since
  $\mathrm{value}(1211) = 1 \cdot 2^0 + 2 \cdot 2^1 + 1 \cdot 2^2 + 1 \cdot 2^3$

From Positional System to Container

Given a positional system, a numerical representation may be derived as follows:

- for a container of size n, consider a representation $b_0 b_1 \ldots b_{m-1}$ of n,
- represent the collection of n elements by a ***sequence of trees*** of size w_i such that there are b_i trees of that size.

For example, given the positional system of exercise 1, a container of size 17 might be represented by 1 tree of size 1, 2 trees of size 2, 1 tree of size 4, and 1 tree of size 8.

What Kind of Trees?

The kind of tree should be chosen depending on needed sizes and properties. Two possibilities:

- ***Complete Binary Leaf Trees***

```
data Tree a = Leaf a
            | Node (Tree a) (Tree a)
```

  Sizes: $2^n, n \geq 0$

- ***Complete Binary Trees***

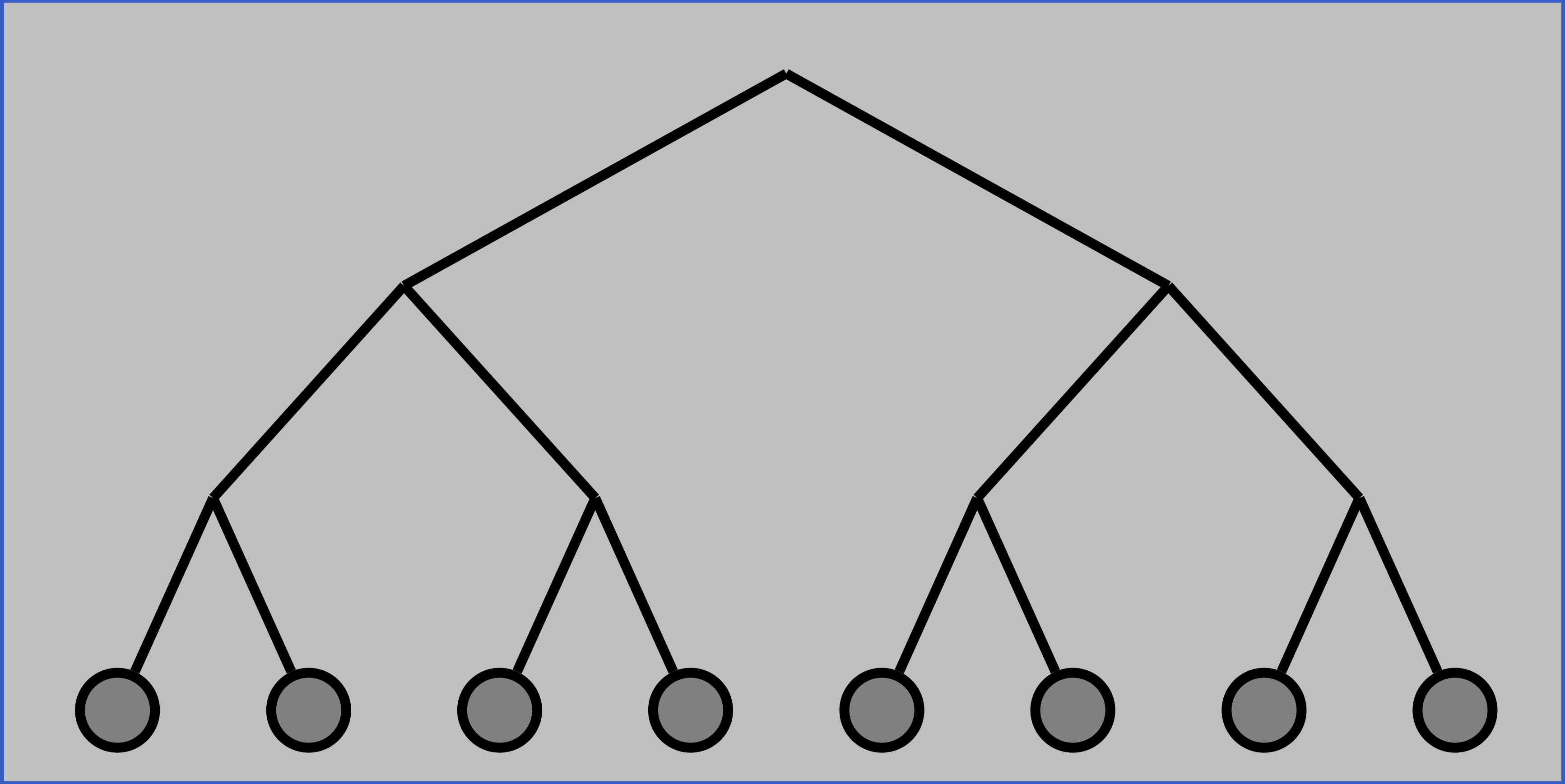
```
data Tree a = Leaf a
            | Node (Tree a) a (Tree a)
```

  Sizes: $2^{n+1} - 1, n \geq 0$

(Balance has to be ensured separately.)

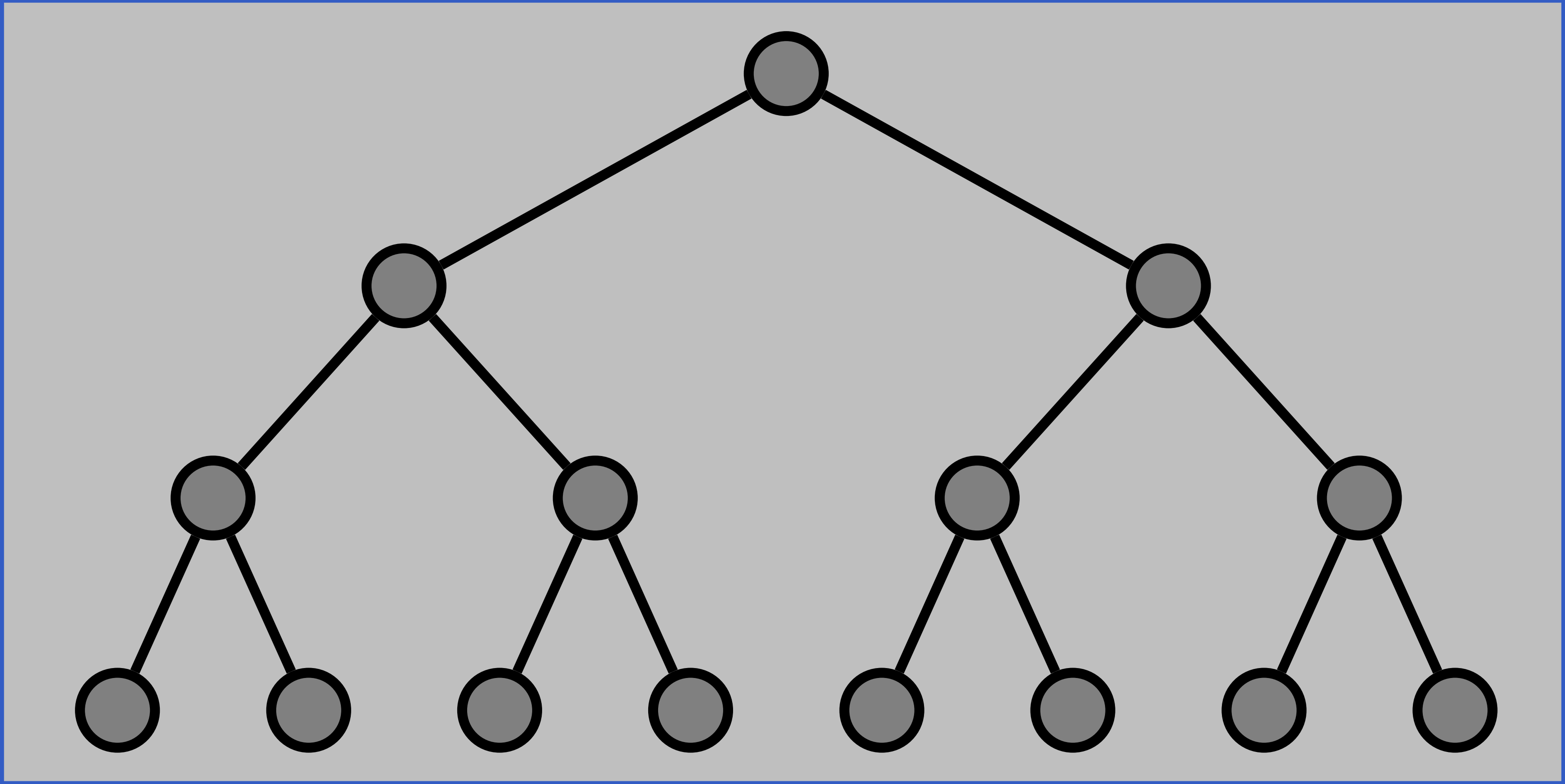
Example: Complete Binary Leaf Tree

Size $2^3 = 8$:



Example: Complete Binary Tree

Size $2^4 - 1 = 15$:



Binary Random Access Lists (1)

Binary Random Access Lists are induced by

- the usual binary representation, i.e. $w_i = 2^i$, $D_i = \{0, 1\}$
- complete binary leaf trees

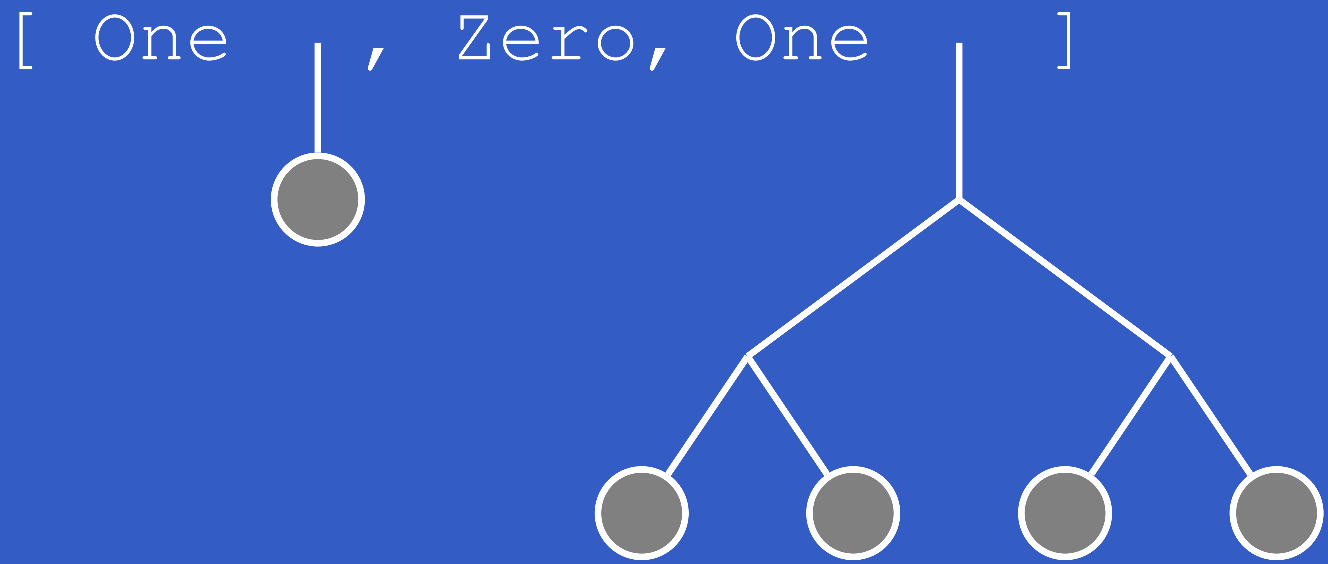
Thus:

```
data Tree a = Leaf a
            | Node Int (Tree a) (Tree a)
data Digit a = Zero | One (Tree a)
type RList a = [Digit a]
```

The `Int` field keeps track of tree size for speed.

Binary Random Access Lists (2)

Example: Binary Random Access List of size 5:



Binary Random Access Lists (3)

The increment function on dense binary numbers:

```
inc []          = [One]
inc (Zero : ds) = One : ds
inc (One  : ds) = Zero : inc ds   -- Carry
```

Binary Random Access Lists (4)

Inserting an element first in a binary random access list is analogous to `inc`:

```
cons :: a -> RList a -> RList a
cons x ts = consTree (Leaf x) ts

consTree :: Tree a -> RList a -> RList a
consTree t []                = [One t]
consTree t (Zero : ts)       = (One t : ts)
consTree t (One t' : ts) =
    Zero : consTree (link t t') ts
```

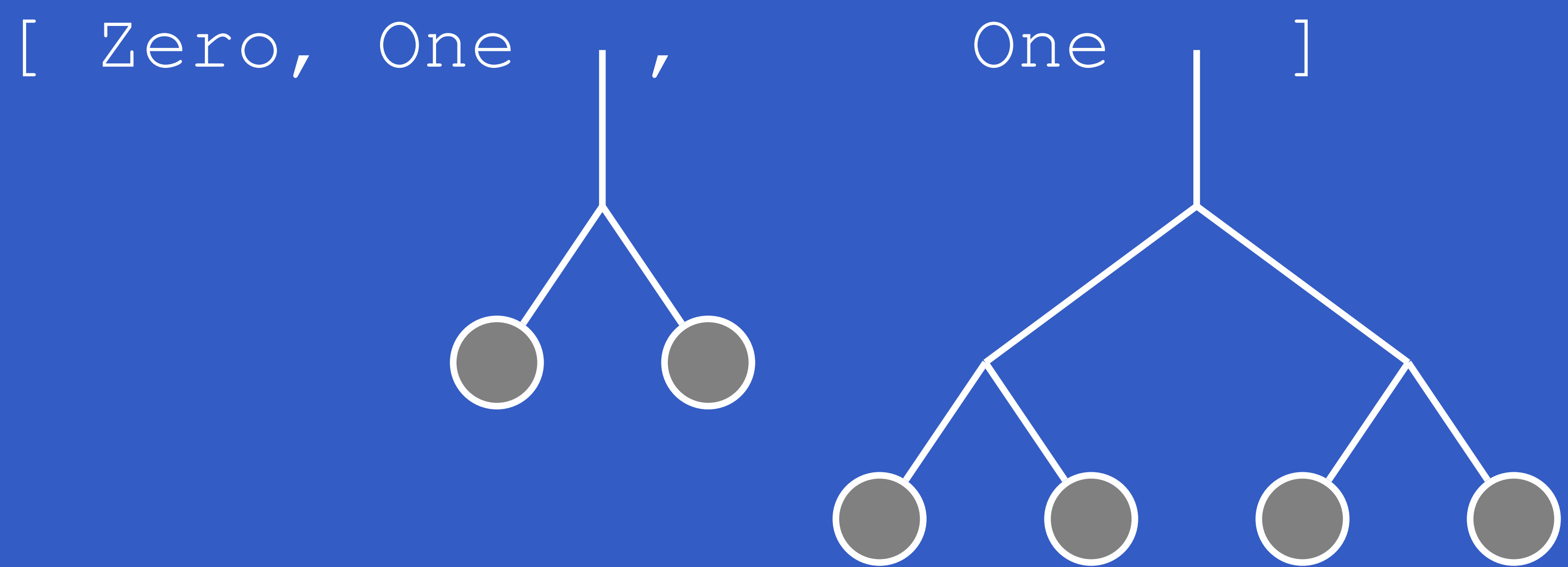
Binary Random Access Lists (5)

The utility function `link` joins two equally sized trees:

```
-- t1 and t2 are assumed to be the same size
link t1 t2 = Node (2 * size t1) t1 t2
```

Binary Random Access Lists (6)

Example: Result of consing element onto list of size 5:



Binary Random Access Lists (7)

Time complexity:

- `cons`, `head`, `tail`, perform $O(1)$ work per digit, thus $O(\log n)$ worst case.
- `lookup` and `update` take $O(\log n)$ to find the right tree, and then $O(\log n)$ to find the right element in that tree, so $O(\log n)$ worst case overall.

Binary Random Access Lists (7)

Time complexity:

- `cons`, `head`, `tail`, perform $O(1)$ work per digit, thus $O(\log n)$ worst case.
- `lookup` and `update` take $O(\log n)$ to find the right tree, and then $O(\log n)$ to find the right element in that tree, so $O(\log n)$ worst case overall.

Time complexity for `cons`, `head`, `tail` disappointing: can we do better?

Skew Binary Numbers (1)

Skew Binary Numbers:

- $w_i = 2^{i+1} - 1$ (rather than 2^i)
- $D_i = \{0, 1, 2\}$

Representation is redundant. But we obtain a ***canonical form*** if we insist that only the least significant non-zero digit may be 2.

Note: The weights correspond to the sizes of ***complete*** binary trees.

Skew Binary Numbers (2)

Theorem: Every natural number n has a unique skew binary canonical form.

Proof sketch. By induction on n.

- Base case: the case for 0 is direct.

Skew Binary Numbers (3)

- Inductive case. Assume n has a unique skew binary representation $b_0 b_1 \ldots b_{m-1}$

Skew Binary Numbers (3)

- Inductive case. Assume n has a unique skew binary representation $b_0 b_1 \ldots b_{m-1}$
  - If the least significant non-zero digit is smaller than 2, then $n + 1$ has a unique skew binary representation obtained by adding 1 to the least significant digit b_0.

Skew Binary Numbers (3)

- Inductive case. Assume n has a unique skew binary representation $b_0 b_1 \ldots b_{m-1}$
  - If the least significant non-zero digit is smaller than 2, then $n+1$ has a unique skew binary representation obtained by adding 1 to the least significant digit b_0.
  - If the least significant non-zero digit b_i is 2, then note that $1 + 2(2^{i+1} - 1) = 2^{i+2} - 1$. Thus $n+1$ has a unique skew binary representation obtained by setting b_i to 0 and adding 1 to b_{i+1}.

Exercise 2: Skew Binary Numbers

Give the canonical skew binary representation for 31, 30, 29, and 28.

Exercise 2: Skew Binary Numbers

Give the canonical skew binary representation for 31, 30, 29, and 28.

Solution: 00001, 0002, 0021, 0211

Inc. Sparse Skew Binary Number

Assume a ***sparse*** skew binary representation of the natural numbers `type Nat = [Int]`, where the integers represent the ***weight*** of each ***non-zero*** digit, in increasing order, except that the first two may be equal indicating smallest non-zero digit is 2.

Function to increment a number:

```
inc :: Nat -> Nat
inc (w1 : w2 : ws)
    | w1 == w2 = w1 * 2 + 1 : ws
inc ws         = 1 : ws
```

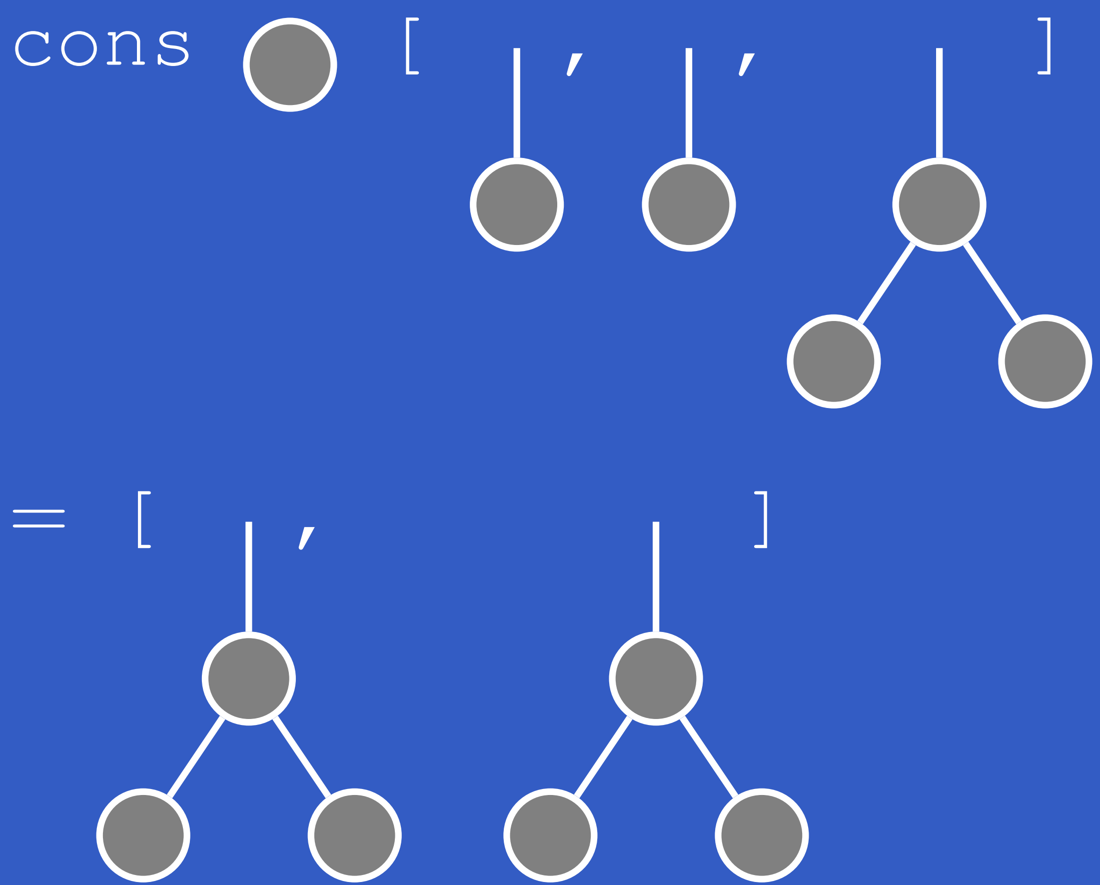
Note: Constant time operation!

Skew Binary Random Access Lists (1)

```
data Tree a = Leaf a | Node (Tree a) a (Tree a)
type RList a = [(Int, Tree a)]

empty :: RList a
empty = []

cons :: a -> RList a -> RList a
cons x ((w1, t1) : (w2, t2) : wts) | w1 == w2 =
     (w1 * 2 + 1, Node t1 x t2) : wts
cons x wts = ((1, Leaf x) : wts)
```

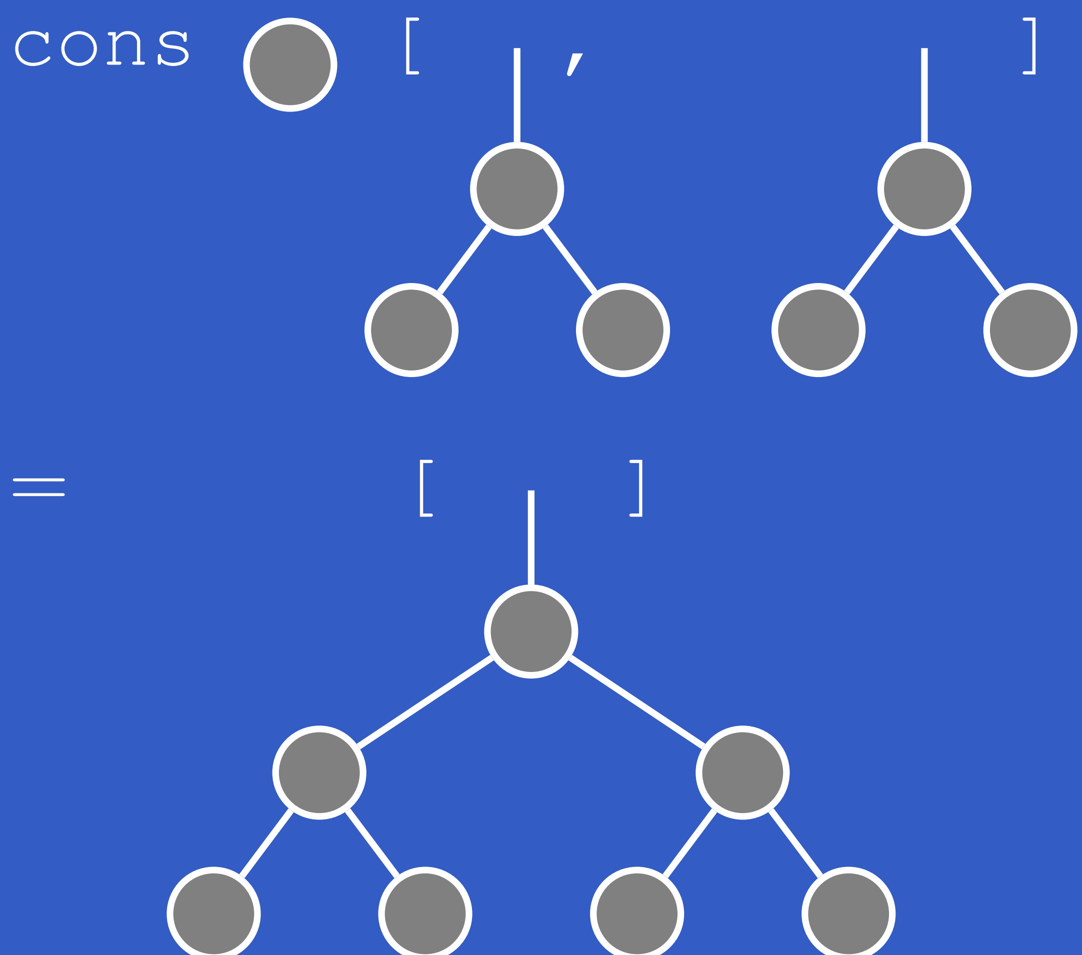
Skew Binary Random Access Lists (2)

Example: Consing onto list of size 5:



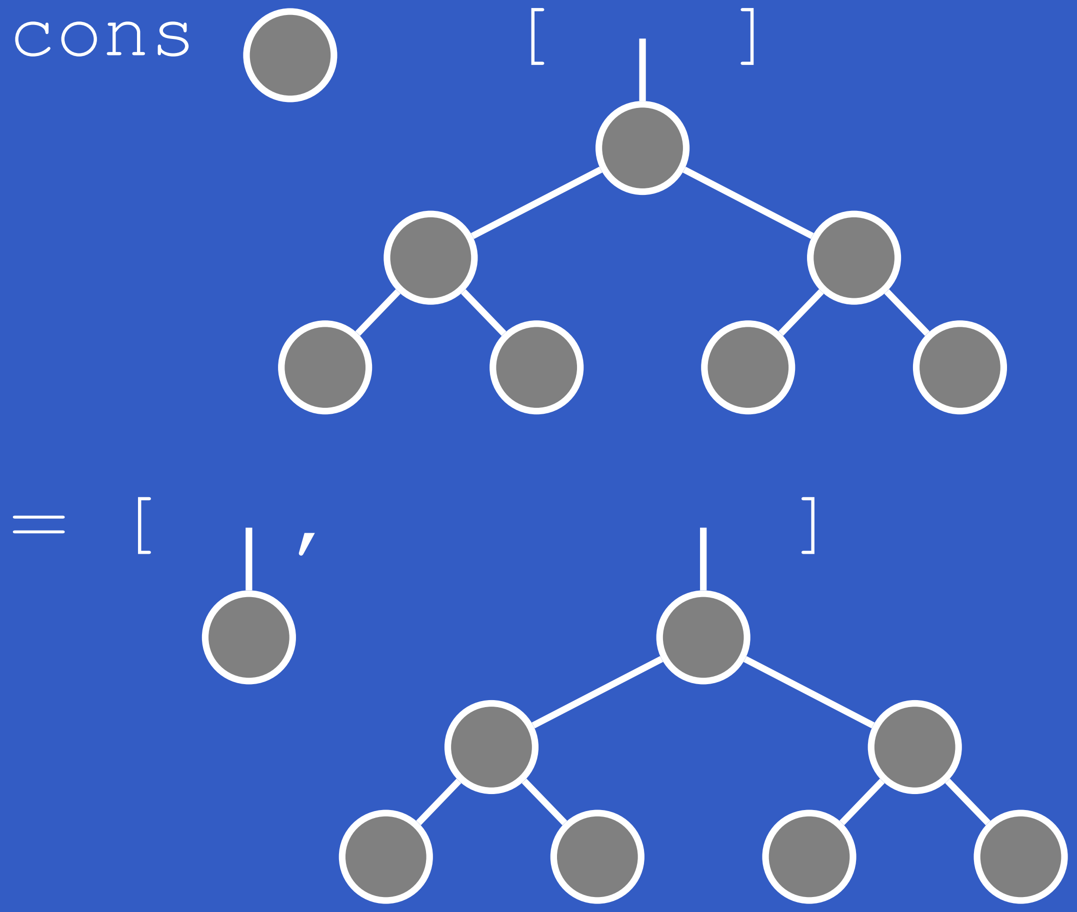
Skew Binary Random Access Lists (3)

Example: Consing onto list of size 6:



Skew Binary Random Access Lists (4)

Example: Consing onto list of size 7:



Skew Binary Random Access Lists (5)

```
head :: RList a -> a
head ((_, Leaf x)         : _) = x
head ((_, Node _ x _) : _) = x

tail :: RList a -> RList a
tail ((_, Leaf _): wts) = wts
tail ((w, Node t1 _ t2) : wts) =
    (w', t1) : (w', t2) : wts
    where
        w' = w `div` 2
```

Note: partial operations.

Skew Binary Random Access Lists (6)

```
lookup :: Int -> RList a -> a
lookup i ((w, t) : wts)
    | i < w         = lookupTree i w t
    | otherwise     = lookup (i - w) wts
lookupTree :: Int -> Int -> Tree a -> a
lookupTree _ _ (Leaf x) = x
lookupTree i w (Node t1 x t2)
    | i == 0       = x
    | i <= w'      = lookupTree (i - 1) w' t1
    | otherwise = lookupTree (i - w' - 1) w' t2
    where
        w' = w `div` 2
```

Skew Binary Random Access Lists (7)

Time complexity:

- `cons`, `head`, `tail`: $O(1)$.
- `lookup` and `update` take $O(\log n)$ to find the right tree, and then $O(\log n)$ to find the right element in that tree, so $O(\log n)$ worst case overall.

Skew Binary Random Access Lists (7)

Time complexity:

- `cons`, `head`, `tail`: $O(1)$.
- `lookup` and `update` take $O(\log n)$ to find the right tree, and then $O(\log n)$ to find the right element in that tree, so $O(\log n)$ worst case overall.

Okasaki:

> "Although there are better implementations of lists, and better implementations of (persistent) arrays, none are better at both."

COMP4075/G54RFP: Lecture 5

Type Classes

Henrik Nilsson

University of Nottingham, UK

Type Classes

- Type classes is one of the distinguishing fetures of Haskell

Type Classes

- Type classes is one of the distinguishing fetures of Haskell
- Introduced to make ad hoc polymorphism, or overloading, less ad hoc

Type Classes

- Type classes is one of the distinguishing fetures of Haskell
- Introduced to make ad hoc polymorphism, or overloading, less ad hoc
- Promotes reuse, making code more readable

Type Classes

- Type classes is one of the distinguishing fetures of Haskell
- Introduced to make ad hoc polymorphism, or overloading, less ad hoc
- Promotes reuse, making code more readable
- Central to elimination of all kinds of “boiler-plate” code and sophisticated datatype-generic programming.

Type Classes

- Type classes is one of the distinguishing fetures of Haskell
- Introduced to make ad hoc polymorphism, or overloading, less ad hoc
- Promotes reuse, making code more readable
- Central to elimination of all kinds of “boiler-plate” code and sophisticated datatype-generic programming.

Key reason why many practitioners like Haskell: lots of “programming” can happen automatically!

Haskell Overloading (1)

What is the type of $(==)$?

E.g. the following both work:

$$1 == 2$$

$$'\mathtt{a}' == '\mathtt{b}'$$

I.e., $(==)$ can be used to compare both numbers and characters.

Haskell Overloading (1)

What is the type of $(==)$?

E.g. the following both work:

$$1 == 2$$

$$\texttt{'a'} == \texttt{'b'}$$

I.e., $(==)$ can be used to compare both numbers and characters.

Maybe $(==) :: a \to a \to Bool$?

Haskell Overloading (1)

What is the type of $(==)$?

E.g. the following both work:

$$1 == 2$$

$$\texttt{'a'} == \texttt{'b'}$$

I.e., $(==)$ can be used to compare both numbers and characters.

Maybe $(==) :: a \to a \to Bool$?

No!!! Cannot work uniformly for arbitrary types!

Haskell Overloading (2)

A function like the identity function

$$id :: a \to a$$
$$id\ x = x$$

is ***polymorphic*** precisely because it works uniformly for all types: there is no need to "inspect" the argument.

Haskell Overloading (2)

A function like the identity function

$$id :: a \to a$$
$$id\ x = x$$

is ***polymorphic*** precisely because it works uniformly for all types: there is no need to "inspect" the argument.

In contrast, to compare two "things" for equality, they very much have to be inspected, and an ***appropriate method of comparison*** needs to be used.

Haskell Overloading (3)

Moreover, some types do not in general admit a decidable equality. E.g. functions (when their domain is infinite).

Haskell Overloading (3)

Moreover, some types do not in general admit a decidable equality. E.g. functions (when their domain is infinite).

Similar remarks apply to many other types. E.g.:

Haskell Overloading (3)

Moreover, some types do not in general admit a decidable equality. E.g. functions (when their domain is infinite).

Similar remarks apply to many other types. E.g.:

- We may want to be able to add numbers of any kind.

Haskell Overloading (3)

Moreover, some types do not in general admit a decidable equality. E.g. functions (when their domain is infinite).

Similar remarks apply to many other types. E.g.:

- We may want to be able to add numbers of any kind.
- But to add properly, we must understand what we are adding.

Haskell Overloading (3)

Moreover, some types do not in general admit a decidable equality. E.g. functions (when their domain is infinite).

Similar remarks apply to many other types. E.g.:

- We may want to be able to add numbers of any kind.
- But to add properly, we must understand what we are adding.
- Not every type admits addition.

Haskell Overloading (4)

Idea:

Haskell Overloading (4)

Idea:

- Introduce the notion of a ***type class***: a set of types that support certain related operations.

Haskell Overloading (4)

Idea:

- Introduce the notion of a ***type class***: a set of types that support certain related operations.
- ***Constrain*** those operations to ***only*** work for types belonging to the corresponding class.

Haskell Overloading (4)

Idea:

- Introduce the notion of a ***type class***: a set of types that support certain related operations.
- ***Constrain*** those operations to ***only*** work for types belonging to the corresponding class.
- Allow a type to be ***made an instance of*** (added to) a type class by providing ***type-specific implementations*** of the operations of the class.

The Type Class Eq

$$
\begin{array}{l}
\textbf{class } Eq \; a \textbf{ where} \\
\quad (==) :: a \to a \to Bool
\end{array}
$$

$(==)$ is not a function, but a ***method*** of the ***type class*** Eq. It's type signature is:

$$(==) :: Eq \; a \Rightarrow a \to a \to Bool$$

$Eq \; a$ is a ***class constraint***. It says that that the equality method works for any type belonging to the type class Eq.

Instances of Eq (1)

Various types can be made instances of a type class like Eq by providing implementations of the class methods for the type in question:

$$\textbf{instance}\ Eq\ Int\ \textbf{where}$$
$$\quad x == y = primEqInt\ x\ y$$
$$\textbf{instance}\ Eq\ Char\ \textbf{where}$$
$$\quad x == y = primEqChar\ x\ y$$

Instances of Eq (2)

Suppose we have a data type:

$$\mathbf{data}\ Answer = Yes \mid No \mid Unknown$$

We can make $Answer$ an instance of Eq as follows:

$$\begin{array}{l}\mathbf{instance}\ Eq\ Answer\ \mathbf{where}\\ \quad \begin{array}{lll} Yes & == Yes & = True\\ No & == No & = True\\ Unknown & == Unknown & = True\\ \_ & == \_ & = False\end{array}\end{array}$$

Instances of Eq (3)

Consider:

$$
\begin{aligned}
\mathbf{data}\ \mathit{Tree}\ a &= \mathit{Leaf}\ a \\
&\mid \mathit{Node}\ (\mathit{Tree}\ a)\ (\mathit{Tree}\ a)
\end{aligned}
$$

Can Tree be made an instance of Eq?

Instances of Eq (4)

Yes, for any type a that is already an instance of Eq:

$$
\begin{array}{llll}
\multicolumn{4}{l}{\textbf{instance}\ (Eq\ a) \Rightarrow Eq\ (Tree\ a)\ \textbf{where}} \\
\quad Leaf\ a1 & == Leaf\ a2 & = a1 == a2 \\
\quad Node\ t1l\ t1r & == Node\ t2l\ t2r & = t1l == t2l \\
 & & \quad \&\&\ t1r == t2r \\
\quad \_ & == \_ & = False
\end{array}
$$

Note that $(==)$ is used at type a (whatever that is) when comparing $a1$ and $a2$, while the use of $(==)$ for comparing subtrees is a recursive call.

Derived Instances (1)

Instance declarations are often obvious and mechanical. Thus, for certain ***built-in*** classes (notably Eq, Ord, $Show$), Haskell provides a way to ***automatically derive*** instances, as long as

- the data type is sufficiently simple
- we are happy with the standard definitions

Thus, we can do:

$$
\begin{aligned}
&\mathbf{data}\ \mathit{Tree}\ a = \mathit{Leaf}\ a \\
&\qquad\qquad\qquad \mid \mathit{Node}\ (\mathit{Tree}\ a)\ (\mathit{Tree}\ a) \\
&\qquad\qquad\quad \mathbf{deriving}\ \mathit{Eq}
\end{aligned}
$$

Derived Instances (2)

GHC provides ***many*** additional possibilities. With the extension `-XGeneralizedNewtypeDeriving`, a new type defined using **newtype** can “inherit” any of the instances of the representation type:

$$\mathbf{newtype}\ Time = Time\ Int\ \mathbf{deriving}\ Num$$

Derived Instances (2)

GHC provides ***many*** additional possibilities. With the extension `-XGeneralizedNewtypeDeriving`, a new type defined using **newtype** can “inherit” any of the instances of the representation type:

$$\mathbf{newtype}\ \mathit{Time} = \mathit{Time}\ \mathit{Int}\ \mathbf{deriving}\ \mathit{Num}$$

With the extension `-XStandaloneDeriving`, instances can be derived separately from a type definition (even in a separate module):

$$\mathbf{deriving\ instance}\ \mathit{Eq}\ \mathit{Time}$$
$$\mathbf{deriving\ instance}\ \mathit{Eq}\ a \Rightarrow \mathit{Eq}\ (\mathit{Tree}\ a)$$

Class Hierarchy

Type classes form a hierarchy. E.g.:

$$\textbf{class } Eq\ a \Rightarrow Ord\ a \textbf{ where}$$
$$\quad (<=) :: a \rightarrow a \rightarrow Bool$$
$$\quad \ldots$$

Eq is a superclass of Ord; i.e., any type in Ord must also be in Eq.

Haskell vs. OO Overloading (1)

A method, or overloaded function, may thus be understood as a family of functions where the right one is chosen depending on the types.

A bit like OO languages like Java. But the underlying mechanism is quite different and much more general. Consider `read`:

$$read :: (Read\ a) \Rightarrow String \rightarrow a$$

Note: overloaded on the ***result*** type! A method that converts from a string to ***any*** other type in class `Read`!

Haskell vs. OO Overloading (2)

$> \textbf{let } xs = [1,2,3] :: [Int]$

$> \textbf{let } ys = [1,2,3] :: [Double]$

$> xs$

$[1,2,3]$

$> ys$

$[1.0,2.0,3.0]$

$> (read \texttt{"42"} : xs)$

$[42,1,2,3]$

$> (read \texttt{"42"} : ys)$

$[42.0,1.0,2.0,3.0]$

Implementation (1)

The class constraints represent extra implicit arguments that are filled in by the compiler. These arguments are (roughly) the functions to use.

Thus, internally $(==)$ is a ***higher order function*** with ***three*** arguments:

$$(==)\ eqF\ x\ y = eqF\ x\ y$$

Implementation (2)

An expression like

$$1 == 2$$

is essentially translated into

$$(==)\ primEqInt\ 1\ 2$$

Implementation (3)

So one way of understanding a type like

$$(==) :: Eq\ a \Rightarrow a \rightarrow a \rightarrow Bool$$

is that $Eq\ a$ corresponds to an extra implicit argument.

The implicit argument corresponds to a so called directory, or tuple/record of functions, one for each method of the type class in question.

Some Basic Haskell Classes (1)

$$
\begin{array}{l}
\textbf{class } Eq\ a \textbf{ where} \\
\quad (==), (/=) :: a \rightarrow a \rightarrow Bool \\
\textbf{class } (Eq\ a) \Rightarrow Ord\ a \textbf{ where} \\
\quad compare \ :: a \rightarrow a \rightarrow Ordering \\
\quad (<), (<=), (>=), (>) :: a \rightarrow a \rightarrow Bool \\
\quad max, min :: a \rightarrow a \rightarrow a \\
\textbf{class } Show\ a \textbf{ where} \\
\quad show :: a \rightarrow String
\end{array}
$$

Some Basic Haskell Classes (2)

$$
\begin{array}{l}
\textbf{class}\ Num\ a\ \textbf{where} \\
\quad (+),(-),(*) :: a \rightarrow a \rightarrow a \\
\quad negate \qquad\quad :: a \rightarrow a \\
\quad abs, signum :: a \rightarrow a \\
\quad fromInteger :: Integer \rightarrow a
\end{array}
$$

Some Basic Haskell Classes (2)

$$
\begin{array}{l}
\textbf{class}\ Num\ a\ \textbf{where} \\
\quad (+), (-), (*) :: a \rightarrow a \rightarrow a \\
\quad negate \qquad\quad :: a \rightarrow a \\
\quad abs, signum :: a \rightarrow a \\
\quad fromInteger :: Integer \rightarrow a
\end{array}
$$

$$
\begin{array}{l}
\textbf{class}\ Num\ a \Rightarrow Fractional\ a\ \textbf{where} \\
\quad (/) \quad\ :: a \rightarrow a \rightarrow a \\
\quad recip :: a \rightarrow a \\
\quad fromRational :: Rational \rightarrow a
\end{array}
$$

Some Basic Haskell Classes (3)

Quiz: What is the type of a numeric literal like 42?

Some Basic Haskell Classes (3)

Quiz: What is the type of a numeric literal like 42? What about 1.23? Why?

Some Basic Haskell Classes (3)

Quiz: What is the type of a numeric literal like 42? What about 1.23? Why?

Haskell's numeric literals are overloaded:

- 42 means $\mathit{fromInteger}\ 42$
- 1.23 means $\mathit{fromRational}\ (133\ \%\ 100)$

Some Basic Haskell Classes (3)

Quiz: What is the type of a numeric literal like 42? What about 1.23? Why?

Haskell's numeric literals are overloaded:

- 42 means $\mathit{fromInteger}\ 42$
- 1.23 means $\mathit{fromRational}\ (133\ \%\ 100)$

Thus:

$$\begin{aligned} 42 &:: \mathit{Num}\ a \Rightarrow a \\ 1.23 &:: \mathit{Fractional}\ a \Rightarrow a \end{aligned}$$

A Typing Conundrum (1)

Overloaded (numeric) literals can lead to some surprises.

What is the type of the following list? Is it even well-typed???

$$[1, [2, 3]]$$

A Typing Conundrum (1)

Overloaded (numeric) literals can lead to some surprises.

What is the type of the following list? Is it even well-typed???

$$[1, [2, 3]]$$

Surprisingly, it is well-typed:

$$> \text{:}\mathbf{type}\ [1, [2, 3]]$$
$$[1, [2, 3]] :: (Num\ [t], Num\ t) \Rightarrow [[t]]$$

Why?

A Typing Conundrum (2)

The list is expanded into:

$$[\mathit{fromInteger}\ 1,$$
$$\ [\mathit{fromInteger}\ 2, \mathit{fromInteger}\ 3]]$$

Thus, if there were some type t for which $[t]$ were an instance of Num, the 1 would be an overloaded literal of that type, matching the type of the second element of the list.

A Typing Conundrum (2)

The list is expanded into:

$$[\mathit{fromInteger}\ 1,\\ \ [\mathit{fromInteger}\ 2, \mathit{fromInteger}\ 3]]$$

Thus, if there were some type t for which $[t]$ were an instance of Num, the 1 would be an overloaded literal of that type, matching the type of the second element of the list.

Normally there are no such instances, so what almost certainly is a mistake will be caught. But the error message is rather confusing.

Multi-parameter Type Classes

GHC supports an extension to allow a class to have more than one parameter; i.e., defining a ***relation*** on types rather than just a predicate:

$$\textbf{class}\ C\ a\ b\ \textbf{where} \ldots$$

Multi-parameter Type Classes

GHC supports an extension to allow a class to have more than one parameter; i.e., defining a ***relation*** on types rather than just a predicate:

$$\mathbf{class}\ C\ a\ b\ \mathbf{where} \ldots$$

This often lead to type inference ambiguities. Can be addressed through ***functional dependencies***:

$$\mathbf{class}\ \mathit{StateMonad}\ s\ m \mid m \to s\ \mathbf{where} \ldots$$

This enforces that all instances will be such that m uniquely determines s.

Application: Automatic Differentiation

- ***Automatic Differentiation***: method for augmenting code so that derivative(s) computed along with main result.
- Purely algebraic method: arbitrary code can be handled
- Exact results
- But no separate, self-contained representation of the derivative.

Automatic Differentiation: Key Idea

Consider a code fragment:

$$
\begin{aligned}
z1 &= x + y \\
z2 &= x * z1
\end{aligned}
$$

Automatic Differentiation: Key Idea

Consider a code fragment:

$$z1 = x + y$$
$$z2 = x * z1$$

Suppose x' and y' are the derivatives of x and y w.r.t. a common variable. Then the code can be augmented to compute the derivatives of $z1$ and $z2$:

$$z1 \ = x + y$$
$$z1' = x' + y'$$
$$z2 \ = x * z1$$
$$z2' = x' * z1 + x * z1'$$

Approaches

- Source-to-source translation
- Overloading of arithmetic operators and mathematical functions

The following variation is due to Jerzy Karczmarczuk. Infinite list of derivatives allows derivatives of ***arbitrary*** order to be computed.

Functional Automatic Differentiation (1)

Introduce a new numeric type C: value of a continuously differentiable function at a point along with ***all*** derivatives at that point:

$$\textbf{data}\ C = C\ Double\ C$$

$$valC\ (C\ \ a\ \_)\ = a$$
$$derC\ (C\ \_\ x')= x'$$

Functional Automatic Differentiation (2)

Constants and the variable of differentiation:

$$zeroC :: C$$
$$zeroC = C\ 0.0\ zeroC$$

$$constC :: Double \rightarrow C$$
$$constC\ \ a = C\ \ a\ zeroC$$

$$dVarC :: Double \rightarrow C$$
$$dVarC\ \ a = C\ \ a\ (constC\ 1.0)$$

Functional Automatic Differentiation (3)

Part of numerical instance:

$$
\begin{array}{l}
\textbf{instance } \mathit{Num}\ C \textbf{ where} \\
\quad (C\ a\ x') + (C\ b\ y') = C\ (a + b)\ (x' + y') \\
\quad (C\ a\ x') - (C\ b\ y') = C\ (a - b)\ (x' - y') \\
\quad x@(C\ a\ x') * y@(C\ b\ y') = \\
\quad\quad C\ (a * b)\ (x' * y + x * y') \\
\quad \mathit{fromInteger}\ n = \mathit{constC}\ (\mathit{fromInteger}\ n)
\end{array}
$$

Functional Automatic Differentiation (4)

Computation of $y = 3t^2 + 7$ at $t = 2$:

$$t = dVarC\ 2$$
$$y = 3 * t * t + 7$$

We can now get whichever derivatives we need:

$$\begin{array}{lcl} valC\ y & \Rightarrow & 19.0 \\ valC\ (derC\ y) & \Rightarrow & 12.0 \\ valC\ (derC\ (derC\ y)) & \Rightarrow & 6.0 \\ valC\ (derC\ (derC\ (derC\ y))) & \Rightarrow & 0.0 \end{array}$$

Functional Automatic Differentiation (5)

Of course, we're not limited to picking just one point. Let $tvals$ be a list of points of interest:

$$[3 * t * t + 7 \mid tval \leftarrow tvals, \mathbf{let}\ t = dVarC\ \ tval]$$

Functional Automatic Differentiation (5)

Of course, we're not limited to picking just one point. Let $tvals$ be a list of points of interest:

$$[3 * t * t + 7 \mid tval \leftarrow tvals, \textbf{let } t = dVarC\ tval]$$

Or we can define a function:

$$\begin{aligned}
&y :: Double \rightarrow C \\
&y\ tval = 3 * t * t + 7 \\
&\quad \textbf{where} \\
&\qquad t = dVarC\ tval
\end{aligned}$$

Reading

- Jerzy Karczmarczuk. Functional differentiation of computer programs. *Higher-Order and Symbolic Computation*, 14(1):35–57, March 2001.

COMP4075/G54RFP: Lecture 6

Functional Programming Patterns: Functor, Foldable, and Friends

Henrik Nilsson

University of Nottingham, UK

Type Classes and Patterns

- In Haskell, many functional programming patterns are captured through specific type classes.

Type Classes and Patterns

- In Haskell, many functional programming patterns are captured through specific type classes.
- Additionally, the type class mechanism itself and the fact that overloading is prevalent in Haskell give raise to other programming patterns.

Semigroups and Monoids (1)

Semigroups and monoids are algebraic structures:

Semigroups and Monoids (1)

Semigroups and monoids are algebraic structures:

- ***Semigroup***: a set (type) S with an ***associative*** binary operation $\cdot : S \times S \to S$:

$$\forall a, b, c \in S \ : \ (a \cdot b) \cdot c = a \cdot (b \cdot c)$$

Semigroups and Monoids (1)

Semigroups and monoids are algebraic structures:

- ***Semigroup***: a set (type) S with an ***associative*** binary operation $\cdot : S \times S \to S$:

$$\forall a, b, c \in S \; : \; (a \cdot b) \cdot c = a \cdot (b \cdot c)$$

- ***Monoid***: a semigroup with an ***identity element***:

$$\exists e \in S, \forall a \in S \; : \; e \cdot a = a \cdot e = a$$

Semigroups and Monoids (2)

- Semigroups and monoids are patterns that appear frequently in everyday programming.

Semigroups and Monoids (2)

- Semigroups and monoids are patterns that appear frequently in everyday programming.
- Being explicit about when such structures are used
  - makes code clearer
  - offer opportunities for reuse

Semigroups and Monoids (2)

- Semigroups and monoids are patterns that appear frequently in everyday programming.
- Being explicit about when such structures are used
  - makes code clearer
  - offer opportunities for reuse
- The standard Haskell libraries provide type classes to capture these notions.

Class $Semigroup$

Class definition (most important methods):

$$
\begin{aligned}
&\textbf{class } Semigroup\ a\ \textbf{where}\\
&\quad (\diamond) \qquad\ \ :: a \to a \to a\\
&\quad sconcat :: NonEmpty\ a \to a
\end{aligned}
$$

Minimum complete definition: $(\diamond)$ (ASCII: $<>$)
(There is thus a default definition for $sconcat$.)

$NonEmpty$ is the non-empty list type:

$$\textbf{data } NonEmpty\ a = a : |\ [a]$$

Instances of $Semigroup$ (1)

A list $[a]$ is a semigroup (for any type a):

$$\textbf{instance } Semigroup\ [a] \textbf{ where}$$
$$(\diamond) = (+\!\!+)$$

Instances of *Semigroup* (1)

A list $[a]$ is a semigroup (for any type a):

$$
\begin{array}{l}
\textbf{instance } \mathit{Semigroup}\ [a]\ \textbf{where} \\
\quad (\diamond) = (+\!\!+)
\end{array}
$$

$\mathit{Maybe}\ a$ is a semigroup if a is one:

$$
\begin{array}{l}
\textbf{instance } \mathit{Semigroup}\ a \\
\qquad\qquad \Rightarrow \mathit{Semigroup}\ (\mathit{Maybe}\ a)\ \textbf{where} \\
\quad \begin{array}{lll}
\mathit{Nothing} \diamond y & & = y \\
x & \diamond\ \mathit{Nothing} & = x \\
\mathit{Just}\ x & \diamond\ \mathit{Just}\ y & = x \diamond y
\end{array}
\end{array}
$$

Instances of *Semigroup* (2)

Addition and multiplication are associative; a numeric type with either operation forms a semigroup.

Instances of *Semigroup* (2)

Addition and multiplication are associative; a numeric type with either operation forms a semigroup.

But which one to pick? Both are equally useful!

Instances of *Semigroup* (2)

Addition and multiplication are associative; a numeric type with either operation forms a semigroup.

But which one to pick? Both are equally useful!

Idea:

- $Sum\ a$: the semigroup $(a, (+))$
- $Product\ a$: the semigroup $(a, (*))$

Instances of *Semigroup* (3)

Semigroup instances for $Sum\ a$ and $Product\ a$:

$$\mathbf{instance}\ Num\ a \Rightarrow Semigroup\ (Sum\ a)\ \mathbf{where}$$
$$\quad (\diamond) = (+)$$
$$\mathbf{instance}\ Num\ a \Rightarrow Semigroup\ (Product\ a)\ \mathbf{where}$$
$$\quad (\diamond) = (*)$$

Instances of *Semigroup* (4)

Similarly, any type with a total ordering forms a semigroup with maximum or minimum as the associative operation:

- $Max\ a$: the semigroup (a, max)
- $Min\ a$: the semigroup (a, min)

Semigroup instances:

$$\textbf{instance}\ Ord\ a \Rightarrow Semigroup\ (Max\ a)\ \textbf{where}$$
$$(\diamond) = max$$
$$\textbf{instance}\ Ord\ a \Rightarrow Semigroup\ (Min\ a)\ \textbf{where}$$
$$(\diamond) = min$$

Instances of *Semigroup* (5)

All products of semigroups are semigroups; e.g.:

$$
\begin{aligned}
&\textbf{instance}\ (\mathit{Semigroup}\ a, \mathit{Semigroup}\ b)\\
&\qquad\qquad \Rightarrow \mathit{Semigroup}\ (a, b)\ \textbf{where}\\
&\quad (x, y) \diamond (x', y') = (x \diamond x', y \diamond y')
\end{aligned}
$$

Instances of *Semigroup* (5)

All products of semigroups are semigroups; e.g.:

$$\textbf{instance}\ (Semigroup\ a, Semigroup\ b)$$
$$\Rightarrow Semigroup\ (a, b)\ \textbf{where}$$
$$(x, y) \diamond (x', y') = (x \diamond x', y \diamond y')$$

$a \rightarrow b$ is a semigroup if the range b is a semigroup:

$$\textbf{instance}\ Semigroup\ b$$
$$\Rightarrow Semigroup\ (a \rightarrow b)\ \textbf{where}$$
$$f \diamond g = \lambda x \rightarrow f\ x \diamond g\ x$$

Exercise: *Semigroup* Instances

What is the value of the following expressions?

$$[1, 3, 7] \diamond [2, 4]$$

$$Sum\ 3 \diamond Sum\ 1 \diamond Sum\ 5$$

$$Just\ (Max\ 42) \diamond Nothing \diamond Just\ (Max\ 3)$$

$$sconcat\ (Product\ 2 :|\ [Product\ 3, Product\ 4])$$

$$([1], Product\ 2) \diamond ([2, 3], Product\ 3)$$

$$((1:) \diamond tail)\ [4, 5, 6]$$

Class *Monoid*

Recall: A monid is a semigroup with an identity element:

$$
\begin{array}{l}
\textbf{class}\ \mathit{Semigroup}\ a \Rightarrow \mathit{Monoid}\ a\ \textbf{where} \\
\quad \mathit{mempty}\ \ :: a \\
\quad \mathit{mappend} :: a \to a \to a \\
\quad \mathit{mappend} = (\diamond) \\
\quad \mathit{mconcat}\ \ :: [a] \to a \\
\quad \mathit{mconcat}\ \ = \mathit{foldr}\ \mathit{mappend}\ \mathit{mempty}
\end{array}
$$

Minimum complete definition: mempty

Instances of *Monoid* (1)

A list $[a]$ is the archetypical example of a monoid:

$$\begin{array}{l} \mathbf{instance}\ \mathit{Monoid}\ [a]\ \mathbf{where} \\ \quad \mathit{mempty} = [\,] \end{array}$$

Any semigroup can be turned into a monoid by adjoining an identity element:

$$\begin{array}{l} \mathbf{instance}\ \mathit{Semigroup}\ a \\ \qquad\qquad \Rightarrow \mathit{Monoid}\ (\mathit{Maybe}\ a)\ \mathbf{where} \\ \quad \mathit{mempty} = \mathit{Nothing} \end{array}$$

Instances of *Monoid* (2)

Monoid instances for $Sum\ a$ and $Product\ a$:

$$\textbf{instance}\ Num\ a \Rightarrow Monoid\ (Sum\ a)\ \textbf{where}$$
$$\quad mempty = Sum\ 0$$
$$\textbf{instance}\ Num\ a \Rightarrow Monoid\ (Product\ a)\ \textbf{where}$$
$$\quad mempty = Product\ 1$$

Instances of *Monoid* (3)

Monoid instances for $Min\ a$ and $Max\ a$:

$$\begin{aligned}
&\textbf{instance}\ (Ord\ a, Bounded\ a) \Rightarrow\\
&\qquad\qquad Monoid\ (Min\ a)\ \textbf{where}\\
&\quad mempty = maxBound
\end{aligned}$$

$$\begin{aligned}
&\textbf{instance}\ (Ord\ a, Bounded\ a) \Rightarrow\\
&\qquad\qquad Monoid\ (Max\ a)\ \textbf{where}\\
&\quad mempty = minBound
\end{aligned}$$

Instances of *Monoid* (4)

All products of monoids are monoids; e.g.:

$$\begin{aligned}&\textbf{instance}\ (Monoid\ a, Monoid\ b)\\&\qquad\qquad \Rightarrow Monoid\ (a, b)\ \textbf{where}\\&\quad mempty = (mempty, mempty)\end{aligned}$$

Instances of *Monoid* (4)

All products of monoids are monoids; e.g.:

$$\begin{aligned}&\textbf{instance}\ (\mathit{Monoid}\ a, \mathit{Monoid}\ b)\\ &\qquad\qquad \Rightarrow \mathit{Monoid}\ (a, b)\ \textbf{where}\\ &\quad \mathit{mempty} = (\mathit{mempty}, \mathit{mempty})\end{aligned}$$

$a \to b$ is a monoid if the range b is a monoid:

$$\begin{aligned}&\textbf{instance}\ \mathit{Monoid}\ b \Rightarrow \mathit{Monoid}\ (a \to b)\ \textbf{where}\\ &\quad \mathit{mempty}\ \_ = \mathit{mempty}\end{aligned}$$

Functors (1)

A Functor is a notion that originated in a branch of mathematics called Category Theory.

However, for our purposes, we can think of functors as type constructors T (of arity 1) for which a function map can be defined:

$$map :: (a \to b) \to T\,a \to T\,b$$

that satisfies the following laws:

$$\begin{aligned} map\ id &= id \\ map(f \circ g) &= map\ f \circ map\ g \end{aligned}$$

Functors (2)

Common examples of functors include (but are not limited to) ***container types*** like lists:

$$
\begin{aligned}
&mapList :: (a \to b) \to [\,a\,] \to [\,b\,] \\
&mapList\ \_\ [\,] \qquad\qquad = [\,] \\
&mapList\ f\ (x : xs) = f\ x : mapList\ f\ xs
\end{aligned}
$$

Functors (3)

And trees; e.g.:

$$
\begin{aligned}
&\mathbf{data}\ \mathit{Tree}\ a = \mathit{Leaf}\ \ a \\
&\qquad\qquad\qquad\ \mid\ \mathit{Node}\ (\mathit{Tree}\ \ a)\ a\ (\mathit{Tree}\ \ a)
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{mapTree} :: (a \to b) \to \mathit{Tree}\ \ a \to \mathit{Tree}\ \ b \\
&\mathit{mapTree}\ f\ (\mathit{Leaf}\ \ x) \qquad\ = \mathit{Leaf}\ \ (f\ \ x) \\
&\mathit{mapTree}\ f\ (\mathit{Node}\ l\ x\ r) = \mathit{Node}\ (\mathit{mapTree}\ f\ \ l) \\
&\qquad\qquad\qquad\qquad\qquad\qquad\quad (f\ \ x) \\
&\qquad\qquad\qquad\qquad\qquad\qquad\quad (\mathit{mapTree}\ f\ \ r)
\end{aligned}
$$

Class *Functor* (1)

Of course, the notion of a functor is captured by a type class in Haskell:

$$
\begin{aligned}
&\textbf{class } \mathit{Functor}\ f \textbf{ where}\\
&\quad \mathit{fmap} :: (a \to b) \to f\ a \to f\ b\\
&\quad (<\$) :: a \to f\ b \to f\ a\\
&\quad (<\$) = \mathit{fmap} \circ \mathit{const}
\end{aligned}
$$

Class *Functor* (2)

However, Haskell's type system is not powerful enough to enforce the functor laws.

Class *Functor* (2)

However, Haskell's type system is not powerful enough to enforce the functor laws.

In general, the programmer is responsible for ensuring that an instance respects all laws associated with a type class.

Class *Functor* (2)

However, Haskell's type system is not powerful enough to enforce the functor laws.

In general, the programmer is responsible for ensuring that an instance respects all laws associated with a type class.

Note that the type of $fmap$ can be read:

$$(a \to b) \to (f\ a \to f\ b)$$

That is, we can see $fmap$ as promoting a function to work in a different context.

Instances of *Functor* (1)

As noted, list is a functor:

$$\textbf{instance } \mathit{Functor}\ [\,]\ \textbf{where}$$
$$\quad \mathit{fmap} = \mathit{listMap}$$

Instances of *Functor* (1)

As noted, list is a functor:

$$\textbf{instance}\ \mathit{Functor}\ [\,]\ \textbf{where}$$
$$\quad \mathit{fmap} = \mathit{listMap}$$

Maybe is also a functor:

$$\textbf{instance}\ \mathit{Functor}\ \mathit{Maybe}\ \textbf{where}$$
$$\quad \mathit{fmap}\ \_\ \mathit{Nothing} = \mathit{Nothing}$$
$$\quad \mathit{fmap}\ f\ (\mathit{Just}\ x) = \mathit{Just}\ (f\ x)$$

Instances of *Functor* (2)

The type of functions from a given domain is a functor with function composition as the map:

$$\textbf{instance}\ \mathit{Functor}\ ((\rightarrow)\ a)\ \textbf{where}$$
$$\quad \mathit{fmap} = (\circ)$$

Instances of *Functor* (2)

The type of functions from a given domain is a functor with function composition as the map:

$$\begin{array}{l} \textbf{instance}\ \mathit{Functor}\ ((\rightarrow)\ a)\ \textbf{where} \\ \quad \mathit{fmap} = (\circ) \end{array}$$

Indeed, there is a GHC extension for deriving *Functor* instances. For example, the functor instance for our tree type can be derived:

$$\begin{array}{lcl} \textbf{data}\ \mathit{Tree}\ a & = & \mathit{Leaf}\ a \\ & \mid & \mathit{Node}\ (\mathit{Tree}\ a)\ a\ (\mathit{Tree}\ a) \\ & \multicolumn{2}{l}{\textbf{deriving}\ \mathit{Functor}} \end{array}$$

Class *Foldable* (1)

Class of data structures that can be folded to a summary value.

Many methods; minimal instance $\mathit{foldMap}$, foldr:

$$
\begin{array}{l}
\textbf{class}\ \mathit{Foldable}\ t\ \textbf{where} \\
\quad \begin{array}{lll}
\mathit{fold} & :: & \mathit{Monoid}\ m \Rightarrow t\ m \to m \\
\mathit{foldMap} & :: & \mathit{Monoid}\ m \Rightarrow (a \to m) \to t\ a \to m \\
\mathit{foldr} & :: & (a \to b \to b) \to b \to t\ a \to b \\
\mathit{foldr}' & :: & (a \to b \to b) \to b \to t\ a \to b \\
\mathit{foldl} & :: & (b \to a \to b) \to b \to t\ a \to b \\
\mathit{foldl}' & :: & (b \to a \to b) \to b \to t\ a \to b
\end{array}
\end{array}
$$

Class *Foldable* (2)

(continued)

$$
\begin{aligned}
foldr1 &:: (a \to a \to a) \to t\ a \to a \\
foldl1 &:: (a \to a \to a) \to t\ a \to a \\
toList &:: t\ a \to [a] \\
null &:: t\ a \to Bool \\
length &:: t\ a \to Int \\
elem &:: Eq\ a \Rightarrow a \to t\ a \to Bool
\end{aligned}
$$

(Note that $length$ should be understood as $size$.)

Class *Foldable* (3)

(continued)

$$
\begin{aligned}
maximum &:: Ord\ a \Rightarrow t\ a \to a\\
minimum &:: Ord\ a \Rightarrow t\ a \to a\\
sum &:: Num\ a \Rightarrow t\ a \to a\\
product &:: Num\ a \Rightarrow t\ a \to a
\end{aligned}
$$

Class *Foldable* (3)

(continued)

$$
\begin{aligned}
\mathit{maximum} &:: \mathit{Ord}\ a \Rightarrow t\ a \to a \\
\mathit{minimum} &:: \mathit{Ord}\ a \Rightarrow t\ a \to a \\
\mathit{sum} &:: \mathit{Num}\ a \Rightarrow t\ a \to a \\
\mathit{product} &:: \mathit{Num}\ a \Rightarrow t\ a \to a
\end{aligned}
$$

Note: foldl typically incurs a large space overhead due to laziness. The version with strict applictaion of the operator,foldl' is typically preferable.

Instances of *Foldable* (1)

All expected instances, e.g.:

- **instance** $Foldable\ [\,]$ **where** $\ldots$
- **instance** $Foldable\ Maybe$ **where** $\ldots$

Instances of *Foldable* (1)

All expected instances, e.g.:

- $\textbf{instance } \mathit{Foldable}\ [\,]\ \textbf{where} \ldots$
- $\textbf{instance } \mathit{Foldable}\ \mathit{Maybe}\ \textbf{where} \ldots$

And GHC extension allows deriving instances in many cases; e.g.

$$\textbf{data } \mathit{Tree}\ a = \ldots \textbf{deriving } \mathit{Foldable}$$

Instances of *Foldable* (2)

But there are also some instances that are less expected, e.g.:

- $\textbf{instance}\ \mathit{Foldable}\ (\mathit{Either}\ a)\ \textbf{where} \ldots$
- $\textbf{instance}\ \mathit{Foldable}\ ((,)\ a)\ \textbf{where} \ldots$

Instances of *Foldable* (2)

But there are also some instances that are less expected, e.g.:

- $\textbf{instance}\ \mathit{Foldable}\ (\mathit{Either}\ a)\ \textbf{where} \ldots$
- $\textbf{instance}\ \mathit{Foldable}\ ((,)\ a)\ \textbf{where} \ldots$

This has some arguably odd consequences:

$$\begin{aligned}
&\mathit{length}\ (1,2) &&\Rightarrow 1\\
&\mathit{sum}\ (1,2) &&\Rightarrow 2\\
&\mathit{length}\ (\mathit{Left}\ 1) &&\Rightarrow 0\\
&\mathit{length}\ (\mathit{Right}\ 2) &&\Rightarrow 1
\end{aligned}$$

Example: Folding Over a Tree (1)

Consider:

$$\begin{array}{l}\mathbf{data}\ \mathit{Tree}\ a = \mathit{Empty} \\ \quad\ \ \mid \mathit{Node}\ (\mathit{Tree}\ a)\ a\ (\mathit{Tree}\ a) \\ \quad\ \ \mathbf{deriving}\ (\mathit{Show}, \mathit{Eq})\end{array}$$

Example: Folding Over a Tree (1)

Consider:

$$\begin{aligned}&\mathbf{data}\ \mathit{Tree}\ a = \mathit{Empty}\\&\quad\mid \mathit{Node}\ (\mathit{Tree}\ a)\ a\ (\mathit{Tree}\ a)\\&\quad\mathbf{deriving}\ (\mathit{Show}, \mathit{Eq})\end{aligned}$$

Let us make it an instance of $\mathit{Foldable}$:

$$\begin{aligned}&\mathbf{instance}\ \mathit{Foldable}\ \mathit{Tree}\ \mathbf{where}\\&\quad \mathit{foldMap}\ f\ \mathit{Empty} = \mathit{mempty}\\&\quad \mathit{foldMap}\ f\ (\mathit{Node}\ l\ a\ r) =\\&\qquad \mathit{foldMap}\ f\ l \diamond f\ a \diamond \mathit{foldMap}\ f\ r\end{aligned}$$

Example: Folding Over a Tree (2)

We wish to compute the sum and max over a tree of Int. One way:

$$sumMax :: Tree\ Int \to (Int, Int)$$
$$sumMax\ t = (foldl\ (+)\ 0\ t, foldl\ max\ minBound\ t)$$

Example: Folding Over a Tree (2)

We wish to compute the sum and max over a tree of Int. One way:

$$sumMax :: Tree\ Int \to (Int, Int)$$
$$sumMax\ t = (foldl\ (+)\ 0\ t, foldl\ max\ minBound\ t)$$

Another way, with a single traversal:

$$sumMax :: Tree\ Int \to (Int, Int)$$
$$sumMax\ t = (sm, mx)$$
$$\quad \textbf{where}$$
$$\qquad (Sum\ sm, Max\ mx) =$$
$$\qquad\quad foldMap\ (\lambda n \to (Sum\ n, Max\ n))\ t$$

Example: Folding Over a Tree (3)

The latter can be generalized to e.g. computing the sum, product, min, and max in a single traversal:

$$\begin{array}{l} \mathit{foldMap} \\ \quad (\lambda n \rightarrow (\mathit{Sum}\ n, \mathit{Product}\ n, \mathit{Min}\ n, \mathit{Max}\ n)) \\ \quad t \end{array}$$

Aside: Foldable?

Note that the kind of “folding” captured by the class *Foldable* in general makes it impossible to recover the structure over which the “folding” takes place.

Aside: Foldable?

Note that the kind of “folding” captured by the class *Foldable* in general makes it impossible to recover the structure over which the “folding” takes place.

Such an operation is also known as “reduce” or “crush”, and some authors prefer to reserve the term “fold” for ***catamorphisms***, where a separate combining function is given for each constructor, making it possible to recover the structure.

Aside: Foldable?

Note that the kind of “folding” captured by the class $Foldable$ in general makes it impossible to recover the structure over which the “folding” takes place.

Such an operation is also known as “reduce” or “crush”, and some authors prefer to reserve the term “fold” for ***catamorphisms***, where a separate combining function is given for each constructor, making it possible to recover the structure.

One might thus argue that $Reducible$ or $Crushable$ would have been a more precise name.

MapReduce

Functional mapping and folding (reducing) inspired the MapReduce programming model; e.g.

- Google's original MapReduce framework
- Apache Hadoop

MapReduce

Functional mapping and folding (reducing) inspired the MapReduce programming model; e.g.

- Google's original MapReduce framework
- Apache Hadoop

Functional mapping and folding with ***associative*** operator (semigroup) is amenable to parallelization and distribution.

MapReduce

Functional mapping and folding (reducing) inspired the MapReduce programming model; e.g.

- Google's original MapReduce framework
- Apache Hadoop

Functional mapping and folding with ***associative*** operator (semigroup) is amenable to parallelization and distribution.

However, achieving scalability in practice required both careful engineering of the frameworks as such, and a good understanding of how to use them on part of the user.

COMP4075/G54RFP: Lecture 7

Introduction to Monads

Henrik Nilsson

University of Nottingham, UK

A Blessing and a Curse

A Blessing and a Curse

- The ***BIG*** advantage of ***pure*** functional programming is

A Blessing and a Curse

- The ***BIG*** advantage of ***pure*** functional programming is

  "everything is explicit;"

  i.e., flow of data manifest, no side effects.

A Blessing and a Curse

- The ***BIG*** advantage of ***pure*** functional programming is

  “everything is explicit;”

  i.e., flow of data manifest, no side effects. Makes it a lot easier to understand large programs.

A Blessing and a Curse

- The ***BIG*** advantage of ***pure*** functional programming is

  “everything is explicit;”

  i.e., flow of data manifest, no side effects. Makes it a lot easier to understand large programs.
- The ***BIG*** problem with ***pure*** functional programming is

A Blessing and a Curse

- The ***BIG*** advantage of ***pure*** functional programming is

  "everything is explicit;"

  i.e., flow of data manifest, no side effects. Makes it a lot easier to understand large programs.

- The ***BIG*** problem with ***pure*** functional programming is

  "everything is explicit."

A Blessing and a Curse

- The ***BIG*** advantage of ***pure*** functional programming is

  "everything is explicit;"

  i.e., flow of data manifest, no side effects. Makes it a lot easier to understand large programs.

- The ***BIG*** problem with ***pure*** functional programming is

  "everything is explicit."

  Can add a lot of clutter, make it hard to maintain code

Conundrum

“Shall I be pure or impure?” (Wadler, 1992)

Conundrum

"Shall I be pure or impure?" (Wadler, 1992)

- Absence of effects
  - facilitates understanding and reasoning
  - makes lazy evaluation viable
  - allows choice of reduction order, e.g. parallel
  - enhances modularity and reuse.

Conundrum

"Shall I be pure or impure?" (Wadler, 1992)

- Absence of effects
  - facilitates understanding and reasoning
  - makes lazy evaluation viable
  - allows choice of reduction order, e.g. parallel
  - enhances modularity and reuse.
- Effects (state, exceptions, . . .) can
  - help making code concise
  - facilitate maintenance
  - improve the efficiency.

Answer to Conundrum: Monads (1)

- Monads bridges the gap: allow effectful programming in a pure setting.

Answer to Conundrum: Monads (1)

- Monads bridges the gap: allow effectful programming in a pure setting.
- Key idea: ***Computational types***: an object of type $M\,A$ denotes a ***computation*** of an object of type A.

Answer to Conundrum: Monads (1)

- Monads bridges the gap: allow effectful programming in a pure setting.
- Key idea: ***Computational types***: an object of type $M\,A$ denotes a ***computation*** of an object of type A.
- ***Thus we shall be both pure and impure, whatever takes our fancy!***

Answer to Conundrum: Monads (1)

- Monads bridges the gap: allow effectful programming in a pure setting.
- Key idea: ***Computational types***: an object of type $M\,A$ denotes a ***computation*** of an object of type A.
- ***Thus we shall be both pure and impure, whatever takes our fancy!***
- Monads originated in Category Theory.

Answer to Conundrum: Monads (1)

- Monads bridges the gap: allow effectful programming in a pure setting.
- Key idea: ***Computational types***: an object of type $M\,A$ denotes a ***computation*** of an object of type A.
- ***Thus we shall be both pure and impure, whatever takes our fancy!***
- Monads originated in Category Theory.
- Adapted by
  - Moggi for structuring denotational semantics
  - Wadler for structuring functional programs

Answer to Conundrum: Monads (2)

Monads

- promote ***disciplined*** use of effects since the type reflects which effects can occur;

Answer to Conundrum: Monads (2)

Monads

- promote ***disciplined*** use of effects since the type reflects which effects can occur;
- allow great ***flexibility*** in tailoring the effect structure to precise needs;

Answer to Conundrum: Monads (2)

Monads

- promote ***disciplined*** use of effects since the type reflects which effects can occur;
- allow great ***flexibility*** in tailoring the effect structure to precise needs;
- support ***changes*** to the effect structure with minimal impact on the overall program structure;

Answer to Conundrum: Monads (2)

Monads

- promote ***disciplined*** use of effects since the type reflects which effects can occur;
- allow great ***flexibility*** in tailoring the effect structure to precise needs;
- support ***changes*** to the effect structure with minimal impact on the overall program structure;
- allow integration into a pure setting of ***real*** effects such as
  - I/O
  - mutable state.

This Lecture

Pragmatic introduction to monads:

- Effectful computations
- Identifying a common pattern
- Monads as a ***design pattern***

Example 1: A Simple Evaluator

$$
\begin{aligned}
\textbf{data}\ Exp &= Lit\ Integer \\
&\mid Add\ Exp\ Exp \\
&\mid Sub\ Exp\ Exp \\
&\mid Mul\ Exp\ Exp \\
&\mid Div\ Exp\ Exp
\end{aligned}
$$

$$
\begin{aligned}
&eval :: Exp \to Integer \\
&eval\ (Lit\ n) &&= n \\
&eval\ (Add\ e1\ e2) &&= eval\ e1 + eval\ e2 \\
&eval\ (Sub\ e1\ e2) &&= eval\ e1 - eval\ e2 \\
&eval\ (Mul\ e1\ e2) &&= eval\ e1 * eval\ e2 \\
&eval\ (Div\ e1\ e2) &&= eval\ e1\ \text{`}div\text{`}\ eval\ e2
\end{aligned}
$$

Making the Evaluator Safe (1)

$$\textbf{data}\ \mathit{Maybe}\ a = \mathit{Nothing} \mid \mathit{Just}\ a$$

$$\begin{array}{l}
\mathit{safeEval} :: \mathit{Exp} \to \mathit{Maybe}\ \mathit{Integer} \\
\mathit{safeEval}\ (\mathit{Lit}\ n) \qquad\quad = \mathit{Just}\ n \\
\mathit{safeEval}\ (\mathit{Add}\ e1\ \ e2) = \\
\quad \textbf{case}\ \mathit{safeEval}\ e1\ \textbf{of} \\
\qquad \mathit{Nothing} \to \mathit{Nothing} \\
\qquad \mathit{Just}\ n1\ \to \textbf{case}\ \mathit{safeEval}\ e2\ \textbf{of} \\
\qquad\qquad\qquad\qquad \mathit{Nothing} \to \mathit{Nothing} \\
\qquad\qquad\qquad\qquad \mathit{Just}\ n2\ \to \mathit{Just}\ (n1 + n2)
\end{array}$$

Making the Evaluator Safe (2)

$$
\begin{array}{l}
\mathit{safeEval}\ (\mathit{Sub}\ e1\ \ e2) = \\
\quad \textbf{case}\ \mathit{safeEval}\ e1\ \textbf{of} \\
\qquad \mathit{Nothing} \rightarrow \mathit{Nothing} \\
\qquad \mathit{Just}\ n1\ \ \rightarrow \textbf{case}\ \mathit{safeEval}\ e2\ \textbf{of} \\
\qquad\qquad\qquad\qquad \mathit{Nothing} \rightarrow \mathit{Nothing} \\
\qquad\qquad\qquad\qquad \mathit{Just}\ n2\ \ \rightarrow \mathit{Just}\ (n1 - n2)
\end{array}
$$

Making the Evaluator Safe (3)

$$
\begin{array}{l}
\mathit{safeEval}\ (\mathit{Mul}\ \mathit{e1}\ \mathit{e2}) = \\
\quad \textbf{case}\ \mathit{safeEval}\ \mathit{e1}\ \textbf{of} \\
\quad\quad \mathit{Nothing} \rightarrow \mathit{Nothing} \\
\quad\quad \mathit{Just}\ \mathit{n1}\ \rightarrow \textbf{case}\ \mathit{safeEval}\ \mathit{e2}\ \textbf{of} \\
\quad\quad\quad\quad\quad\quad \mathit{Nothing} \rightarrow \mathit{Nothing} \\
\quad\quad\quad\quad\quad\quad \mathit{Just}\ \mathit{n2}\ \rightarrow \mathit{Just}\ (\mathit{n1} * \mathit{n2})
\end{array}
$$

Making the Evaluator Safe (4)

$$
\begin{array}{l}
\mathit{safeEval}\ (\mathit{Div}\ \mathit{e1}\ \ \mathit{e2}) = \\
\quad \textbf{case}\ \mathit{safeEval}\ \mathit{e1}\ \textbf{of} \\
\qquad \mathit{Nothing} \rightarrow \mathit{Nothing} \\
\qquad \mathit{Just}\ \mathit{n1}\ \ \rightarrow \textbf{case}\ \mathit{safeEval}\ \mathit{e2}\ \textbf{of} \\
\qquad\qquad\qquad\qquad \mathit{Nothing} \rightarrow \mathit{Nothing} \\
\qquad\qquad\qquad\qquad \mathit{Just}\ \mathit{n2}\ \ \rightarrow \\
\qquad\qquad\qquad\qquad\quad \textbf{if}\ \mathit{n2} \equiv 0 \\
\qquad\qquad\qquad\qquad\quad \textbf{then}\ \mathit{Nothing} \\
\qquad\qquad\qquad\qquad\quad \textbf{else}\ \mathit{Just}\ (\mathit{n1}\ \text{`}\mathit{div}\text{`}\ \mathit{n2})
\end{array}
$$

Any Common Pattern?

Clearly a lot of code duplication!
Can we factor out a common pattern?

Any Common Pattern?

Clearly a lot of code duplication!
Can we factor out a common pattern?

We note:

- ***Sequencing*** of evaluations (or ***computations***).

Any Common Pattern?

Clearly a lot of code duplication!
Can we factor out a common pattern?

We note:

- ***Sequencing*** of evaluations (or ***computations***).
- If one evaluation fails, fail overall.

Any Common Pattern?

Clearly a lot of code duplication!
Can we factor out a common pattern?

We note:

- ***Sequencing*** of evaluations (or ***computations***).
- If one evaluation fails, fail overall.
- Otherwise, make result available to following evaluations.

Sequencing Evaluations

$$
\begin{aligned}
&\mathit{evalSeq} :: \mathit{Maybe\ Integer} \\
&\qquad\qquad \rightarrow (\mathit{Integer} \rightarrow \mathit{Maybe\ Integer}) \\
&\qquad\qquad \rightarrow \mathit{Maybe\ Integer} \\
&\mathit{evalSeq}\ \mathit{ma}\ f = \mathbf{case}\ \mathit{ma}\ \mathbf{of} \\
&\qquad\qquad\qquad\qquad \mathit{Nothing} \rightarrow \mathit{Nothing} \\
&\qquad\qquad\qquad\qquad \mathit{Just}\ a \rightarrow f\ a
\end{aligned}
$$

Exercise 1: Refactoring *safeEval*

Rewrite *safeEval*, case *Add*, using *evalSeq*:

```
safeEval (Add e1 e2) =
    case safeEval e1 of
        Nothing -> Nothing
        Just n1 ->
            case safeEval e2 of
                Nothing -> Nothing
                Just n2 -> Just (n1 + n2)

evalSeq ma f =
    case ma of
        Nothing -> Nothing
        Just a  -> f a
```

Exercise 1: Solution

$$
\begin{array}{l}
\mathit{safeEval} :: \mathit{Exp} \to \mathit{Maybe\ Integer} \\
\mathit{safeEval}\ (\mathit{Add}\ e1\ \ e2) = \\
\quad \mathit{evalSeq}\ (\mathit{safeEval}\ \ e1) \\
\qquad\qquad\quad (\lambda n1\ \to\ \mathit{evalSeq}\ (\mathit{safeEval}\ \ e2) \\
\qquad\qquad\qquad\qquad\qquad\qquad (\lambda n2\ \to\ \mathit{Just}\ (n1 + n2)))
\end{array}
$$

or

$$
\begin{array}{l}
\mathit{safeEval} :: \mathit{Exp} \to \mathit{Maybe\ Integer} \\
\mathit{safeEval}\ (\mathit{Add}\ e1\ \ e2) = \\
\quad \mathit{safeEval}\ \ e1\ \text{`}\mathit{evalSeq}\text{`}\ \lambda n1\ \to \\
\quad \mathit{safeEval}\ \ e2\ \text{`}\mathit{evalSeq}\text{`}\ \lambda n2\ \to \\
\quad \mathit{Just}\ (n1 + n2)
\end{array}
$$

Refactored Safe Evaluator (1)

$$
\begin{aligned}
&safeEval :: Exp \rightarrow Maybe\ Integer \\
&safeEval\ (Lit\ n) = Just\ n \\
&safeEval\ (Add\ e1\ e2) = \\
&\quad safeEval\ e1\ `evalSeq`\ \lambda n1 \rightarrow \\
&\quad safeEval\ e2\ `evalSeq`\ \lambda n2 \rightarrow \\
&\quad Just\ (n1 + n2) \\
&safeEval\ (Sub\ e1\ e2) = \\
&\quad safeEval\ e1\ `evalSeq`\ \lambda n1 \rightarrow \\
&\quad safeEval\ e2\ `evalSeq`\ \lambda n2 \rightarrow \\
&\quad Just\ (n1 - n2)
\end{aligned}
$$

Refactored Safe Evaluator (2)

$$
\begin{array}{l}
\mathit{safeEval}\ (\mathit{Mul}\ \mathit{e1}\ \ \mathit{e2}) = \\
\quad \mathit{safeEval}\ \mathit{e1}\ `\mathit{evalSeq}`\ \lambda \mathit{n1} \rightarrow \\
\quad \mathit{safeEval}\ \mathit{e2}\ `\mathit{evalSeq}`\ \lambda \mathit{n2} \rightarrow \\
\quad \mathit{Just}\ (\mathit{n1} * \mathit{n2}) \\
\mathit{safeEval}\ (\mathit{Div}\ \mathit{e1}\ \ \mathit{e2}) = \\
\quad \mathit{safeEval}\ \mathit{e1}\ `\mathit{evalSeq}`\ \lambda \mathit{n1} \rightarrow \\
\quad \mathit{safeEval}\ \mathit{e2}\ `\mathit{evalSeq}`\ \lambda \mathit{n2} \rightarrow \\
\quad \mathbf{if}\ \mathit{n2} \equiv 0 \\
\quad \mathbf{then}\ \mathit{Nothing} \\
\quad \mathbf{else}\ \mathit{Just}\ (\mathit{n1}\ `\mathit{div}`\ \mathit{n2})
\end{array}
$$

`Maybe` Viewed as a Computation (1)

- Consider a value of type `Maybe a` as denoting a ***computation*** of a value of type `a` that ***may fail***.

`Maybe` Viewed as a Computation (1)

- Consider a value of type `Maybe a` as denoting a ***computation*** of a value of type `a` that ***may fail***.
- When sequencing possibly failing computations, a natural choice is to fail overall once a subcomputation fails.

`Maybe` Viewed as a Computation (1)

- Consider a value of type `Maybe a` as denoting a ***computation*** of a value of type `a` that ***may fail***.
- When sequencing possibly failing computations, a natural choice is to fail overall once a subcomputation fails.
- I.e. ***failure is an effect***, implicitly affecting subsequent computations.

`Maybe` Viewed as a Computation (1)

- Consider a value of type `Maybe a` as denoting a ***computation*** of a value of type `a` that ***may fail***.
- When sequencing possibly failing computations, a natural choice is to fail overall once a subcomputation fails.
- I.e. ***failure is an effect***, implicitly affecting subsequent computations.
- Let's generalize and adopt names reflecting our intentions.

`Maybe` Viewed as a Computation (2)

Successful computation of a value:

$$mbReturn :: a \rightarrow Maybe\ a$$
$$mbReturn = Just$$

Sequencing of possibly failing computations:

$$mbSeq :: Maybe\ a \rightarrow (a \rightarrow Maybe\ b) \rightarrow Maybe\ b$$
$$\begin{aligned} mbSeq\ ma\ f = \mathbf{case}\ ma\ \mathbf{of} \\ Nothing &\rightarrow Nothing \\ Just\ a &\rightarrow f\ a \end{aligned}$$

`Maybe` Viewed as a Computation (3)

Failing computation:

$$
\begin{aligned}
&mbFail :: Maybe\ a\\
&mbFail = Nothing
\end{aligned}
$$

The Safe Evaluator Revisited

$$
\begin{array}{l}
\mathit{safeEval} :: \mathit{Exp} \rightarrow \mathit{Maybe\ Integer} \\
\mathit{safeEval}\ (\mathit{Lit}\ n) = \mathit{mbReturn}\ n \\
\mathit{safeEval}\ (\mathit{Add}\ e1\ e2) = \\
\quad \mathit{safeEval}\ e1\ \text{`}\mathit{mbSeq}\text{`}\ \lambda n1 \rightarrow \\
\quad \mathit{safeEval}\ e2\ \text{`}\mathit{mbSeq}\text{`}\ \lambda n2 \rightarrow \\
\quad \mathit{mbReturn}\ (n1 + n2) \\
\ldots \\
\mathit{safeEval}\ (\mathit{Div}\ e1\ e2) = \\
\quad \mathit{safeEval}\ e1\ \text{`}\mathit{mbSeq}\text{`}\ \lambda n1 \rightarrow \\
\quad \mathit{safeEval}\ e2\ \text{`}\mathit{mbSeq}\text{`}\ \lambda n2 \rightarrow \\
\quad \mathbf{if}\ n2 \equiv 0\ \mathbf{then}\ \mathit{mbFail}\ \mathbf{else}\ \mathit{mbReturn}\ (n1\ \text{`}\mathit{div}\text{`}\ n
\end{array}
$$

Example 2: Numbering Trees

$$\textbf{data } \mathit{Tree}\ a = \mathit{Leaf}\ a \mid \mathit{Node}\ (\mathit{Tree}\ a)\ (\mathit{Tree}\ a)$$

$$\mathit{numberTree} :: \mathit{Tree}\ a \to \mathit{Tree}\ \mathit{Int}$$

$$\mathit{numberTree}\ t = \mathit{fst}\ (\mathit{ntAux}\ t\ 0)$$

$$\quad \textbf{where}$$

$$\qquad \mathit{ntAux} :: \mathit{Tree}\ a \to \mathit{Int} \to (\mathit{Tree}\ \mathit{Int}, \mathit{Int})$$

$$\qquad \mathit{ntAux}\ (\mathit{Leaf}\ \_)\ n \qquad = (\mathit{Leaf}\ n, n + 1)$$

$$\qquad \mathit{ntAux}\ (\mathit{Node}\ \mathit{t1}\ \mathit{t2})\ n =$$

$$\qquad\quad \textbf{let } (\mathit{t1}', n') = \mathit{ntAux}\ \mathit{t1}\ n$$

$$\qquad\quad \textbf{in let } (\mathit{t2}', n'') = \mathit{ntAux}\ \mathit{t2}\ n'$$

$$\qquad\qquad \textbf{in } (\mathit{Node}\ \mathit{t1}'\ \mathit{t2}', n'')$$

Observations

- Repetitive pattern: threading a counter through a ***sequence*** of tree numbering ***computations***.

Observations

- Repetitive pattern: threading a counter through a ***sequence*** of tree numbering ***computations***.
- It is very easy to pass on the wrong version of the counter!

Observations

- Repetitive pattern: threading a counter through a ***sequence*** of tree numbering ***computations***.
- It is very easy to pass on the wrong version of the counter!

Can we do better?

Stateful Computations (1)

- A ***stateful computation*** consumes a state and returns a result along with a possibly updated state.

Stateful Computations (1)

- A ***stateful computation*** consumes a state and returns a result along with a possibly updated state.
- The following type synonym captures this idea:

$$\mathbf{type}\ S\ a = Int \to (a, Int)$$

(Only Int state for the sake of simplicity.)

Stateful Computations (1)

- A ***stateful computation*** consumes a state and returns a result along with a possibly updated state.
- The following type synonym captures this idea:

$$\mathbf{type}\ S\ a = Int \to (a, Int)$$

(Only Int state for the sake of simplicity.)

- A value (function) of type $S\ a$ can now be viewed as denoting a stateful computation computing a value of type a.

Stateful Computations (2)

- When sequencing stateful computations, the resulting state should be passed on to the next computation.

Stateful Computations (2)

- When sequencing stateful computations, the resulting state should be passed on to the next computation.
- I.e. ***state updating is an effect***, implicitly affecting subsequent computations.
  (As we would expect.)

Stateful Computations (3)

Computation of a value without changing the state (For ref.: $S\ a \quad = \quad Int \rightarrow (a, Int)$):

$$sReturn :: a \rightarrow S\ a$$
$$sReturn\ a = ???$$

Stateful Computations (3)

Computation of a value without changing the state (For ref.: $S\ a \quad = \quad Int \rightarrow (a, Int)$):

$$sReturn :: a \rightarrow S\ a$$
$$sReturn\ a = \lambda n \rightarrow (a, n)$$

Stateful Computations (3)

Computation of a value without changing the state (For ref.: $S\ a \quad = \quad Int \to (a, Int)$):

$$sReturn :: a \to S\ a$$
$$sReturn\ a = \lambda n \to (a, n)$$

Sequencing of stateful computations:

$$sSeq :: S\ a \to (a \to S\ b) \to S\ b$$
$$sSeq\ sa\ f = ???$$

Stateful Computations (3)

Computation of a value without changing the state (For ref.: $S\ a \quad = \quad Int \rightarrow (a, Int)$):

$$sReturn :: a \rightarrow S\ a$$
$$sReturn\ a = \lambda n \rightarrow (a, n)$$

Sequencing of stateful computations:

$$sSeq :: S\ a \rightarrow (a \rightarrow S\ b) \rightarrow S\ b$$
$$\begin{aligned} &sSeq\ sa\ f = \lambda n \rightarrow \\ &\quad \mathbf{let}\ (a, n') = sa\ n \\ &\quad \mathbf{in}\ f\ a\ n' \end{aligned}$$

Stateful Computations (4)

Reading and incrementing the state
(For ref.: $S\ a \quad = \quad Int \rightarrow (a, Int)$):

$$sInc :: S\ Int$$
$$sInc = \lambda n \rightarrow (n, n + 1)$$

Numbering trees revisited

$$\textbf{data}\ \mathit{Tree}\ a = \mathit{Leaf}\ a \mid \mathit{Node}\ (\mathit{Tree}\ a)\ (\mathit{Tree}\ a)$$

$$\mathit{numberTree} :: \mathit{Tree}\ a \to \mathit{Tree}\ \mathit{Int}$$

$$\mathit{numberTree}\ t = \mathit{fst}\ (\mathit{ntAux}\ t\ 0)$$

$$\quad \textbf{where}$$

$$\qquad \mathit{ntAux} :: \mathit{Tree}\ a \to S\ (\mathit{Tree}\ \mathit{Int})$$

$$\qquad \mathit{ntAux}\ (\mathit{Leaf}\ \_) =$$

$$\qquad\quad \mathit{sInc}\ \text{`}\mathit{sSeq}\text{`}\ \lambda n \to \mathit{sReturn}\ (\mathit{Leaf}\ n)$$

$$\qquad \mathit{ntAux}\ (\mathit{Node}\ t1\ t2) =$$

$$\qquad\quad \mathit{ntAux}\ t1\ \text{`}\mathit{sSeq}\text{`}\ \lambda t1' \to$$

$$\qquad\quad \mathit{ntAux}\ t2\ \text{`}\mathit{sSeq}\text{`}\ \lambda t2' \to$$

$$\qquad\quad \mathit{sReturn}\ (\mathit{Node}\ t1'\ t2')$$

Observations

- The “plumbing” has been captured by the abstractions.

Observations

- The “plumbing” has been captured by the abstractions.
- In particular:
  - counter no longer manipulated directly
  - no longer any risk of “passing on” the wrong version of the counter!

Comparison of the examples

- Both examples characterized by sequencing of effectful computations.

Comparison of the examples

- Both examples characterized by sequencing of effectful computations.
- Both examples could be neatly structured by introducing:

Comparison of the examples

- Both examples characterized by sequencing of effectful computations.
- Both examples could be neatly structured by introducing:
  - A type denoting computations

Comparison of the examples

- Both examples characterized by sequencing of effectful computations.
- Both examples could be neatly structured by introducing:
  - A type denoting computations
  - A function constructing an effect-free computation of a value

Comparison of the examples

- Both examples characterized by sequencing of effectful computations.
- Both examples could be neatly structured by introducing:
  - A type denoting computations
  - A function constructing an effect-free computation of a value
  - A function constructing a computation by sequencing computations

Comparison of the examples

- Both examples characterized by sequencing of effectful computations.
- Both examples could be neatly structured by introducing:
  - A type denoting computations
  - A function constructing an effect-free computation of a value
  - A function constructing a computation by sequencing computations
- In fact, both examples are instances of the general notion of a ***MONAD***.

Monads in Functional Programming

A monad is represented by:

- A type constructor

  $$M :: * \to *$$

  $M\ T$ represents computations of value of type T.

- A polymorphic function

  $$return :: a \to M\ a$$

  for lifting a value to a computation.

- A polymorphic function

  $$(\gg=) :: M\ a \to (a \to M\ b) \to M\ b$$

  for sequencing computations.

Exercise 2: *join* and *fmap*

Equivalently, the notion of a monad can be captured through the following functions:

$$\begin{aligned} &\mathit{return} :: a \to M\ a \\ &\mathit{join} \quad\ :: (M\ (M\ a)) \to M\ a \\ &\mathit{fmap} \quad :: (a \to b) \to M\ a \to M\ b \end{aligned}$$

join “flattens” a computation, *fmap* “lifts” a function to map computations to computations.

Define *join* and *fmap* in terms of $(\gg\!=)$ (and *return*), and $(\gg\!=)$ in terms of *join* and *fmap*.

$$(\gg\!=) :: M\ a \to (a \to M\ b) \to M\ b$$

Exercise 2: Solution

$$join :: M\ (M\ a) \to M\ a$$
$$join\ mm = mm \gg\!= id$$

$$fmap :: (a \to b) \to M\ a \to M\ b$$
$$fmap\ f\ m = m \gg\!= return \circ f$$

$$(\gg\!=) :: M\ a \to (a \to M\ b) \to M\ b$$
$$m \gg\!= f = join\ (fmap\ f\ m)$$

Monad laws

Additionally, the following ***laws*** must be satisfied:

$$
\begin{aligned}
\mathit{return}\ x \gg\!= f &= f\ x \\
m \gg\!= \mathit{return} &= m \\
(m \gg\!= f) \gg\!= g &= m \gg\!= (\lambda x \to f\ x \gg\!= g)
\end{aligned}
$$

I.e., return is the right and left identity for $(\gg\!=)$, and $(\gg\!=)$ is associative.

Exercise 3: The Identity Monad

The ***Identity Monad*** can be understood as representing ***effect-free*** computations:

$$\mathbf{type}\ I\ a = a$$

1. Provide suitable definitions of $return$ and $(\gg=)$.
2. Verify that the monad laws hold for your definitions.

Exercise 3: Solution

$$return :: a \to I\ a$$
$$return = id$$
$$(\gg=) :: I\ a \to (a \to I\ b) \to I\ b$$
$$m \gg= f = f\ m$$

(Or: $(\gg=) = \mathit{flip}\ (\$)$)

Simple calculations verify the laws, e.g.:

$$\begin{aligned} return\ x \gg= f &= id\ x \gg= f \\ &= x \gg= f \\ &= f\ x \end{aligned}$$

Reading

- Philip Wadler. The Essence of Functional Programming. *Proceedings of the 19th ACM Symposium on Principles of Programming Languages (POPL'92)*, 1992.
- Nick Benton, John Hughes, Eugenio Moggi. Monads and Effects. In *International Summer School on Applied Semantics 2000*, Caminha, Portugal, 2000.
- *All About Monads.* `http://www.haskell.org/all_about_monads`

COMP4075/G54RFP: Lecture 8

Monads in Haskell

Henrik Nilsson

University of Nottingham, UK

This Lecture

- Monads in Haskell
- The Haskell Monad Class Hierarchy
- Some Standard Monads and Library Functions

Monads in Haskell (1)

In Haskell, the notion of a monad is captured by a ***Type Class***. In principle (but not quite from GHC 7.8 onwards):

$$
\begin{array}{l}
\textbf{class}\ Monad\ m\ \textbf{where} \\
\quad return :: a \to m\ a \\
\quad (\gg\!=)\ \ :: m\ a \to (a \to m\ b) \to m\ b
\end{array}
$$

Allows names of the common functions to be overloaded and sharing of derived definitions.

Monads in Haskell (2)

The Haskell monad class has two further methods with default definitions:

$$(\gg) :: m\ a \to m\ b \to m\ b$$
$$m \gg k = m \ggg\!= \lambda\_ \to k$$
$$\mathit{fail} :: \mathit{String} \to m\ a$$
$$\mathit{fail}\ s = \mathit{error}\ s$$

(However, fail will likely be moved into a separate class $\mathit{MonadFail}$ in the future.)

The *Maybe* Monad in Haskell

$$
\begin{array}{l}
\textbf{instance}\ \mathit{Monad}\ \mathit{Maybe}\ \textbf{where} \\
\quad \mathit{return} = \mathit{Just} \\
\quad \mathit{Nothing} \gg\!= \_ = \mathit{Nothing} \\
\quad (\mathit{Just}\ x) \gg\!= f = f\ x
\end{array}
$$

The Monad Type Class Hierachy (1)

Monads are mathematically related to two other notions:

- Functors
- Applicative Functors (or just Applicatives)

Every monad is an applicative functor, and every applicative functor (and thus monad) is a functor.

Class hierarchy:

$$\textbf{class}\ \mathit{Functor}\ f\ \textbf{where}\ldots$$
$$\textbf{class}\ \mathit{Functor}\ f \Rightarrow \mathit{Applicative}\ f\ \textbf{where}\ldots$$
$$\textbf{class}\ \mathit{Applicative}\ m \Rightarrow \mathit{Monad}\ m\ \textbf{where}\ldots$$

The Monad Type Class Hierachy (2)

For example, fmap can be defined in terms of $\gg\!=$ and return, demonstrating that a monad is a functor:

$$\mathit{fmap}\ f\ m = m \gg\!= \lambda x \rightarrow \mathit{return}\ (f\ x)$$

The Monad Type Class Hierachy (2)

For example, $fmap$ can be defined in terms of $\gg=$ and $return$, demonstrating that a monad is a functor:

$$fmap\ f\ m = m \gg= \lambda x \rightarrow return\ (f\ x)$$

A consequence of this class hierarchy is that to make some T an instance of $Monad$, an instance of T for both $Functor$ and $Applicative$ must also be provided.

The Monad Type Class Hierachy (2)

For example, fmap can be defined in terms of $\gg\!=$ and return, demonstrating that a monad is a functor:

$$\mathit{fmap}\ f\ m = m \gg\!= \lambda x \rightarrow \mathit{return}\ (f\ x)$$

A consequence of this class hierarchy is that to make some T an instance of Monad, an instance of T for both $\mathit{Functor}$ and $\mathit{Applicative}$ must also be provided.

Note: Not a mathematical necessity, but a result of how these notions are defined in Haskell at present. E.g. monads can be understood in isolation.

Applicative Functors (1)

An applicative functor is a functor with application, providing operations to:

- embed pure expressions ($pure$), and
- sequence computations and combine their results ($\circledast$)

$$
\begin{array}{l}
\mathbf{class}\ \mathit{Functor}\ f \Rightarrow \mathit{Applicative}\ f\ \mathbf{where} \\
\quad \begin{array}{ll}
pure & :: a \to f\ a \\
(\circledast) & :: f\ (a \to b) \to f\ a \to f\ b \\
(\ast\!\!>) & :: f\ a \to f\ b \to f\ b \\
(<\!\!\ast) & :: f\ a \to f\ b \to f\ a
\end{array}
\end{array}
$$

Applicative Functors (2)

- Like monads, applicative functors is a notion of computation.

Applicative Functors (2)

- Like monads, applicative functors is a notion of computation.
- The key difference is that the result of one computation is not made available to subsequent computations. As a result:
  - The ***structure*** of a computation is static.
  - Scope for running computations in ***parallel***.

Applicative Functors (3)

Laws:

$$\begin{array}{rcl} pure\ id \mathbin{<\!\!*\!\!>} v & = & v \\ pure\ (\circ) \mathbin{<\!\!*\!\!>} u \mathbin{<\!\!*\!\!>} v \mathbin{<\!\!*\!\!>} w & = & u \mathbin{<\!\!*\!\!>} (v \mathbin{<\!\!*\!\!>} w) \\ pure\ f \mathbin{<\!\!*\!\!>} pure\ x & = & pure\ (f\ x) \\ u \mathbin{<\!\!*\!\!>} pure\ y & = & pure\ (\$y) \mathbin{<\!\!*\!\!>} u \end{array}$$

Applicative Functors (3)

Laws:

$$\begin{aligned}
pure\ id \ {<}{*}{>}\ v &= v \\
pure\ (\circ)\ {<}{*}{>}\ u\ {<}{*}{>}\ v\ {<}{*}{>}\ w &= u\ {<}{*}{>}\ (v\ {<}{*}{>}\ w) \\
pure\ f\ {<}{*}{>}\ pure\ x &= pure\ (f\ x) \\
u\ {<}{*}{>}\ pure\ y &= pure\ (\$y)\ {<}{*}{>}\ u
\end{aligned}$$

Default definitions:

$$\begin{aligned}
u\ {*}{>}\ v &= pure\ (const\ id)\ {<}{*}{>}\ u\ {<}{*}{>}\ v \\
u\ {<}{*}\ v &= pure\ const\ {<}{*}{>}\ u\ {<}{*}{>}\ v
\end{aligned}$$

Instances of $Applicative$

$$
\begin{array}{l}
\textbf{instance}\ Applicative\ [\,]\ \textbf{where} \\
\quad pure\ x = [x] \\
\quad fs \circledast xs = [f\ x \mid f \leftarrow fs, x \leftarrow xs]
\end{array}
$$

Instances of $Applicative$

$$
\begin{array}{l}
\textbf{instance}\ Applicative\ [\,]\ \textbf{where} \\
\quad pure\ x = [x] \\
\quad fs \circledast xs = [f\ x \mid f \leftarrow fs, x \leftarrow xs]
\end{array}
$$

$$
\begin{array}{l}
\textbf{instance}\ Applicative\ Maybe\ \textbf{where} \\
\quad pure = Just \\
\quad Just\ f \quad \circledast m = fmap\ f\ m \\
\quad Nothing \circledast \_ \ = Nothing
\end{array}
$$

Class *Alternative*

The class *Alternative* is a monoid on applicative functors:

$$
\begin{array}{l}
\textbf{class}\ \mathit{Applicative}\ f \Rightarrow \mathit{Alternative}\ f\ \textbf{where} \\
\quad \mathit{empty} :: f\ a \\
\quad (<\!|\!>)\ :: f\ a \rightarrow f\ a \rightarrow f\ a \\
\quad \mathit{some}\ \ :: f\ a \rightarrow f\ [a] \\
\quad \mathit{many}\ :: f\ a \rightarrow f\ [a] \\
\quad \mathit{some}\ v\ = \mathit{pure}\ (:)\ <\!\!*\!\!>\ v\ <\!\!*\!\!>\ \mathit{many}\ v \\
\quad \mathit{many}\ v = \mathit{some}\ v\ <\!|\!>\ \mathit{pure}\ [\,]
\end{array}
$$

Class *Alternative*

The class *Alternative* is a monoid on applicative functors:

$$
\begin{aligned}
&\mathbf{class}\ \mathit{Applicative}\ f \Rightarrow \mathit{Alternative}\ f\ \mathbf{where}\\
&\quad \mathit{empty} :: f\ a\\
&\quad (<\!|\!>)\ :: f\ a \to f\ a \to f\ a\\
&\quad \mathit{some}\ \ :: f\ a \to f\ [a]\\
&\quad \mathit{many}\ :: f\ a \to f\ [a]\\
&\quad \mathit{some}\ v\ \ = \mathit{pure}\ (:)\ <\!\!*\!\!>\ v\ <\!\!*\!\!>\ \mathit{many}\ v\\
&\quad \mathit{many}\ v = \mathit{some}\ v\ <\!|\!>\ \mathit{pure}\ [\,]
\end{aligned}
$$

$<\!|\!>$ can be understood as “one or the other”, *some* as “at least one”, and *many* as “zero or more”.

Instances of *Alternative*

$$
\begin{array}{l}
\textbf{instance}\ \mathit{Alternative}\ [\,]\ \textbf{where} \\
\quad \mathit{empty} = [\,] \\
\quad (\langle|\rangle) = (+\!\!+)
\end{array}
$$

Instances of *Alternative*

$$
\begin{array}{l}
\textbf{instance } \mathit{Alternative}\ [\,]\ \textbf{where} \\
\quad \mathit{empty} = [\,] \\
\quad (<\!|\!>) = (+\!\!+)
\end{array}
$$

$$
\begin{array}{l}
\textbf{instance } \mathit{Alternative}\ \mathit{Maybe}\ \textbf{where} \\
\quad \mathit{empty} = \mathit{Nothing} \\
\quad \mathit{Nothing} <\!|\!> r = r \\
\quad l \qquad\qquad\ \, <\!|\!> \_ = l
\end{array}
$$

Example: Applicative Parser (1)

Applicative functors are frequently used in the context of parsing combinators. In fact, that is where their origin lies.

Example: Applicative Parser (1)

Applicative functors are frequently used in the context of parsing combinators. In fact, that is where their origin lies.

A $Parser$ computation allows reading of input and trying alternatives:

$$\textbf{instance } Applicative\ Parser \textbf{ where} \ldots$$
$$\textbf{instance } Alternative\ Parser \textbf{ where} \ldots$$

Example: Applicative Parser (2)

$$
\begin{array}{l}
\mathit{command} :: \mathit{Parser}\ \mathit{Command} \\
\mathit{command} = \\
\qquad\qquad \mathit{pure}\ \mathit{If} \\
\qquad\qquad\ \ \lessdot\!\ast \quad \mathit{kwd}\ \texttt{"if"} \qquad \lessdot\!\ast\!\gtrdot\ \mathit{expr} \\
\qquad\qquad\ \ \lessdot\!\ast \quad \mathit{kwd}\ \texttt{"then"}\ \lessdot\!\ast\!\gtrdot\ \mathit{command} \\
\qquad\qquad\ \ \lessdot\!\ast \quad \mathit{kwd}\ \texttt{"else"}\ \lessdot\!\ast\!\gtrdot\ \mathit{command} \\
\qquad <\!|\!>\ \mathit{pure}\ \mathit{Block} \\
\qquad\qquad\ \ \lessdot\!\ast \quad \mathit{kwd}\ \texttt{"begin"} \\
\qquad\qquad\ \ \lessdot\!\ast\!\gtrdot\ \mathit{some}\ (\mathit{command}\ \ast\!\gtrdot\ \mathit{symb}\ \texttt{";"}) \\
\qquad\qquad\ \ \lessdot\!\ast \quad \mathit{kwd}\ \texttt{"end"}
\end{array}
$$

...

Applicative Functors and Monads

A requirement is $return = pure$.

In fact, the $Monad$ class provides a default definition of $return$ defined that way:

$$\textbf{class}\ Applicative\ m \Rightarrow Monad\ m\ \textbf{where}$$
$$return :: a \to m\ a$$
$$return = pure$$
$$(\gg\!=) :: m\ a \to (a \to m\ b) \to m\ b$$

Exercise: A State Monad in Haskell

Recall that a type $Int \rightarrow (a, Int)$ can be viewed as a state monad.

Haskell 2010 does not permit type synonyms to be instances of classes. Hence we have to define a new type:

$$\mathbf{newtype}\ S\ a = S\ \{\, unS :: (Int \rightarrow (a, Int))\,\}$$

Thus: $unS :: S\ a \rightarrow (Int \rightarrow (a, Int))$

Provide a $Functor$, $Applicative$, and $Monad$ instance for S.

Solution: *Functor* Instance

$$
\begin{array}{l}
\textbf{instance}\ \mathit{Functor}\ S\ \textbf{where} \\
\quad \mathit{fmap}\ f\ \ \mathit{sa} = S\ \$\ \lambda s \rightarrow \\
\qquad \textbf{let} \\
\qquad\quad (a, s') = \mathit{unS}\ \ \mathit{sa}\ \ s \\
\qquad \textbf{in} \\
\qquad\quad (f\ \ a, s')
\end{array}
$$

Solution: *Applicative* Instance

$$
\begin{array}{l}
\mathbf{instance}\ \mathit{Applicative}\ S\ \mathbf{where} \\
\quad \mathit{pure}\ a = S\ \$\ \lambda s \rightarrow (a, s) \\
\quad \mathit{sf}\ \ll\!\!\ast\!\!\gg\ \mathit{sa} = S\ \$\ \lambda s \rightarrow \\
\qquad \mathbf{let} \\
\qquad\quad (f, s') = \mathit{unS}\ \mathit{sf}\ s \\
\qquad \mathbf{in} \\
\qquad\quad \mathit{unS}\ (\mathit{fmap}\ f\ \mathit{sa})\ s'
\end{array}
$$

Solution: *Monad* Instance

$$
\begin{array}{l}
\textbf{instance}\ \mathit{Monad}\ S\ \textbf{where} \\
\quad m \gg\!= f = S\ \$\ \lambda s \rightarrow \\
\qquad \textbf{let}\ (a, s') = \mathit{unS}\ m\ s \\
\qquad \textbf{in}\ \mathit{unS}\ (f\ a)\ s'
\end{array}
$$

(Using the default definition `return = pure`.)

The List Monad

Computation with many possible results, "nondeterminism":

$$\begin{aligned}
&\textbf{instance } \mathit{Monad}\ [\,]\ \textbf{where} \\
&\quad \mathit{return}\ a = [\,a\,] \\
&\quad m \gg\!= f = \mathit{concat}\ (\mathit{map}\ f\ m) \\
&\quad \mathit{fail}\ s \quad\ \ = [\,]
\end{aligned}$$

Example:

$$\begin{aligned}
&x \leftarrow [1,2] \\
&y \leftarrow [\,\texttt{'a'},\texttt{'b'}\,] \\
&\mathit{return}\ (x, y)
\end{aligned}$$

Result:

$$\begin{aligned}
&[(1,\texttt{'a'}),(1,\texttt{'b'}), \\
&(2,\texttt{'a'}),(2,\texttt{'b'})]
\end{aligned}$$

The Reader Monad

Computation in an environment:

$$\textbf{instance } \mathit{Monad}\ ((\to)\ e)\ \textbf{where}$$
$$\quad \mathit{return}\ a = \mathit{const}\ a$$
$$\quad m \gg\!= f = \lambda e \to f\ (m\ e)\ e$$

$$\mathit{getEnv} :: ((\to)\ e)\ e$$
$$\mathit{getEnv} = \mathit{id}$$

Monad-specific Operations (1)

To be useful, monads need to be equipped with additional operations specific to the effects in question. For example:

$$
\begin{aligned}
&\mathit{fail} :: \mathit{String} \rightarrow \mathit{Maybe}\ a \\
&\mathit{fail}\ s = \mathit{Nothing} \\
&\mathit{catch} :: \mathit{Maybe}\ a \rightarrow \mathit{Maybe}\ a \rightarrow \mathit{Maybe}\ a \\
&m1\ \text{`}\mathit{catch}\text{`}\ m2 = \\
&\quad \textbf{case}\ m1\ \textbf{of} \\
&\qquad \mathit{Just}\ \_ \quad \rightarrow m1 \\
&\qquad \mathit{Nothing} \rightarrow m2
\end{aligned}
$$

Monad-specific Operations (2)

Typical operations on a state monad:

$$set :: Int \to S\ ()$$
$$set\ a = S\ (\lambda\_ \to ((), a))$$
$$get :: S\ Int$$
$$get = S\ (\lambda s \to (s, s))$$

Moreover, need to "run" a computation. E.g.:

$$runS :: S\ a \to a$$
$$runS\ m = fst\ (unS\ m\ 0)$$

The do-notation (1)

Haskell provides convenient syntax for programming with monads:

$$
\begin{aligned}
&\mathbf{do} \\
&\quad a \leftarrow exp_1 \\
&\quad b \leftarrow exp_2 \\
&\quad return\ exp_3
\end{aligned}
$$

is syntactic sugar for

$$
\begin{aligned}
&exp_1 \gg\!= \lambda a \rightarrow \\
&exp_2 \gg\!= \lambda b \rightarrow \\
&return\ exp_3
\end{aligned}
$$

Note: a in scope in exp_2, a and b in exp_3.

The do-notation (2)

Computations can be done solely for effect, ignoring the computed value:

$$
\begin{array}{l}
\mathbf{do} \\
\quad exp_1 \\
\quad exp_2 \\
\quad return\ exp_3
\end{array}
$$

is syntactic sugar for

$$
\begin{array}{l}
exp_1 \gg\!= \lambda\_ \rightarrow \\
exp_2 \gg\!= \lambda\_ \rightarrow \\
return\ exp_3
\end{array}
$$

The do-notation (3)

A $\mathbf{let}$-construct is also provided:

$$
\begin{array}{l}
\mathbf{do} \\
\quad \mathbf{let}\ a = exp_1 \\
\quad \phantom{\mathbf{let}\ } b = exp_2 \\
\quad return\ exp_3
\end{array}
$$

is equivalent to

$$
\begin{array}{l}
\mathbf{do} \\
\quad a \leftarrow return\ exp_1 \\
\quad b \leftarrow return\ exp_2 \\
\quad return\ exp_3
\end{array}
$$

Numbering Trees in do-notation

$$
\begin{array}{l}
numberTree\ t = runS\ (ntAux\ t) \\
\quad \textbf{where} \\
\quad\quad ntAux :: Tree\ a \to S\ (Tree\ Int) \\
\quad\quad ntAux\ (Leaf\ \_) = \textbf{do} \\
\quad\quad\quad n \leftarrow get \\
\quad\quad\quad set\ (n + 1) \\
\quad\quad\quad return\ (Leaf\ n) \\
\quad\quad ntAux\ (Node\ t1\ t2) = \textbf{do} \\
\quad\quad\quad t1' \leftarrow ntAux\ t1 \\
\quad\quad\quad t2' \leftarrow ntAux\ t2 \\
\quad\quad\quad return\ (Node\ t1'\ t2')
\end{array}
$$

Applicative do-notation (1)

A variation of the **do**-notation is also available for applicatives:

$$
\begin{array}{l}
\mathbf{do} \\
\quad a \leftarrow exp_1 \\
\quad b \leftarrow exp_2 \\
\quad return\ (\ldots a \ldots b \ldots)
\end{array}
$$

Note that the bound variables may only be used in the $return$-expression, or the code becomes monadic.

In this case, a must not occur in exp_2.

Applicative do-notation (2)

For example, an applicative parser:

$$
\begin{array}{l}
\mathit{commandIf} :: \mathit{Parser}\ \mathit{Command} \\
\mathit{commandIf} = \\
\quad \mathit{kwd}\ \texttt{"if"} \\
\quad c \leftarrow \mathit{expr} \\
\quad \mathit{kwd}\ \texttt{"then"} \\
\quad t \leftarrow \mathit{command} \\
\quad \mathit{kwd}\ \texttt{"else"} \\
\quad e \leftarrow \mathit{command} \\
\quad \mathit{return}\ (\mathit{If}\ c\ t\ e)
\end{array}
$$

Monadic Utility Functions

Some monad utilities:

$$
\begin{array}{ll}
\mathit{sequence} :: \mathit{Monad}\ m \Rightarrow [m\ a] \rightarrow m\ [a] \\
\mathit{sequence\_} :: \mathit{Monad}\ m \Rightarrow [m\ a] \rightarrow m\ () \\
\mathit{mapM} & :: \mathit{Monad}\ m \Rightarrow (a \rightarrow m\ b) \rightarrow [a] \rightarrow m\ [b] \\
\mathit{mapM\_} & :: \mathit{Monad}\ m \Rightarrow (a \rightarrow m\ b) \rightarrow [a] \rightarrow m\ () \\
\mathit{when} & :: \mathit{Monad}\ m \Rightarrow \mathit{Bool} \rightarrow m\ () \rightarrow m\ () \\
\mathit{foldM} & :: \mathit{Monad}\ m \Rightarrow \\
& \quad (a \rightarrow b \rightarrow m\ a) \rightarrow a \rightarrow [b] \rightarrow m\ a \\
\mathit{liftM} & :: \mathit{Monad}\ m \Rightarrow (a \rightarrow b) \rightarrow m\ a \rightarrow m\ b \\
\mathit{liftM2} & :: \mathit{Monad}\ m \Rightarrow \\
& \quad (a \rightarrow b \rightarrow c) \rightarrow m\ a \rightarrow m\ b \rightarrow m\ c
\end{array}
$$

The Haskell IO Monad (1)

In Haskell, IO is handled through the IO monad.
IO is ***abstract***! Conceptually:

$$\mathbf{newtype}\ IO\ a = IO\ (World \to (a, World))$$

Some operations:

$$\begin{aligned}
&putChar &&:: Char \to IO\ () \\
&putStr &&:: String \to IO\ () \\
&putStrLn &&:: String \to IO\ () \\
&getChar &&:: IO\ Char \\
&getLine &&:: IO\ String \\
&getContents &&:: String
\end{aligned}$$

The Haskell IO Monad (2)

IO essentially provides all effects of typical imperative languages. Besides input/output:

- Pointers and imperative state (through $IORef$)
- Raising and handling exceptions
- Concurrency
- Foreign function interface

The Haskell IO Monad (2)

IO essentially provides all effects of typical imperative languages. Besides input/output:

- Pointers and imperative state (through $IORef$)
- Raising and handling exceptions
- Concurrency
- Foreign function interface

IO is sometimes referred to as the "sin bin"!

The ST Monad: "Real" State

The ST monad (common Haskell extension) provides real, imperative state behind the scenes to allow efficient implementation of imperative algorithms:

$$\textbf{data}\ ST\ s\ a \quad \text{-- abstract}$$
$$\textbf{instance}\ Monad\ (ST\ s)$$
$$newSTRef\ \ ::\ s\ ST\ a\ (STRef\ s\ a)$$
$$readSTRef\ \ ::\ STRef\ s\ a \to ST\ s\ a$$
$$writeSTRef ::\ STRef\ s\ a \to a \to ST\ s\ ()$$
$$runST ::\ (forall\ s\ .\ st\ s\ a) \to a$$

ST vs IO

Why use ST if IO also gives access to imperative state?

ST vs IO

Why use ST if IO also gives access to imperative state?

- ST much more focused: provides only state, not a lot more besides.

ST vs IO

Why use ST if IO also gives access to imperative state?

- ST much more focused: provides only state, not a lot more besides.
- ST computations can be run safely inside pure code.

ST vs IO

Why use ST if IO also gives access to imperative state?

- ST much more focused: provides only state, not a lot more besides.
- ST computations can be run safely inside pure code.

It ***is*** possible to run IO comp. inside pure code:

$$unsafePerformIO :: IO\ a \rightarrow a$$

But make sure you know what you are doing!

Reading

- Philip Wadler. The Essence of Functional Programming. *Proceedings of the 19th ACM Symposium on Principles of Programming Languages (POPL'92)*, 1992.
- Nick Benton, John Hughes, Eugenio Moggi. Monads and Effects. In *International Summer School on Applied Semantics 2000*, Caminha, Portugal, 2000.

COMP4075/G54RFP: Lecture 9

Concurrency

Henrik Nilsson

University of Nottingham, UK

This Lecture

- A concurrency monad (adapted from Claessen (1999))
- Basic concurrent programming in Haskell
- Software Transactional Memory (the STM monad)

A Concurrency Monad (1)

A Thread represents a (branching) process: a stream of primitive ***atomic*** operations:

$$
\begin{aligned}
\mathbf{data}\ \mathit{Thread} = {} & \mathit{Print}\ \mathit{Char}\ \mathit{Thread} \\
\mid {} & \mathit{Fork}\ \mathit{Thread}\ \mathit{Thread} \\
\mid {} & \mathit{End}
\end{aligned}
$$

A Concurrency Monad (1)

A $Thread$ represents a (branching) process: a stream of primitive ***atomic*** operations:

$$\begin{aligned}\textbf{data}\ Thread = {} & Print\ Char\ Thread \\ \mid {} & Fork\ Thread\ Thread \\ \mid {} & End\end{aligned}$$

Note that a $Thread$ represents the ***entire rest*** of a computation.

Note also that a $Thread$ can spawn other $Thread$s (so we get a tree, if you prefer).

A Concurrency Monad (2)

Introduce a monad representing “interleavable computations”. At this stage, this amounts to little more than a convenient way to construct threads by sequential composition.

A Concurrency Monad (2)

Introduce a monad representing “interleavable computations”. At this stage, this amounts to little more than a convenient way to construct threads by sequential composition.

How can Threads be constructed sequentially? The only way is to parameterize thread prefixes on the rest of the Thread. This leads directly to ***continuations***.

A Concurrency Monad (3)

$$\textbf{newtype}\ CM\ a = CM\ ((a \to Thread) \to Thread)$$

$$fromCM :: CM\ a \to ((a \to Thread) \to Thread)$$
$$fromCM\ (CM\ x) = x$$

$$thread :: CM\ a \to Thread$$
$$thread\ m = fromCM\ m\ (const\ End)$$

$$\textbf{instance}\ Monad\ CM\ \textbf{where}$$
$$\quad return\ x = CM\ (\lambda k \to k\ x)$$
$$\quad m \gg\!= f \ = CM\ \$\ \lambda k \to$$
$$\qquad fromCM\ m\ (\lambda x \to fromCM\ (f\ x)\ k)$$

A Concurrency Monad (4)

Atomic operations:

$$cPrint :: Char \to CM\ ()$$
$$cPrint\ c = CM\ (\lambda k \to Print\ c\ (k\ ()))$$
$$cFork :: CM\ a \to CM\ ()$$
$$cFork\ m = CM\ (\lambda k \to Fork\ (thread\ m)\ (k\ ()))$$
$$cEnd :: CM\ a$$
$$cEnd = CM\ (\backslash\_ \to End)$$

Running a Concurrent Computation (1)

$$
\begin{array}{l}
\mathbf{type}\ \mathit{Output} = [\,\mathit{Char}\,] \\
\mathbf{type}\ \mathit{ThreadQueue} = [\,\mathit{Thread}\,] \\
\mathbf{type}\ \mathit{State} = (\mathit{Output}, \mathit{ThreadQueue}) \\
\mathit{runCM} :: \mathit{CM}\ a \to \mathit{Output} \\
\mathit{runCM}\ m = \mathit{runHlp}\ (\texttt{""}, [\,])\ (\mathit{thread}\ m) \\
\quad \mathbf{where} \\
\quad\quad \mathit{runHlp}\ s\ t = \\
\quad\quad\quad \mathbf{case}\ \mathit{dispatch}\ s\ t\ \mathbf{of} \\
\quad\quad\quad\quad \mathit{Left}\ (s', t) \to \mathit{runHlp}\ s'\ t \\
\quad\quad\quad\quad \mathit{Right}\ o \to o
\end{array}
$$

Running a Concurrent Computation (2)

Dispatch on the operation of the currently running Thread. Then call the scheduler.

$$
\begin{aligned}
&\mathit{dispatch} :: \mathit{State} \to \mathit{Thread} \\
&\quad \to \mathit{Either}\ (\mathit{State}, \mathit{Thread})\ \mathit{Output} \\
&\mathit{dispatch}\ (o, rq)\ (\mathit{Print}\ c\ t) = \\
&\quad \mathit{schedule}\ (o + [c], rq + [t]) \\
&\mathit{dispatch}\ (o, rq)\ (\mathit{Fork}\ t1\ t2) = \\
&\quad \mathit{schedule}\ (o, rq + [t1, t2]) \\
&\mathit{dispatch}\ (o, rq)\ \mathit{End} = \\
&\quad \mathit{schedule}\ (o, rq)
\end{aligned}
$$

Running a Concurrent Computation (3)

Selects next $Thread$ to run, if any.

$$
\begin{aligned}
&schedule :: State \rightarrow Either\ (State, Thread) \\
&\qquad\qquad\qquad\qquad\qquad\qquad Output \\
&schedule\ (o, [\,]) \qquad = Right\ o \\
&schedule\ (o, t : ts) = Left\ ((o, ts), t)
\end{aligned}
$$

Running a Concurrent Computation (3)

Selects next $Thread$ to run, if any.

$$\begin{aligned}
&schedule :: State \to Either\ (State, Thread) \\
&\qquad\qquad\qquad\qquad\qquad\qquad Output \\
&schedule\ (o, [\,]) \qquad = Right\ o \\
&schedule\ (o, t : ts) = Left\ ((o, ts), t)
\end{aligned}$$

This all amounts to a ***topological sorting*** of the nodes in the $Thread$-tree.

Example: Concurrent Processes

$$
\begin{array}{lll}
p1 :: CM\ () & p2 :: CM\ () & p3 :: CM\ () \\
p1 = \mathbf{do} & p2 = \mathbf{do} & p3 = \mathbf{do} \\
\quad cPrint\ \texttt{'a'} & \quad cPrint\ \texttt{'1'} & \quad cFork\ p1 \\
\quad cPrint\ \texttt{'b'} & \quad cPrint\ \texttt{'2'} & \quad cPrint\ \texttt{'A'} \\
\quad \ldots & \quad \ldots & \quad cFork\ p2 \\
\quad cPrint\ \texttt{'j'} & \quad cPrint\ \texttt{'0'} & \quad cPrint\ \texttt{'B'}
\end{array}
$$

$$main = print\ (runCM\ p3)$$

Result: `aAbc1Bd2e3f4g5h6i7j890`

Note: As it stands, the output is only made available after ***all*** threads have terminated.)

Incremental Output

Incremental output:

$$
\begin{aligned}
&\mathit{runCM} :: \mathit{CM}\ a \to \mathit{Output}\\
&\mathit{runCM}\ m = \mathit{dispatch}\ [\,]\ (\mathit{thread}\ m)
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{dispatch} :: \mathit{ThreadQueue} \to \mathit{Thread} \to \mathit{Output}\\
&\mathit{dispatch}\ \mathit{rq}\ (\mathit{Print}\ c\ t) &&= c : \mathit{schedule}\ (\mathit{rq} +\!\!+ [t])\\
&\mathit{dispatch}\ \mathit{rq}\ (\mathit{Fork}\ t1\ t2) &&= \mathit{schedule}\ (\mathit{rq} +\!\!+ [t1, t2])\\
&\mathit{dispatch}\ \mathit{rq}\ \mathit{End} &&= \mathit{schedule}\ \mathit{rq}
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{schedule} :: \mathit{ThreadQueue} \to \mathit{Output}\\
&\mathit{schedule}\ [\,] &&= [\,]\\
&\mathit{schedule}\ (t : \mathit{ts}) &&= \mathit{dispatch}\ \mathit{ts}\ t
\end{aligned}
$$

Example: Concurrent processes 2

$$
\begin{array}{lll}
p1 :: CM\ () & p2 :: CM\ () & p3 :: CM\ () \\
p1 = \mathbf{do} & p2 = \mathbf{do} & p3 = \mathbf{do} \\
\quad cPrint\ \mathtt{'a'} & \quad cPrint\ \mathtt{'1'} & \quad cFork\ p1 \\
\quad cPrint\ \mathtt{'b'} & \quad undefined & \quad cPrint\ \mathtt{'A'} \\
\quad \ldots & \quad \ldots & \quad cFork\ p2 \\
\quad cPrint\ \mathtt{'j'} & \quad cPrint\ \mathtt{'0'} & \quad cPrint\ \mathtt{'B'}
\end{array}
$$

$$main = print\ (runCM\ p3)$$

Result: $aAbc1Bd *** Exception : Prelude.undefined$

Any Use?

- A number of libraries and embedded langauges use similar ideas, e.g.
  - Fudgets: A GUI library
  - Yampa: A FRP library
- Studying semantics of concurrent programs.
- Aid for testing, debugging, and reasoning about concurrent programs.

Concurrent Programming in Haskell

Primitives for concurrent programming provided as operations of the IO monad. They are in the module $Control.Concurrent$. Excerpts:

$$
\begin{array}{ll}
forkIO & :: IO\ () \to IO\ \ ThreadId \\
killThread & :: ThreadId \to IO\ () \\
threadDelay & :: Int \to IO\ () \\
newMVar & :: a \to IO\ (MVar\ \ a) \\
newEmptyMVar & :: IO\ (MVar\ \ a) \\
putMVar & :: MVar\ \ a \to a \to IO\ () \\
takeMVar & :: MVar\ \ a \to IO\ \ a
\end{array}
$$

$MVars$

- The fundamental synchronisation mechanism is the $MVar$ (“em-var”).
- An $MVar$ is a “one-item box” that may be ***empty*** or ***full***.
- Reading ($takeMVar$) and writing ($putMVar$) are ***atomic*** operations:
  - Writing to an empty $MVar$ makes it full.
  - Writing to a full $MVar$ blocks.
  - Reading from an empty $MVar$ blocks.
  - Reading from a full $MVar$ makes it empty.

Example: Basic Synchronization (1)

$$\textbf{module}\ \mathit{Main}\ \textbf{where}$$

$$\textbf{import}\ \mathit{Control.Concurrent}$$

$$\mathit{countFromTo} :: \mathit{Int} \to \mathit{Int} \to \mathit{IO}\ ()$$
$$\mathit{countFromTo}\ m\ n$$
$$\quad \mid m > n \qquad = \mathit{return}\ ()$$
$$\quad \mid \mathit{otherwise} = \textbf{do}$$
$$\qquad\quad \mathit{putStrLn}\ (\mathit{show}\ m)$$
$$\qquad\quad \mathit{countFromTo}\ (m+1)\ n$$

Example: Basic Synchronization (2)

```
main = do
  start ← newEmptyMVar
  done ← newEmptyMVar
  forkIO $ do
    takeMVar start
    countFromTo 1 10
    putMVar done ()
  putStrLn "Go!"
  putMVar start ()
  takeMVar done
  countFromTo 11 20
  putStrLn "Done!"
```

Example: Unbounded Buffer (1)

$$\textbf{module } \mathit{Main} \textbf{ where}$$

$$\textbf{import } \mathit{Control.Monad}\ (\mathit{when})$$
$$\textbf{import } \mathit{Control.Concurrent}$$

$$\textbf{newtype } \mathit{Buffer}\ a =$$
$$\quad \mathit{Buffer}\ (\mathit{MVar}\ (\mathit{Either}\ [a]\ (\mathit{Int}, \mathit{MVar}\ a)))$$

$$\mathit{newBuffer} :: \mathit{IO}\ (\mathit{Buffer}\ a)$$
$$\mathit{newBuffer} = \textbf{do}$$
$$\quad b \leftarrow \mathit{newMVar}\ (\mathit{Left}\ [\,])$$
$$\quad \mathit{return}\ (\mathit{Buffer}\ b)$$

Example: Unbounded Buffer (2)

$$
\begin{array}{l}
\mathit{readBuffer} :: \mathit{Buffer}\ a \to \mathit{IO}\ a \\
\mathit{readBuffer}\ (\mathit{Buffer}\ b) = \mathbf{do} \\
\quad \mathit{bc} \leftarrow \mathit{takeMVar}\ b \\
\quad \mathbf{case}\ \mathit{bc}\ \mathbf{of} \\
\qquad \mathit{Left}\ (x : \mathit{xs}) \to \mathbf{do} \\
\qquad\quad \mathit{putMVar}\ b\ (\mathit{Left}\ \mathit{xs}) \\
\qquad\quad \mathit{return}\ x \\
\qquad \mathit{Left}\ [\,] \to \mathbf{do} \\
\qquad\quad w \leftarrow \mathit{newEmptyMVar} \\
\qquad\quad \mathit{putMVar}\ b\ (\mathit{Right}\ (1, w)) \\
\qquad\quad \mathit{takeMVar}\ w
\end{array}
$$

Example: Unbounded Buffer (3)

$$\ldots$$

$$Right\ (n, w) \rightarrow \mathbf{do}$$

$$\quad putMVar\ b\ (Right\ (n + 1, w))$$

$$\quad takeMVar\ w$$

Example: Unbounded Buffer (4)

$$
\begin{array}{l}
\mathit{writeBuffer} :: \mathit{Buffer}\ a \to a \to \mathit{IO}\ () \\
\mathit{writeBuffer}\ (\mathit{Buffer}\ b)\ x = \textbf{do} \\
\quad \mathit{bc} \leftarrow \mathit{takeMVar}\ b \\
\quad \textbf{case}\ \mathit{bc}\ \textbf{of} \\
\qquad \mathit{Left}\ \mathit{xs} \to \\
\qquad\quad \mathit{putMVar}\ b\ (\mathit{Left}\ (\mathit{xs} +\!\!+ [x])) \\
\qquad \mathit{Right}\ (n, w) \to \textbf{do} \\
\qquad\quad \mathit{putMVar}\ w\ x \\
\qquad\quad \textbf{if}\ n > 1 \\
\qquad\quad \textbf{then}\ \mathit{putMVar}\ b\ (\mathit{Right}\ (n - 1, w)) \\
\qquad\quad \textbf{else}\ \ \mathit{putMVar}\ b\ (\mathit{Left}\ [\,])
\end{array}
$$

Example: Unbounded Buffer (4)

The buffer can now be used as a channel of communication between a set of "writers" and a set of "readers". E.g.:

$$
\begin{array}{l}
main = \mathbf{do} \\
\quad b \leftarrow newBuffer \\
\quad forkIO\ (writer\ b) \\
\quad forkIO\ (writer\ b) \\
\quad forkIO\ (reader\ b) \\
\quad forkIO\ (reader\ b) \\
\quad \ldots
\end{array}
$$

Example: Unbounded Buffer (5)

$$
\begin{aligned}
&\mathit{reader} :: \mathit{Buffer}\ \mathit{Int} \to \mathit{IO}\ ()\\
&\mathit{reader}\ n\ b = \mathit{rLoop}\\
&\quad \textbf{where}\\
&\qquad \mathit{rLoop} = \textbf{do}\\
&\qquad\quad x \leftarrow \mathit{readBuffer}\ b\\
&\qquad\quad \mathit{when}\ (x > 0)\ \$\ \textbf{do}\\
&\qquad\qquad \mathit{putStrLn}\ (n +\!\!+ \texttt{": "} +\!\!+ \mathit{show}\ x)\\
&\qquad\qquad \mathit{rLoop}
\end{aligned}
$$

Compositionality? (1)

Suppose we would like to read two ***consecutive*** elements from a buffer `b`?

That is, ***sequential composition***.

Would the following work?

$$x1 \leftarrow readBuffer\ b$$
$$x2 \leftarrow readBuffer\ b$$

Compositionality? (2)

What about this?

$$mutex \leftarrow newMVar\ ()$$
$$\ldots$$
$$takeMVar\ mutex$$
$$x1 \leftarrow readBuffer\ b$$
$$x2 \leftarrow readBuffer\ b$$
$$putMVar\ mutex\ ()$$

Compositionality? (3)

Suppose we would like to read from ***one of two*** buffers.

That is, ***composing alternatives***.

Compositionality? (3)

Suppose we would like to read from ***one of two*** buffers.

That is, ***composing alternatives***.

Hmmm. How do we even begin?

Compositionality? (3)

Suppose we would like to read from ***one of two*** buffers.

That is, ***composing alternatives***.

Hmmm. How do we even begin?

- No way to attempt reading a buffer without risking blocking.

Compositionality? (3)

Suppose we would like to read from ***one of two*** buffers.

That is, ***composing alternatives***.

Hmmm. How do we even begin?

- No way to attempt reading a buffer without risking blocking.
- We have to change or enrich the buffer implementation. E.g. add a $tryReadBuffer$ operation, and then repeatedly poll the two buffers in a tight loop. Not so good!

Software Transactional Memory (1)

- Operations on shared mutable variables grouped into ***transactions***.
- A transaction either succeeds or fails in its ***entirety***. I.e., ***atomic*** w.r.t. other transactions.
- Failed transactions are automatically ***retried*** until they succeed.
- ***Transaction logs***, which records reading and writing of shared variables, maintained to enable transactions to be validated, partial transactions to be rolled back, and to determine when worth trying a transaction again.

Software Transactional Memory (2)

- ***Basic consistency requirement***: The effects of reading and writing within a transaction must be indistinguishable from the transaction having been carried out in isolation.
- ***No locks!*** (At the application level.)

STM and Pure Declarative Languages

- STM perfect match for ***purely declarative languages***:
  - reading and writing of shared mutable variables explicit and relatively rare;
  - most computations are pure and need not be logged.
- Disciplined use of effects through monads a ***huge*** payoff: easy to ensure that ***only*** effects that can be undone can go inside a transaction.

  (Imagine the havoc of arbitrary I/O actions if part of transaction: How to undo? What if retried?)

The STM monad

The software transactional memory abstraction provided by a monad STM. ***Distinct from IO!*** Defined in $Control.Concurrent.STM$.

Excerpts:

$$
\begin{aligned}
newTVar &:: a \rightarrow STM\ (TVar\ a) \\
writeTVar &:: TVar\ a \rightarrow a \rightarrow STM\ () \\
readTVar &:: TVar\ a \rightarrow STM\ a \\
retry &:: STM\ a \\
atomically &:: STM\ a \rightarrow IO\ a
\end{aligned}
$$

Example: Buffer Revisited (1)

Unbounded buffer using the STM monad:

$$\textbf{module}\ \mathit{Main}\ \textbf{where}$$

$$\textbf{import}\ \mathit{Control.Monad}\ (\mathit{when})$$
$$\textbf{import}\ \mathit{Control.Concurrent}$$
$$\textbf{import}\ \mathit{Control.Concurrent.STM}$$

$$\textbf{newtype}\ \mathit{Buffer}\ a = \mathit{Buffer}\ (\mathit{TVar}\ [a])$$

$$\begin{array}{l}
\mathit{newBuffer} :: \mathit{STM}\ (\mathit{Buffer}\ a) \\
\mathit{newBuffer} = \textbf{do} \\
\quad b \leftarrow \mathit{newTVar}\ [\,] \\
\quad \mathit{return}\ (\mathit{Buffer}\ b)
\end{array}$$

Example: Buffer Revisited (2)

$$
\begin{array}{l}
readBuffer :: Buffer\ a \rightarrow STM\ \ a \\
readBuffer\ (Buffer\ b) = \mathbf{do} \\
\quad xs \leftarrow readTVar\ b \\
\quad \mathbf{case}\ xs\ \mathbf{of} \\
\quad\quad [\,] \rightarrow retry \\
\quad\quad (x : xs') \rightarrow \mathbf{do} \\
\quad\quad\quad writeTVar\ b\ xs' \\
\quad\quad\quad return\ x
\end{array}
$$

Example: Buffer Revisited (3)

$$
\begin{aligned}
&\mathit{writeBuffer} :: \mathit{Buffer}\ a \to a \to \mathit{STM}\ ()\\
&\mathit{writeBuffer}\ (\mathit{Buffer}\ b)\ x = \mathbf{do}\\
&\quad \mathit{xs} \leftarrow \mathit{readTVar}\ b\\
&\quad \mathit{writeTVar}\ b\ (\mathit{xs} +\!\!+ [x])
\end{aligned}
$$

Example: Buffer Revisited (4)

The main program and code for readers and writers can remain unchanged, except that STM operations must be carried out ***atomically***:

$$
\begin{aligned}
& \mathit{main} = \mathbf{do} \\
& \quad b \leftarrow \mathit{atomically}\ \mathit{newBuffer} \\
& \quad \mathit{forkIO}\ (\mathit{writer}\ b) \\
& \quad \mathit{forkIO}\ (\mathit{writer}\ b) \\
& \quad \mathit{forkIO}\ (\mathit{reader}\ b) \\
& \quad \mathit{forkIO}\ (\mathit{reader}\ b) \\
& \quad \ldots
\end{aligned}
$$

Example: Buffer Revisited (5)

$$
\begin{array}{l}
reader :: Buffer\ Int \rightarrow IO\ () \\
reader\ n\ b = rLoop \\
\quad \mathbf{where} \\
\qquad rLoop = \mathbf{do} \\
\qquad\quad x \leftarrow atomically\ (readBuffer\ b) \\
\qquad\quad when\ (x > 0)\ \$\ \mathbf{do} \\
\qquad\qquad putStrLn\ (n +\!\!+ \texttt{": "} +\!\!+ show\ x) \\
\qquad\qquad rLoop
\end{array}
$$

Composition (1)

STM operations can be ***robustly composed***. That's the reason for making $readBuffer$ and $writeBuffer$ STM operations, and leaving it to client code to decide the scope of atomic blocks.

Example, sequential composition: reading two consecutive elements from a buffer b:

$$
\begin{array}{l}
atomically \; \$ \; \mathbf{do} \\
\quad x1 \leftarrow readBuffer \; b \\
\quad x2 \leftarrow readBuffer \; b \\
\quad \ldots
\end{array}
$$

Composition (2)

Example, composing alternatives: reading from one of two buffers $b1$ and $b2$:

$$x \leftarrow atomically \; \$$$
$$readBuffer \; b1$$
$$`orElse` \; readBuffer \; b2$$

The buffer operations thus composes nicely. No need to change the implementation of any of the operations!

Further STM Functionality (1)

$TMVar$: STM version of $MVars$ for synchoronisation; built on top of $TVar$s:

$$TMVar\ a \approx TVar\ (Maybe\ a)$$

Some operations:

- $newTMVar :: a \rightarrow STM\ (TMVar\ a)$
- $newEmptyTMVar :: STM\ (TMVar\ a)$
- $putTMVar :: TMVar\ a \rightarrow a \rightarrow STM\ ()$
- $takeTMVar :: TMVar\ a \rightarrow STM\ a$
- $readTMVar :: TMVar\ a \rightarrow STM\ a$
- $swapTMVar :: TMVar\ a \rightarrow a \rightarrow STM\ a$

Further STM Functionality (2)

Some non-blocking operations:

- $isEmptyTMVar :: TMVar\ a \to STM\ \ Bool$
- $tryPutTMVar :: TMVar\ a \to a \to STM\ \ Bool$
- $tryTakeTMVar :: TMVar\ a \to STM\ (Maybe\ a)$
- $tryReadTMVar :: TMVar\ a \to STM\ (Maybe\ a)$

Further STM Functionality (3)

Other process communication and synchronization facilities:

- $TChan\ a$: Unbounded FIFO channel
- $TQueue\ a$: Variation of $TChan$ with faster (amortised) throughput.
- $TBQueue\ a$: Bounded FIFO channel
- $TSem$: Transactional counting semaphore

Reading

- Koen Claessen. A Poor Man's Concurrency Monad. *Journal of Functional Programming*, 9(3), 1999.
- Wouter Swierstra and Thorsten Altenkirch. Beauty in the Beast: A Functional Semantics for the Awkward Squad. In *Proceedings of Haskell'07*, 2007.
- Tim Harris, Simon Marlow, Simon Peyton Jones, Maurice Herlihy. Composable Memory Transactions. In *Proceedings of PPoPP'05*, 2005
- Simon Peyton Jones. Beautiful Concurrency. Chapter from *Beautiful Code*, ed. Greg Wilson, O'Reilly 2007.

COMP4075/G54RFP: Lecture 10

Monad Transformers

Henrik Nilsson

University of Nottingham, UK

Monad Transformers (1)

What if we need to support more than one type of effect?

Monad Transformers (1)

What if we need to support more than one type of effect?

For example: State and Error/Partiality?

Monad Transformers (1)

What if we need to support more than one type of effect?

For example: State and Error/Partiality?

We could implement a suitable monad from scratch:

$$\mathbf{newtype}\ SE\ s\ a = SE\ (s \to Maybe\ (a, s))$$

Monad Transformers (2)

However:

Monad Transformers (2)

However:

- Not always obvious how: e.g., should the combination of state and error have been

$$\mathbf{newtype}\ SE\ s\ a = SE\ (s \to (Maybe\ a, s))$$

Monad Transformers (2)

However:

- Not always obvious how: e.g., should the combination of state and error have been

$$\mathbf{newtype}\ SE\ s\ a = SE\ (s \to (Maybe\ a, s))$$

- Duplication of effort: similar patterns related to specific effects are going to be repeated over and over in the various combinations.

Monad Transformers (3)

Monad Transformers can help:

Monad Transformers (3)

Monad Transformers can help:

- A ***monad transformer*** transforms a monad by adding support for an additional effect.

Monad Transformers (3)

Monad Transformers can help:

- A ***monad transformer*** transforms a monad by adding support for an additional effect.
- Monad transformer libraries can be developed, each transformer each adding a specific effect (state, error, . . .).

Monad Transformers (3)

Monad Transformers can help:

- A ***monad transformer*** transforms a monad by adding support for an additional effect.
- Monad transformer libraries can be developed, each transformer each adding a specific effect (state, error, . . .).
- A form of ***aspect-oriented programming***.

Monad Transformers (3)

Monad Transformers can help:

- A ***monad transformer*** transforms a monad by adding support for an additional effect.
- Monad transformer libraries can be developed, each transformer each adding a specific effect (state, error, . . .).
- A form of ***aspect-oriented programming***.
- MTL is one example of such a library.

Monad Transformers (3)

Monad Transformers can help:

- A ***monad transformer*** transforms a monad by adding support for an additional effect.
- Monad transformer libraries can be developed, each transformer each adding a specific effect (state, error, . . .).
- A form of ***aspect-oriented programming***.
- MTL is one example of such a library.

Will consider the general idea of monad transformers first; specific libraries discussed later.

Monad Transformers in Haskell (1)

- A ***monad transformer*** maps monads to monads. Represented by a type constructor T of the following kind:

$$T :: (* \to *) \to (* \to *)$$

Monad Transformers in Haskell (1)

- A ***monad transformer*** maps monads to monads. Represented by a type constructor T of the following kind:

$$T :: (* \to *) \to (* \to *)$$

- Additionally, a monad transformer ***adds*** computational effects.

Monad Transformers in Haskell (1)

- A ***monad transformer*** maps monads to monads. Represented by a type constructor T of the following kind:

$$T :: (* \to *) \to (* \to *)$$

- Additionally, a monad transformer ***adds*** computational effects.
- A mapping lift maps a computation in the underlying monad to one in the transformed monad:

$$\mathit{lift} :: M\ \ a \to T\ \ M\ \ a$$

Monad Transformers in Haskell (2)

- These requirements are captured by the following (multi-parameter) type class:

$$
\begin{aligned}
&\mathbf{class}\ (\mathit{Monad}\ m, \mathit{Monad}\ (t\ m)) \\
&\qquad \Rightarrow \mathit{MonadTransformer}\ t\ m\ \mathbf{where} \\
&\quad \mathit{lift} :: m\ a \to t\ m\ a
\end{aligned}
$$

Classes for Specific Effects

A monad transformer adds specific effects to ***any*** monad. Thus the effect-specific operations needs to be overloaded. For example:

$$
\begin{array}{l}
\textbf{class}\ Monad\ m \Rightarrow E\ m\ \textbf{where} \\
\quad eFail \quad\;\; :: m\ a \\
\quad eHandle :: m\ a \to m\ a \to m\ a \\
\textbf{class}\ Monad\ m \Rightarrow S\ m\ s \mid m \to s\ \textbf{where} \\
\quad sSet\ :: s \to m\ () \\
\quad sGet :: m\ s
\end{array}
$$

The Identity Monad

We are going to construct monads by successive transformations of the identity monad:

$$\mathbf{newtype}\ I\ a = I\ a$$

$$unI\ (I\ a) = a$$

$$\mathbf{instance}\ Monad\ I\ \mathbf{where}$$

$$\quad return\ a = I\ a$$

$$\quad m \gg\!= f\ = f\ (unI\ m)$$

$$runI :: I\ a \to a$$

$$runI = unI$$

The Error Monad Transformer (1)

$$\textbf{newtype}\ ET\ m\ a = ET\ (m\ (Maybe\ a))$$

$$unET\ (ET\ m) = m$$

The Error Monad Transformer (2)

Any monad transformed by ET is a monad:

$$
\begin{array}{l}
\textbf{instance}\ Monad\ m \Rightarrow Monad\ (ET\ m)\ \textbf{where} \\
\quad return\ a = ET\ (return\ (Just\ a)) \\
\quad m \gg\!= f = ET\ \$\ \textbf{do} \\
\quad\quad ma \leftarrow unET\ m \\
\quad\quad \textbf{case}\ ma\ \textbf{of} \\
\quad\quad\quad Nothing \rightarrow return\ Nothing \\
\quad\quad\quad Just\ a \quad\ \rightarrow unET\ (f\ a)
\end{array}
$$

The Error Monad Transformer (3)

We need the ability to run transformed monads:

$$
\begin{array}{l}
runET :: Monad\ m \Rightarrow ET\ m\ a \rightarrow m\ a \\
runET\ etm = \mathbf{do} \\
\quad ma \leftarrow unET\ etm \\
\quad \mathbf{case}\ ma\ \mathbf{of} \\
\qquad Just\ a \quad\ \rightarrow return\ a \\
\qquad Nothing \rightarrow error\ \texttt{"Should not happen"}
\end{array}
$$

The Error Monad Transformer (3)

We need the ability to run transformed monads:

$$
\begin{aligned}
&runET :: Monad\ m \Rightarrow ET\ m\ a \rightarrow m\ a\\
&runET\ etm = \mathbf{do}\\
&\quad ma \leftarrow unET\ etm\\
&\quad \mathbf{case}\ ma\ \mathbf{of}\\
&\qquad Just\ a \quad\ \rightarrow return\ a\\
&\qquad Nothing \rightarrow error\ \texttt{"Should not happen"}
\end{aligned}
$$

(Note: To simplify use, we discarded information about the effect, but as a result, we get a partial function. Returning $Maybe\ a$ better in general.)

The Error Monad Transformer (4)

ET is a monad transformer:

$$\begin{array}{l} \textbf{instance } \mathit{Monad}\ m \Rightarrow \\ \qquad\qquad\qquad \mathit{MonadTransformer}\ \mathit{ET}\ m\ \textbf{where} \\ \quad \mathit{lift}\ m = \mathit{ET}\ (m \gg\!= \lambda a \rightarrow \mathit{return}\ (\mathit{Just}\ a)) \end{array}$$

The Error Monad Transformer (5)

Any monad transformed by ET is an instance of E:

$$
\begin{array}{l}
\textbf{instance}\ Monad\ m \Rightarrow E\ (ET\ m)\ \textbf{where} \\
\quad eFail = ET\ (return\ Nothing) \\
\quad m1\ `eHandle`\ m2 = ET\ \$\ \textbf{do} \\
\qquad ma \leftarrow unET\ m1 \\
\qquad \textbf{case}\ ma\ \textbf{of} \\
\qquad\quad Nothing \rightarrow unET\ m2 \\
\qquad\quad Just\ \_ \quad \rightarrow return\ ma
\end{array}
$$

The Error Monad Transformer (6)

A state monad transformed by ET is a state monad:

$$
\begin{aligned}
&\textbf{instance } S\ m\ s \Rightarrow S\ (ET\ m)\ s \textbf{ where}\\
&\quad sSet\ s = \mathit{lift}\ (sSet\ s)\\
&\quad sGet\ \ = \mathit{lift}\ sGet
\end{aligned}
$$

Exercise 1: Running Transf. Monads

Let

```
ex2 = eFail `eHandle` return 1
```

1. Suggest a possible type for $ex2$. (Assume $1 :: Int$.)
2. Given your type, use the appropriate combination of “run functions” to run $ex2$.

Exercise 1: Solution

$$ex2 :: ET\ I\ Int$$
$$ex2 = eFail\ `eHandle`\ return\ 1$$

$$ex2result :: Int$$
$$ex2result = runI\ (runET\ ex2)$$

The State Monad Transformer (1)

$$\mathbf{newtype}\ ST\ s\ m\ a = ST\ (s \to m\ (a, s))$$
$$unST\ (ST\ m) = m$$

Any monad transformed by ST is a monad:

$$\begin{array}{l}
\mathbf{instance}\ Monad\ m \Rightarrow Monad\ (ST\ s\ m)\ \mathbf{where} \\
\quad return\ a = ST\ (\lambda s \to return\ (a, s)) \\
\quad m \gg\!= f = ST\ \$\ \lambda s \to \mathbf{do} \\
\qquad (a, s') \leftarrow unST\ m\ s \\
\qquad unST\ (f\ a)\ s'
\end{array}$$

The State Monad Transformer (2)

We need the ability to run transformed monads:

$$
\begin{aligned}
&\mathit{runST} :: \mathit{Monad}\ m \Rightarrow \mathit{ST}\ s\ m\ a \rightarrow s \rightarrow m\ a\\
&\mathit{runST}\ \mathit{stf}\ \mathit{s0} = \mathbf{do}\\
&\quad (a, \_) \leftarrow \mathit{unST}\ \mathit{stf}\ \mathit{s0}\\
&\quad \mathit{return}\ a
\end{aligned}
$$

The State Monad Transformer (2)

We need the ability to run transformed monads:

$$
\begin{aligned}
&\mathit{runST} :: \mathit{Monad}\ m \Rightarrow \mathit{ST}\ s\ m\ a \to s \to m\ a\\
&\mathit{runST}\ \mathit{stf}\ \mathit{s0} = \mathbf{do}\\
&\quad (a, \_) \leftarrow \mathit{unST}\ \mathit{stf}\ \mathit{s0}\\
&\quad \mathit{return}\ a
\end{aligned}
$$

(We are again discarding information to keep things simple. Returning the final state along with result would be more general.)

The State Monad Transformer (3)

ST is a monad transformer:

$$\mathbf{instance}\ \mathit{Monad}\ m \Rightarrow$$
$$\mathit{MonadTransformer}\ (ST\ s)\ m\ \mathbf{where}$$
$$\mathit{lift}\ m = ST\ (\lambda s \rightarrow m \ggg \lambda a \rightarrow \mathit{return}\ (a, s))$$

The State Monad Transformer (3)

Any monad transformed by ST is an instance of S:

$$\begin{array}{l} \textbf{instance } Monad\ m \Rightarrow S\ (ST\ s\ m)\ s \textbf{ where} \\ \quad sSet\ s = ST\ (\backslash\_ \rightarrow return\ ((), s)) \\ \quad sGet = ST\ (\lambda s \rightarrow return\ (s, s)) \end{array}$$

The State Monad Transformer (4)

An error monad transformed by ST is an error monad:

$$
\begin{aligned}
&\textbf{instance } E\ m \Rightarrow E\ (ST\ s\ m) \textbf{ where}\\
&\quad eFail = \mathit{lift}\ eFail\\
&\quad m1\ `eHandle`\ m2 = ST\ \$\ \lambda s \rightarrow\\
&\qquad unST\ m1\ s\ `eHandle`\ unST\ m2\ s
\end{aligned}
$$

Exercise 2: Effect Ordering

Consider the code fragment

$$ex3a :: (ST\ Int\ (ET\ I))\ Int$$
$$ex3a = (sSet\ 42 \gg eFail)\ `eHandle`\ sGet$$

Note that the exact same code fragment also can be typed as follows:

$$ex3b :: (ET\ (ST\ Int\ I))\ Int$$
$$ex3b = (sSet\ 42 \gg eFail)\ `eHandle`\ sGet$$

What is

$$runI\ (runET\ (runST\ ex3a\ 0))$$
$$runI\ (runST\ (runET\ ex3b)\ 0)$$

Exercise 2: Solution

$$
\begin{aligned}
runI\ (runET\ (runST\ ex3a\ 0)) &= 0 \\
runI\ (runST\ (runET\ ex3b)\ 0) &= 42
\end{aligned}
$$

Why? Because:

$$
\begin{aligned}
ST\ s\ (ET\ I)\ a &\cong s \to (ET\ I)\ (a, s) \\
&\cong s \to I\ (Maybe\ (a, s)) \\
&\cong s \to Maybe\ (a, s) \\
ET\ (ST\ s\ I)\ a &\cong (ST\ s\ I)\ (Maybe\ a) \\
&\cong s \to I\ (Maybe\ a, s) \\
&\cong s \to (Maybe\ a, s)
\end{aligned}
$$

MTL: Monad Transformer Library

Provides a number of standard monads, associated transformers, and all possible liftings in the style we have seen; e.g.:

- State ($Control.Monad.State$, lazy and strict)
- Exceptions ($Control.Monad.Except$)
- Lists ($Control.Monad.List$)
- Reader ($Control.Monad.Reader$)
- Writer ($Control.Monad.Writer$)
- Contiunations ($Control.Monad.Cont$)

MTL: State

$$
\begin{array}{l}
\mathbf{class}\ Monad\ m \Rightarrow MonadState\ s\ m \mid m \rightarrow s\ \mathbf{where} \\
\quad get\ \ \ :: m\ s \\
\quad put\ \ \ :: s \rightarrow m\ () \\
\quad state :: (s \rightarrow (a, s)) \rightarrow m\ a
\end{array}
$$

Transformer: $\mathbf{newtype}\ StateT\ s\ (m :: * \rightarrow *)\ a$

Run functions:

$$
\begin{array}{l}
runState :: State\ s\ a \rightarrow s \rightarrow (a, s) \\
evalState :: State\ s\ a \rightarrow s \rightarrow a \\
execState :: State\ s\ a \rightarrow s \rightarrow s
\end{array}
$$

MTL: Exception

$$\begin{array}{l}\textbf{class } Monad\ m \Rightarrow \\ \qquad\qquad MonadError\ e\ m \mid m \to e \textbf{ where} \\ \quad throwError :: e \to m\ a \\ \quad catchError\ :: m\ a \to (e \to m\ a) \to m\ a\end{array}$$

Transformer: $\textbf{newtype}\ ExceptT\ e\ (m :: * \to *)\ a$

Run function:

$$runExcept :: Except\ e\ a \to Either\ e\ a$$

MTL: Reader

$$\begin{aligned}
&\textbf{class}\ \mathit{Monad}\ m \Rightarrow \\
&\qquad \mathit{MonadReader}\ r\ m \mid m \to r\ \textbf{where} \\
&\quad \mathit{ask} \quad :: m\ r \\
&\quad \mathit{local} \quad :: (r \to r) \to m\ a \to m\ a \\
&\quad \mathit{reader} :: (r \to a) \to m\ a
\end{aligned}$$

Transformer: $\mathit{ReaderT}$

Run function:

$$\mathit{runReader} :: \mathit{Reader}\ r\ a \to r \to a$$

MTL: Writer

$$\textbf{class } (\mathit{Monoid}\ w, \mathit{Monad}\ m) \Rightarrow$$
$$\qquad \mathit{MonadWriter}\ w\ m \mid m \rightarrow w \textbf{ where}$$
$$\mathit{writer} :: (a, w) \rightarrow m\ a$$
$$\mathit{tell} \quad :: w \rightarrow m\ ()$$
$$\mathit{listen} :: m\ a \rightarrow m\ (a, w)$$
$$\mathit{pass} \quad :: m\ (a, w \rightarrow w) \rightarrow m\ a$$

Transformer: $\textbf{newtype}\ \mathit{WriterT}\ w\ (m :: * \rightarrow *)\ a$

Run function:

$$\mathit{runWriter} :: \mathit{Writer}\ w\ a \rightarrow (a, w)$$

Problems with Monad Transformers

- With one transformer for each possible effect we get a quadratic number of combinations; each has to be instantiated explicitly.
- Jaskelioff (2008,2009) has proposed a possible, more extensible alternative:
  - Traditional approach: unsystematic lifting on case-by-case basis.
  - Jaskelioff: systematic lifting based on theoretical principles where each operation is paired with a type of its implementation allowing implementations to be transformed generically.

Reading (1)

- Nick Benton, John Hughes, Eugenio Moggi. Monads and Effects. In *International Summer School on Applied Semantics 2000*, Caminha, Portugal, 2000.
- Sheng Liang, Paul Hudak, Mark Jones. Monad Transformers and Modular Interpreters. In *Proceedings of the 22nd ACM Symposium on Principles of Programming Languages (POPL'95)*, January 1995, San Francisco, California

Reading (2)

- Mauro Jaskelioff. Monatron: An Extensible Monad Transformer Library. In *Implementation of Functional Languages (IFL'08)*, 2008.
- Mauro Jaskelioff. Modular Monad Transformers. In *European Symposium on Programming (ESOP,09)*, 2009.

COMP4075/G54RFP: Lecture 11 & 12

The Threepenny GUI Toolkit

Henrik Nilsson

University of Nottingham, UK

What is Threepenny (1)

- Threepenny is a GUI framework written in Haskell that uses the web browser as a display.

What is Threepenny (1)

- Threepenny is a GUI framework written in Haskell that uses the web browser as a display.
- A program written with Threepenny is a small web server that:

What is Threepenny (1)

- Threepenny is a GUI framework written in Haskell that uses the web browser as a display.
- A program written with Threepenny is a small web server that:
  - displays the UI as a web page

What is Threepenny (1)

- Threepenny is a GUI framework written in Haskell that uses the web browser as a display.
- A program written with Threepenny is a small web server that:
  - displays the UI as a web page
  - allows the HTML ***Document Object Model*** (DOM) to be manipulated

What is Threepenny (1)

- Threepenny is a GUI framework written in Haskell that uses the web browser as a display.
- A program written with Threepenny is a small web server that:
  - displays the UI as a web page
  - allows the HTML ***Document Object Model*** (DOM) to be manipulated
  - handles JavaScript events in Haskell

What is Threepenny (1)

- Threepenny is a GUI framework written in Haskell that uses the web browser as a display.
- A program written with Threepenny is a small web server that:
  - displays the UI as a web page
  - allows the HTML ***Document Object Model*** (DOM) to be manipulated
  - handles JavaScript events in Haskell
- Works by sending JavaScript code to the client.

What is Threepenny (2)

- Frequent communication between browser and server: Threepenny is best used running on localhost or over the local network.

What is Threepenny (2)

- Frequent communication between browser and server: Threepenny is best used running on localhost or over the local network.
- Written by Heinrich Apfelmus.

Rich API

- Full set of widgets (buttons, menus, etc.)
- Drag and Drop
- HTML elements
- Support for CSS
- Canvas for general drawing
- Functional Reactive Programming (FRP)

Conceptual Model

- Build and manipulate a Document Object Model (DOM): a tree-structured element hierarchy representing the document displayed by the browser.

Conceptual Model

- Build and manipulate a Document Object Model (DOM): a tree-structured element hierarchy representing the document displayed by the browser.
- Set up event handlers to act on events from the elements.

Conceptual Model

- Build and manipulate a Document Object Model (DOM): a tree-structured element hierarchy representing the document displayed by the browser.
- Set up event handlers to act on events from the elements.
- Knowing a bit of HTML helps.

The UI Monad

Most work take place in the the ***User Interface*** monad UI:

The UI Monad

Most work take place in the the ***User Interface*** monad UI:

- Wrapper around IO; keeps track of e.g. window context.

The *UI* Monad

Most work take place in the the ***User Interface*** monad UI:

- Wrapper around IO; keeps track of e.g. window context.
- Instance of MonadIO, meaning that any IO operation can be lifted into UI:

$$\mathit{liftIO} :: \mathit{IO}\ a \rightarrow \mathit{UI}\ a$$

The Browser $Window$

- Type $Window$ represents a browser window.

The Browser $Window$

- Type $Window$ represents a browser window.
- It has an attribute $title$ that may be written:

$$title :: WriteAttr\ Window\ String$$

The Browser $Window$

- Type $Window$ represents a browser window.
- It has an attribute $title$ that may be written:

$$title :: WriteAttr\ Window\ String$$

- Retrieving the current window context:

$$askWindow :: UI\ Window$$

The Browser $Window$

- Type $Window$ represents a browser window.
- It has an attribute $title$ that may be written:

$$title :: WriteAttr\ Window\ String$$

- Retrieving the current window context:

$$askWindow :: UI\ Window$$

- Window passed to GUI code when server started:

$$startGUI :: Config \to (Window \to UI\ ()) \to IO\ ()$$

Elements

DOM made up of elements:

$$mkElement :: String \to UI\ Element$$

An element ***created*** when action run. Argument is an HTML elemen name: `"div"`, `"h1"`, `"p"`, etc.

Elements

DOM made up of elements:

$$mkElement :: String \rightarrow UI\ Element$$

An element ***created*** when action run. Argument is an HTML elemen name: `"div"`, `"h1"`, `"p"`, etc.

Standard elements predefined:

$$\begin{aligned} div &:: UI\ Element \\ h1 &:: UI\ Element \\ br &:: UI\ Element \\ button &:: UI\ Element \end{aligned}$$

Attributes (1)

Elements and other entities like windows have attributes that can be read and written:

$$\textbf{type}\ Attr\ x\ a = ReadWriteAttr\ x\ a\ a$$
$$\textbf{type}\ WriteAttr\ x\ i = ReadWriteAttr\ x\ i\ ()$$
$$\textbf{type}\ ReadAttr\ x\ o = ReadWriteAttr\ x\ ()\ o$$

$$set :: ReadWriteAttr\ x\ i\ o \to i \to UI\ x \to UI\ x$$

$$get :: ReadWriteAttr\ x\ i\ o \to x \to UI\ o$$

$ReadWriteAttr$, $WriteAttr$ etc. are records of functions for attribute reading and/or writing.

set and get work for any type of entity.

Attributes (2)

Sample attributes:

$$title :: WriteAttr\ Window\ String$$

$$color :: WriteAttr\ Element\ String$$

$$children :: WriteAttr\ Element\ [Element]$$

$$value :: Attr\ Element\ String$$

$$(\#+) :: UI\ Element \to [UI\ Element] \to UI\ Element$$

$$(\#.) :: UI\ Element \to String \to UI\ Element$$

$(\#+)$ appends children to a DOM element.
$(\#.)$ sets the CSS class.

Attributes (3)

Example usage ($(\#)$ is reverse function application):

$$
\begin{array}{l}
\mathit{mkElement}\ \texttt{"div"} \\
\quad \#\ \mathit{set}\ \mathit{style} \qquad [(\texttt{"color"}, \texttt{"\#CCAABB"})] \\
\quad \#\ \mathit{set}\ \mathit{draggable}\ \mathit{True} \\
\quad \#\ \mathit{set}\ \mathit{children}\ \mathit{otherElements}
\end{array}
$$

Events (1)

- The type $Event\ a$ represents streams of time-stamped events carrying values of type a.

Events (1)

- The type $Event\ a$ represents streams of time-stamped events carrying values of type a.
- Semantically: $Event\ a \approx [(Time, a)]$

Events (1)

- The type $Event\ a$ represents streams of time-stamped events carrying values of type a.
- Semantically: $Event\ a \approx [(Time, a)]$
- $Event$ is an instance of $Functor$.

Events (1)

- The type $Event\ a$ represents streams of time-stamped events carrying values of type a.
- Semantically: $Event\ a \approx [(Time, a)]$
- $Event$ is an instance of $Functor$.
- $Event$ is ***not*** an instance of $Applicative$. The type for $<\!\!*\!\!>$ would be

$$Event\ (a \to b) \to Event\ a \to Event\ b$$

However, this makes no sense as event streams in general are not synchronised.

Events (2)

Most events originate from UI elements; e.g.:

- $valueChange :: Element \to Event\ String$
- $click :: Element \to Event\ ()$
- $mousemove :: Element \to Event\ (Int, Int)$ (coordinates relative to the element)
- $hover :: Element \to Event\ ()$
- $focus :: Element \to Event\ ()$
- $keypress :: Element \to Event\ Char$

Events (3)

One or more handlers can be registered for events:

$$register :: Event\ a \rightarrow Handler\ a \rightarrow IO\ (IO\ ())$$

The resulting action is intended for deregistering a handler; future functionality.

Events (4)

Usually, registration is done using convenience functions designed for use directly with elements and in the UI monad:

$$on :: (element \to Event\ a)$$
$$\to element \to (a \to UI\ void) \to UI\ ()$$

For example:

$$\mathbf{do}$$
$$\ldots$$
$$on\ click\ element\,\$\,\lambda\_ \to \ldots$$
$$\ldots$$

Behaviors (1)

- The type $Behavior\ a$ represents continuously time-varying values of type a.

Behaviors (1)

- The type $Behavior\ a$ represents continuously time-varying values of type a.
- Semantically: $Behavior\ a \approx Time \rightarrow a$

Behaviors (1)

- The type $Behavior\ a$ represents continuously time-varying values of type a.
- Semantically: $Behavior\ a \approx Time \to a$
- $Behavior$ is an instance of $Functor$ ***and*** $Applicative$.

Behaviors (1)

- The type $Behavior\ a$ represents continuously time-varying values of type a.
- Semantically: $Behavior\ a \approx Time \rightarrow a$
- $Behavior$ is an instance of $Functor$ ***and*** $Applicative$.
- Recall that events are not an applicative. However, the following provides similar functionality:

$$
\begin{aligned}
(\texttt{<@>}) :: Behavior\ (a \rightarrow b) \\
\rightarrow Event\ a \rightarrow Event\ b
\end{aligned}
$$

Behaviors (2)

- Attributes can be set to time-varying values:

$$
\begin{aligned}
&\mathit{sink} :: \mathit{ReadWriteAttr}\ x\ i\ o \\
&\qquad \to \mathit{Behavior}\ i \to \mathit{UI}\ x \to \mathit{UI}\ x
\end{aligned}
$$

Behaviors (2)

- Attributes can be set to time-varying values:

$$sink :: ReadWriteAttr\ x\ i\ o \\ \rightarrow Behavior\ i \rightarrow UI\ x \rightarrow UI\ x$$

- There is also:

$$onChanges :: Behavior\ a \\ \rightarrow (a \rightarrow UI\ void) \rightarrow UI\ ()$$

But conceptually questionable as a behavior in general is ***always*** changing.

FRP (1)

Threepenny offers support for Functional Reactive Programming (FRP): transforming and composing behaviours and events as “whole values”.

FRP (1)

Threepenny offers support for Functional Reactive Programming (FRP): transforming and composing behaviours and events as “whole values”.

For example:

- $\mathit{filterJust} :: \mathit{Event}\ (\mathit{Maybe}\ a) \rightarrow \mathit{Event}\ a$
- $\mathit{unionWith} :: (a \rightarrow a \rightarrow a)$
  $\qquad\qquad \rightarrow \mathit{Event}\ a \rightarrow \mathit{Event}\ a \rightarrow \mathit{Event}\ a$
- $\mathit{unions} :: [\mathit{Event}\ a] \rightarrow \mathit{Event}\ [a]$
- $\mathit{split} :: \mathit{Event}\ (\mathit{Either}\ a\ b) \rightarrow (\mathit{Event}\ a, \mathit{Event}\ b)$

FRP (2)

- $accumE :: MonadIO\ m$
  $\Rightarrow a \rightarrow Event\ (a \rightarrow a) \rightarrow m\ (Event\ a)$
- $accumB :: MonadIO\ m$
  $\Rightarrow a \rightarrow Event\ (a \rightarrow a) \rightarrow m\ (Behavior\ a)$
- $stepper :: MonadIO\ m$
  $\Rightarrow a \rightarrow Event\ a \rightarrow m\ (Behavior\ a)$
- $(\lt@\gt) :: Behavior\ (a \rightarrow b)$
  $\rightarrow Event\ a \rightarrow Event\ b$

FRP (2)

- $accumE :: MonadIO\ m$
  $\Rightarrow a \rightarrow Event\ (a \rightarrow a) \rightarrow m\ (Event\ a)$
- $accumB :: MonadIO\ m$
  $\Rightarrow a \rightarrow Event\ (a \rightarrow a) \rightarrow m\ (Behavior\ a)$
- $stepper :: MonadIO\ m$
  $\Rightarrow a \rightarrow Event\ a \rightarrow m\ (Behavior\ a)$
- $(\lessdot\!@\!\gtrdot) :: Behavior\ (a \rightarrow b)$
  $\rightarrow Event\ a \rightarrow Event\ b$

Note: Stateful events and behaviors are returned as monadic computations.

Hello World (1)

A simple “Hello World” example:

Hello World (1)

A simple “Hello World” example:

- Display a button

Hello World (1)

A simple “Hello World” example:

- Display a button
- Change its text when clicked

Hello World (1)

A simple “Hello World” example:

- Display a button
- Change its text when clicked

First import the module. Large API, so partly qualified import recommended:

$$\textbf{module}\ \mathit{Main}\ \textbf{where}$$

$$\textbf{import}\ \mathit{qualified}\ \mathit{Graphics.UI.Threepenny}\ \mathit{as}\ \mathit{UI}$$
$$\textbf{import}\ \mathit{Graphics.UI.Threepenny.Core}$$

Hello World (2)

The $startGUI$ function starts a server:

$$startGUI :: Config \to (Window \to UI\ ()) \to IO\ ()$$

- $Config$-records carry configuartion parameters.
- $Window$ represents a browser window.
- The function $Window \to UI\ ()$ is called whenever a browser connects to the server and builds the initial HTML page.

Hello World (3)

Start a server listening on port 8023;
static content served from `../wwwroot`:

$$
\begin{array}{l}
main :: IO\ () \\
main = \mathbf{do} \\
\quad startGUI \\
\qquad defaultConfig \\
\qquad\quad \{jsPort \quad = Just\ 8023, \\
\qquad\qquad jsStatic = Just\ \texttt{"../wwwroot"}\} \\
\qquad setup
\end{array}
$$

Hello World (4)

Start by setting the window title:

$$setup :: Window \to UI\ ()$$
$$setup\ window = \mathbf{do}$$
$$\quad return\ window\ \#\ set\ UI.title\ \texttt{"Hello World!"}$$

Reversed function application: $(\#) :: a \to (a \to b) \to b$

set has type:

$$set :: ReadWriteAttr\ x\ i\ o \to i \to UI\ x \to UI\ x$$

The window reference is a pure value, passed in, hence the need to lift it into a UI computation using $return$.

Hello World (5)

Then create a button element:

$$button \leftarrow UI.button \; \# \; set \; UI.text \; \texttt{"Click me!"}$$

Hello World (5)

Then create a button element:

$$button \leftarrow UI.button \# set\ UI.text\ \texttt{"Click me!"}$$

Note that $UI.button$ has type:

$$button :: UI\ \ Element$$

A new button is is ***created*** whenever that action is run.

Hello World (5)

Then create a button element:

$$button \leftarrow UI.button \;\#\; set \;\; UI.text \; \texttt{"Click me!"}$$

Note that $UI.button$ has type:

$$button :: UI \;\; Element$$

A new button is is ***created*** whenever that action is run.

DOM elements can be accessed much like in JavaScript: searched, updated, moved, inspected.

Hello World (6)

To display the button, it must be attached to the DOM:

$$getBody\ window\ \#+\ [element\ button]$$

Hello World (6)

To display the button, it must be attached to the DOM:

$$getBody\ window\ \#+\ [element\ button]$$

The combinator $(\#+)$ appends DOM elements as children to a given element:

$$\begin{aligned}(\#+) &:: UI\ Element \to [UI\ Element] \\ &\quad \to UI\ Element\end{aligned}$$

Hello World (6)

To display the button, it must be attached to the DOM:

$$getBody\ window\ \#+\ [\,element\ \ button\,]$$

The combinator $(\#+)$ appends DOM elements as children to a given element:

$$(\#+) :: UI\ \ Element \to [\,UI\ \ Element\,] \to UI\ \ Element$$

$getBody$ gets the body DOM element:

$$getBody :: Window \to UI\ \ Element$$

Here, $element$ is just $return$.

Hello World (7)

Finally, register an event handler for the click event to change the text of the button:

$$
\begin{array}{l}
on\ UI.click\ button\ \$\ const\ \$\ \mathbf{do} \\
\quad element\ button \\
\qquad \#\ set\ UI.text\ \texttt{"I have been clicked!"}
\end{array}
$$

Types:

$$
\begin{array}{l}
on :: (element \to Event\ a) \to element \\
\qquad \to (a \to UI\ void) \to UI\ () \\
UI.click :: Element \to Event\ ()
\end{array}
$$

Buttons (1)

$$
\begin{array}{l}
mkButton :: String \rightarrow UI\ (Element, Element) \\
mkButton\ title = \mathbf{do} \\
\quad button \leftarrow UI.button\ \#.\ \texttt{"button"}\ \#+\ [string\ title] \\
\quad view \leftarrow UI.p\ \#+\ [element\ button] \\
\quad return\ (button, view) \\
mkButtons :: UI\ [Element] \\
mkButtons = \mathbf{do} \\
\quad list \leftarrow UI.ul\ \#.\ \texttt{"buttons-list"} \\
\quad \ldots
\end{array}
$$

Buttons (2)

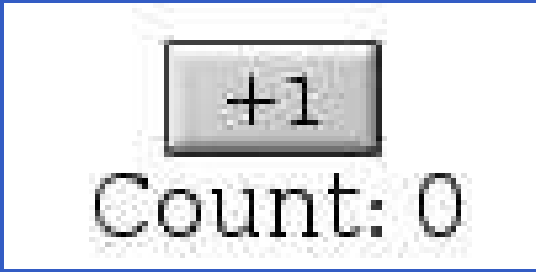
```haskell
(button1, view1) ← mkButton button1Title
on UI.hover button1 $ \_ → do
  element button1 # set text (button1Title ++ " [hover]")
on UI.leave button1 $ \_ → do
  element button1 # set text button1Title
on UI.click button1 $ \_ → do
  element button1 # set text (button1Title ++ " [pressed]"
  liftIO $ threadDelay $ 1000 ∗ 1000 ∗ 1
  element list
    #+ [UI.li # set html "<b>Delayed</b> result!"]
```

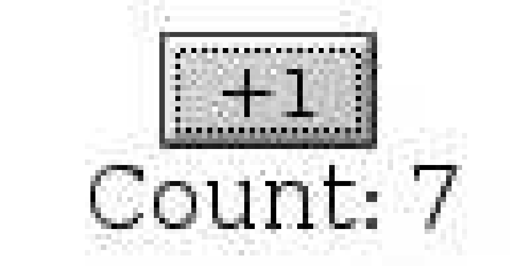
Buttons (3)

```
(button2, view2) ← mkButton button2Title
on UI.hover button2 $ \_ → do
   element button2 # set text (button2Title ++ " [hover]")
on UI.leave button2 $ \_ → do
   element button2 # set text button2Title
on UI.click button2 $ \_ → do
   element button2 # set text (button2Title ++ " [pressed]"
   element list
      #+ [UI.li # set html "Zap! Quick result!"]
return [list, view1, view2]
```

Counter Example 1 (1)

Simple counter, basic imperative style.





Idea:

- Keep the count in an imperative variable
- The click event handler increments the counter and updates the display accordingly.

Counter Example 1 (2)

$$
\begin{array}{l}
\mathit{setup} :: \mathit{Window} \rightarrow \mathit{UI}\ () \\
\mathit{setup}\ \mathit{window} = \mathbf{do} \\
\quad \mathit{return}\ \mathit{window} \\
\qquad \#\ \mathit{set}\ \mathit{UI}.\mathit{title}\ \texttt{"Counter Example 1"} \\
\quad \mathbf{let}\ \mathit{initCount} = 0 \\
\quad \mathit{counter} \leftarrow \mathit{liftIO}\ \$\ \mathit{newIORef}\ \mathit{initCount} \\
\quad \mathit{button}\ \ \leftarrow \mathit{UI}.\mathit{button}\ \#\ \mathit{set}\ \mathit{UI}.\mathit{text}\ \texttt{"+1"} \\
\quad \mathit{label}\ \quad \leftarrow \mathit{UI}.\mathit{label}\ \#\ \mathit{set}\ \mathit{UI}.\mathit{text} \\
\qquad\qquad\qquad\qquad\qquad\qquad (\texttt{"Count: "} +\!\!+ \\
\qquad\qquad\qquad\qquad\qquad\qquad\ \mathit{show}\ \mathit{initCount})
\end{array}
$$

Counter Example 1 (3)

$$
\begin{aligned}
&\mathit{getBody\ window} \mathbin{\#+} [\,\mathit{UI.center} \\
&\qquad\qquad\qquad\qquad \mathbin{\#+} [\,\mathit{element\ button}, \\
&\qquad\qquad\qquad\qquad\qquad \mathit{UI.br}, \\
&\qquad\qquad\qquad\qquad\qquad \mathit{element\ label}]] \\
&\mathit{on\ UI.click\ button} \mathbin{\$} \mathit{const} \mathbin{\$} \mathbf{do} \\
&\quad \mathit{count} \leftarrow \mathit{liftIO} \mathbin{\$} \mathbf{do} \\
&\qquad \mathit{modifyIORef\ counter}\ (+1) \\
&\qquad \mathit{readIORef\ counter} \\
&\quad \mathit{element\ label} \mathbin{\#} \mathit{set\ UI.text}\ (\texttt{"Count: "} \mathbin{+\!\!+} \\
&\qquad\qquad\qquad\qquad\qquad \mathit{show\ count})
\end{aligned}
$$

Counter Example 2 (1)

Counter with reset, “object-oriented” style.

Idea:

- Make a counter object with encapsulated state and two operations: reset and increment.
- Make a display object with a method for displaying a value.

Counter Example 2 (2)

Make a counter object:

$$
\begin{array}{l}
\mathit{mkCounter} :: \mathit{Int} \to \mathit{UI}\ (\mathit{UI}\ \mathit{Int}, \mathit{UI}\ \mathit{Int}) \\
\mathit{mkCounter}\ \mathit{initCount} = \mathbf{do} \\
\quad \mathit{counter} \leftarrow \mathit{liftIO}\ \$\ \mathit{newIORef}\ \mathit{initCount} \\
\quad \mathbf{let}\ \mathit{reset} = \mathit{liftIO}\ \$\ \mathit{writeIORef}\ \mathit{counter}\ \mathit{initCount} \\
\qquad\qquad\qquad\qquad\qquad \gg \mathit{return}\ \mathit{initCount} \\
\qquad\quad \mathit{incr} \ \ = \mathit{liftIO}\ \$\ \mathit{modifyIORef}\ \mathit{counter}\ (+1) \\
\qquad\qquad\qquad\qquad\qquad \gg \mathit{readIORef}\ \mathit{counter} \\
\quad \mathit{return}\ (\mathit{reset}, \mathit{incr})
\end{array}
$$

Counter Example 2 (3)

Make a display object:

$$
\begin{aligned}
& mkDisplay :: Int \rightarrow UI\ (Element, Int \rightarrow UI\ ())\\
& mkDisplay\ initCount = \mathbf{do}\\
& \quad \mathbf{let}\ showCount\ count =\\
& \qquad\qquad \texttt{"Count: "} +\!\!+\ show\ count\\
& \quad display \leftarrow UI.label\ \#\ set\ UI.text\\
& \qquad\qquad\qquad\qquad\qquad (showCount\ initCount)\\
& \quad \mathbf{let}\ dispCount\ count =\\
& \qquad\qquad ()\ <\$ element\ display\\
& \qquad\qquad\qquad\qquad \#\ set\ UI.text\ (showCount\ count)\\
& \quad return\ (display, dispCount)
\end{aligned}
$$

Counter Example 2 (4)

$$
\begin{array}{l}
setup :: Window \rightarrow UI\ () \\
setup\ window = \mathbf{do} \\
\quad return\ window \\
\qquad \#\ set\ UI.title\ \texttt{"Counter Example 2"} \\
\quad \mathbf{let}\ initCount = 0 \\
\quad (reset, incr) \leftarrow mkCounter\ initCount \\
\quad (display, dispCount) \leftarrow mkDisplay\ initCount \\
\quad buttonRst \leftarrow UI.button\ \#\ set\ UI.text\ \texttt{"RST"} \\
\quad buttonInc \leftarrow UI.button\ \#\ set\ UI.text\ \texttt{"+1"}
\end{array}
$$

Counter Example 2 (5)

$$
\begin{array}{l}
getBody\ window \\
\quad \mathbin{\#+} [\, UI.center \mathbin{\#+} [\, element\ buttonRst, \\
\qquad\qquad\qquad\qquad\qquad\quad element\ buttonInc, \\
\qquad\qquad\qquad\qquad\qquad\quad UI.br, \\
\qquad\qquad\qquad\qquad\qquad\quad element\ display \,]\,] \\
on\ UI.click\ buttonRst \mathbin{\$} const \mathbin{\$} reset \ggg dispCount \\
on\ UI.click\ buttonInc \mathbin{\$} const \mathbin{\$} incr \ggg dispCount
\end{array}
$$

Counter Example 3 (1)

Counter with reset, FRP style.

Idea:

- Accumulate the button clicks into a ***time-varying*** count; i.e., a $Behavior\ Int$.
- Make the text attribute of the display a time-varying text directly derived from the count; i.e., a $Behavior\ String$.

Counter Example 3 (2)

$$
\begin{array}{l}
setup :: Window \to UI\ () \\
setup\ window = \mathbf{do} \\
\quad return\ window \\
\qquad \#\ set\ UI.title\ \texttt{"Counter Example 3"} \\
\quad \mathbf{let}\ initCount = 0 \\
\quad buttonRst \leftarrow UI.button\ \#\ set\ UI.text\ \texttt{"RST"} \\
\quad buttonInc \leftarrow UI.button\ \#\ set\ UI.text\ \texttt{"+1"} \\
\quad \mathbf{let}\ reset\ \ = (const\ 0) <\$ UI.click\ buttonRst \\
\quad \mathbf{let}\ incr\ \ \ = (+1) \qquad\quad <\$ UI.click\ buttonInc
\end{array}
$$

Note: *Event* and *Behavior* are instances of *Functor*.

Counter Example 3 (3)

$$
\begin{aligned}
\mathit{count} &\leftarrow \mathit{accumB}\ 0\ \$\ \mathit{unionWith}\ \mathit{const}\ \mathit{reset}\ \mathit{incr} \\
\mathit{display} &\leftarrow \mathit{UI.label} \\
&\qquad\qquad \#\ \mathit{sink}\ \mathit{UI.text} \\
&\qquad\qquad\qquad\qquad (\mathit{fmap}\ \mathit{showCount}\ \mathit{count})
\end{aligned}
$$

Type signatures:

$$
\begin{aligned}
&\mathit{accumB} :: \mathit{MonadIO}\ m \Rightarrow \\
&\qquad\qquad a \rightarrow \mathit{Event}\ (a \rightarrow a) \rightarrow m\ (\mathit{Behavior}\ a) \\
&\mathit{unionWith} :: (a \rightarrow a \rightarrow a) \\
&\qquad\qquad\quad \rightarrow \mathit{Event}\ a \rightarrow \mathit{Event}\ a \rightarrow \mathit{Event}\ a \\
&\mathit{sink} :: \mathit{ReadWriteAttr}\ x\ i\ o \\
&\qquad\quad \rightarrow \mathit{Behavior}\ i \rightarrow \mathit{UI}\ x \rightarrow \mathit{UI}\ x
\end{aligned}
$$

Counter Example 3 (4)

$$
\begin{aligned}
& getBody\ window \\
& \quad \mathrel{\#\!+} [UI.center \mathrel{\#\!+} [element\ buttonRst, \\
& \qquad\qquad\qquad\qquad\qquad element\ buttonInc, \\
& \qquad\qquad\qquad\qquad\qquad UI.br, \\
& \qquad\qquad\qquad\qquad\qquad element\ display]]
\end{aligned}
$$

Counter Example 3 (4)

$$
\begin{aligned}
&\mathit{getBody}\ \mathit{window}\\
&\quad \#\!+ [\mathit{UI.center}\ \#\!+ [\mathit{element}\ \mathit{buttonRst},\\
&\qquad\qquad\qquad\qquad\quad \mathit{element}\ \mathit{buttonInc},\\
&\qquad\qquad\qquad\qquad\quad \mathit{UI.br},\\
&\qquad\qquad\qquad\qquad\quad \mathit{element}\ \mathit{display}]]
\end{aligned}
$$

- No callbacks.

Counter Example 3 (4)

$$
\begin{aligned}
&getBody\ window \\
&\quad \#\!+ [UI.center \#\!+ [element\ buttonRst, \\
&\qquad\qquad\qquad\qquad\quad element\ buttonInc, \\
&\qquad\qquad\qquad\qquad\quad UI.br, \\
&\qquad\qquad\qquad\qquad\quad element\ display]]
\end{aligned}
$$

- No callbacks.
- Thus no “callback soup” or “callback hell”!

Counter Example 3 (4)

$$
\begin{aligned}
&getBody\ window \\
&\quad \mathbin{\#+} [\,UI.center \mathbin{\#+} [\,element\ buttonRst, \\
&\qquad\qquad\qquad\qquad\qquad element\ buttonInc, \\
&\qquad\qquad\qquad\qquad\qquad UI.br, \\
&\qquad\qquad\qquad\qquad\qquad element\ display\,]\,]
\end{aligned}
$$

- No callbacks.
- Thus no “callback soup” or “callback hell”!
- Fairly declarative description of system: ***Whole-value Programming***.

Counter Example 3 (4)

$$
\begin{aligned}
&\mathit{getBody}\ \mathit{window}\\
&\qquad \#{+}\ [\mathit{UI}.\mathit{center}\ \#{+}\ [\mathit{element}\ \mathit{buttonRst},\\
&\qquad\qquad\qquad\qquad\qquad\qquad \mathit{element}\ \mathit{buttonInc},\\
&\qquad\qquad\qquad\qquad\qquad\qquad \mathit{UI}.\mathit{br},\\
&\qquad\qquad\qquad\qquad\qquad\qquad \mathit{element}\ \mathit{display}]]
\end{aligned}
$$

- No callbacks.
- Thus no “callback soup” or “callback hell”!
- Fairly declarative description of system: ***Whole-value Programming***.
- This style of programming has had significant impact on programming practice well beyond FP.

Currency Converter (1)

```
return window # set title "Currency Converter"
dollar ← UI.input
euro ← UI.input
getBody window #+ [
    column [
        grid [[string "Dollar:", element dollar]
              ,[string "Euro:", element euro]]
    , string "Amounts update while typing."
    ]]
```

Currency Converter (2)

$$euroIn \leftarrow stepper\ \texttt{"0"}\ \$\ UI.valueChange\ euro$$

$$dollarIn \leftarrow stepper\ \texttt{"0"}\ \$\ UI.valueChange\ dollar$$

let

$$rate = 0.7 :: Double$$

$$withString\ f =$$

$$maybe\ \texttt{"-"}\ (printf\ \texttt{"\%.2f"}) \circ fmap\ f \circ readMay$$

$$dollarOut = withString\ (/rate) <\$> euroIn$$

$$euroOut = withString\ (*rate) <\$> dollarIn$$

$$element\ euro\ \#\ sink\ value\ euroOut$$

$$element\ dollar\ \#\ sink\ value\ dollarOut$$

Reading

- Overview, including references to tutorials and examples:
  `http://wiki.haskell.org/Threepenny-gui`
- API reference:
  `http://hackage.haskell.org/package/threepenny-gui`

COMP4075/G54RFP: Lecture 13 & 14

Functional Programming with Structured Graphs

Henrik Nilsson

University of Nottingham, UK

Overview (1)

While pure languages are excellent for expressing computation over tree-like structures . . .

Overview (1)

While pure languages are excellent for expressing computation over tree-like structures . . .

. . . you could even argue it's a match made in heaven . . .

Overview (2)

…pure languages and graphs is more of a shotgun wedding:

Overview (3)

Summary of approaches:

Overview (3)

Summary of approaches:

- Resort to an essentially imperative formulation; e.g.:
  - Monads for structure and possibly performance (King and Launchbury 1994)
  - Clever tricks exploiting lazy evaluation (Johnsson 1998)

Overview (3)

Summary of approaches:

- Resort to an essentially imperative formulation; e.g.:
  - Monads for structure and possibly performance (King and Launchbury 1994)
  - Clever tricks exploiting lazy evaluation (Johnsson 1998)
- Implicitly exploiting the implementation-level graph structure of lazy evaluation
  - Limited applicability and fragile (Hughes 1985)

Overview (4)

- Explicitly exploiting the lazy evaluation graph structure
    - Impure, fragile (Gill 2009)

Overview (4)

- Explicitly exploiting the lazy evaluation graph structure
  - Impure, fragile (Gill 2009)
- Inductive graphs
  - Elegant, but imperative features needed in library implementation to realise standard asymptotic time complexity (Erwig 2001)
  - Foundation of the package FGL (Functional Graph Lipbrary) which is current.

Overview (5)

Oliveira & Cook (2012) propose a novel approach for ***structured graphs***:

Overview (5)

Oliveira & Cook (2012) propose a novel approach for ***structured graphs***:

- Key idea: account for sharing and cycles using ***parametric higher-order abstract syntax*** (PHOAS) (Chlipala 2008)

Overview (5)

Oliveira & Cook (2012) propose a novel approach for ***structured graphs***:

- Key idea: account for sharing and cycles using ***parametric higher-order abstract syntax*** (PHOAS) (Chlipala 2008)
- Similar ideas have been explored in the past (e.g.: Fegaras & Sheard 1996; Ghani, Hamana, Uustalu, Vene 2006), but none is as flexible or easy to use.

Overview (6)

- Good fit for functional programming (e.g. Haskell, Agda):

Overview (6)

- Good fit for functional programming (e.g. Haskell, Agda):
  - graphs can be seen as extension of algebraic data types

Overview (6)

- Good fit for functional programming (e.g. Haskell, Agda):
  - graphs can be seen as extension of algebraic data types
  - amenable to conventional functional programming and reasoning techniques (e.g., folds, induction)

Overview (6)

- Good fit for functional programming (e.g. Haskell, Agda):
  - graphs can be seen as extension of algebraic data types
  - amenable to conventional functional programming and reasoning techniques (e.g., folds, induction)
  - relatively light-weight; does not assume too exotic language features (rank 2 types)

Structured Graphs?

So what ***is*** a structured graph, then?

Structured Graphs?

So what ***is*** a structured graph, then?

Oliveira & Cook:

> *Structured graphs can be viewed as an extension of conventional algebraic datatypes that allow explicit definition and manipulation of cycles or sharing by using recursive binders and variables to explicitly represent possible sharing points.*

Structured Graphs? Take 2

A structured graph is a directed graph where:

Structured Graphs? Take 2

A structured graph is a directed graph where:

- the nodes are grouped into a ***hierarchy of regions***;

Structured Graphs? Take 2

A structured graph is a directed graph where:

- the nodes are grouped into a ***hierarchy of regions***;
- one or more designated ***named nodes*** in a region are the only possible targets for ***back-edges*** and ***cross-edges*** from nodes ***within*** that region (and its sub-regions).

Structured Graphs? Take 2

A structured graph is a directed graph where:

- the nodes are grouped into a ***hierarchy of regions***;
- one or more designated ***named nodes*** in a region are the only possible targets for ***back-edges*** and ***cross-edges*** from nodes ***within*** that region (and its sub-regions).

Think of ***scope*** in programming language terms, which is where PHOAS enters the picture, leveraging the host language to enforce the above constraints and facilitate the manipulation of such graphs.

Higher-order Abstract Syntax (HOAS)

Conventional representation of λ-terms:

$$\begin{aligned}&\mathbf{data}\ Term = \\ &\quad\mid Var\ Id \\ &\quad\mid Lam\ Id\ Term \\ &\quad\mid App\ Term\ Term\end{aligned}$$

Higher-order Abstract Syntax (HOAS)

Conventional representation of λ-terms:

$$
\begin{aligned}
&\mathbf{data}\ \mathit{Term} = \\
&\qquad \mid\ \mathit{Var}\ \mathit{Id} \\
&\qquad \mid\ \mathit{Lam}\ \mathit{Id}\ \mathit{Term} \\
&\qquad \mid\ \mathit{App}\ \mathit{Term}\ \mathit{Term}
\end{aligned}
$$

HOAS representation of λ-terms:

$$
\begin{aligned}
&\mathbf{data}\ \mathit{Term} = \\
&\qquad\ \ \ \mathit{Lam}\ (\mathit{Term} \to \mathit{Term}) \\
&\qquad \mid\ \mathit{App}\ \mathit{Term}\ \mathit{Term}
\end{aligned}
$$

Parametric HOAS

HOAS representation of λ-terms:

$$
\begin{aligned}
&\textbf{data}\ \mathit{Term} = \\
&\qquad \mathit{Lam}\ (\mathit{Term} \to \mathit{Term}) \\
&\quad\ \mid \mathit{App}\ \mathit{Term}\ \mathit{Term}
\end{aligned}
$$

Parametric HOAS

HOAS representation of λ-terms:

$$
\begin{array}{l}
\textbf{data}\ \mathit{Term} = \\
\qquad \mathit{Lam}\ (\mathit{Term} \to \mathit{Term}) \\
\quad \mid \mathit{App}\ \mathit{Term}\ \mathit{Term}
\end{array}
$$

PHOAS representation of λ-terms:

$$
\begin{array}{l}
\textbf{data}\ \mathit{PTerm}\ a = \\
\qquad \mathit{Var}\ a \\
\quad \mid \mathit{Lam}\ (a \to \mathit{PTerm}\ a) \\
\quad \mid \mathit{App}\ (\mathit{PTerm}\ a)\ (\mathit{PTerm}\ a)
\end{array}
$$

$$
\textbf{newtype}\ \mathit{Term} = \downarrow \{\uparrow :: \forall a\ .\ \mathit{PTerm}\ a\}
$$

Advantages of PHOAS

- Well-scopedness guaranteed (parametricity)
- No explicit environments
- Easy to define operations; in particular, HOAS often necessitates a function ***reify***: the inverse of the operation being defined.

Recursive PHOAS Binders (1)

Recursive binders can easily be added and given a fixed-point semantics. E.g., evaluation of λ-terms:

$$\mathbf{data}\ \mathit{PTerm}\ a = \mathit{Mu}_1\ (a \to \mathit{PTerm}\ a) \mid \ldots$$

$$\mathit{peval} :: \mathit{PTerm}\ \mathit{Value} \to \mathit{Value}$$

$$\ldots$$

$$\mathit{peval}\ (\mathit{Mu}_1\ f) = \mathit{fix}\ (\mathit{peval} \circ f)$$

Recursive PHOAS Binders (1)

Recursive binders can easily be added and given a fixed-point semantics. E.g., evaluation of λ-terms:

$$\textbf{data}\ PTerm\ a = Mu_1\ (a \to PTerm\ a) \mid \ldots$$

$$peval :: PTerm\ Value \to Value$$

$$\ldots$$

$$peval\ (Mu_1\ f) = \mathit{fix}\ (peval \circ f)$$

(Intuition: When applied to a $Value$, f returns a $PTerm\ Value$ representing the the body of f with the $Value$ substituted for the function argument; evaluation of that term yields the $Value$ we applied f to in the first place; i.e. the ***fixed point***.)

Recursive PHOAS Binders (2)

Or a letrec-like construct:

$$\mathbf{data}\ PTerm\ a = Mu_2\ ([\,a\,] \to [\,PTerm\ a\,])\ |\ \ldots$$

$$peval :: PTerm\ Value \to Value$$

$$\ldots$$

$$peval\ (Mu_2\ f) = head\ \$\ fix\ (map\ peval \circ f)$$

Recursive PHOAS Binders (2)

Or a letrec-like construct:

$$\mathbf{data}\ PTerm\ a = Mu_2\ ([a] \to [PTerm\ a]) \mid \ldots$$

$$peval :: PTerm\ Value \to Value$$

$$\ldots$$

$$peval\ (Mu_2\ f) = head\ \$\ fix\ (map\ peval \circ f)$$

- Note that the guarantee of well-formedness has been (subjectively) weakened. E.g.:

$$Mu_2\ (\lambda xs \to \ldots\ Var\ (xs\ !!\ n) \ldots)$$

Recursive PHOAS Binders (2)

Or a letrec-like construct:

$$\mathbf{data}\ PTerm\ a = Mu_2\ ([a] \to [PTerm\ a]) \mid \ldots$$

$$peval :: PTerm\ Value \to Value$$

$$\ldots$$

$$peval\ (Mu_2\ f) = head\ \$\ fix\ (map\ peval \circ f)$$

- Note that the guarantee of well-formedness has been (subjectively) weakened. E.g.:

$$Mu_2\ (\lambda xs \to \ldots Var\ (xs\ !!\ n) \ldots)$$

- Length-indexed vectors could help.

Cyclic Streams

$$
\begin{aligned}
&\textbf{data}\ PStream\ a\ v = \\
&\qquad Var\ v \\
&\quad\ \mid Mu\ (v \rightarrow PStream\ a\ v) \\
&\quad\ \mid Cons\ a\ (PStream\ a\ v) \\
&\textbf{newtype}\ Stream\ a = \downarrow \{\uparrow :: \forall v\ .\ PStream\ a\ v\}
\end{aligned}
$$

Finitely representable cyclic streams if inductive interpretation chosen.

Cyclic Streams

$$
\begin{aligned}
&\textbf{data}\ \mathit{PStream}\ a\ v = \\
&\qquad \mathit{Var}\ v \\
&\quad\ \mid \mathit{Mu}\ (v \to \mathit{PStream}\ a\ v) \\
&\quad\ \mid \mathit{Cons}\ a\ (\mathit{PStream}\ a\ v) \\
&\textbf{newtype}\ \mathit{Stream}\ a = \downarrow \{ \uparrow :: \forall v\ .\ \mathit{PStream}\ a\ v \}
\end{aligned}
$$

Finitely representable cyclic streams if inductive interpretation chosen. Example:

$$s_1 = \downarrow\ (\mathit{Cons}\ 1\ (\mathit{Mu}\ (\lambda x \to (\mathit{Cons}\ 2\ (\mathit{Cons}\ 3\ (\mathit{Var}\ x))))))$$

represents the stream $s_1 = 1 : \boxed{2 : 3 : \bullet}$

Fold on Cyclic Streams (1)

$$
\begin{aligned}
& \mathit{sfold} :: (a \to b \to b) \to b \to \mathit{Stream}\ a \to b \\
& \mathit{sfold}\ f\ b\ s = \mathit{sfAux}\ (\uparrow\ s) \\
& \quad \textbf{where} \\
& \qquad \mathit{sfAux}\ (\mathit{Var}\ v) \qquad\quad = v \\
& \qquad \mathit{sfAux}\ (\mathit{Mu}\ g) \qquad\quad = \mathit{sfAux}\ (g\ b) \\
& \qquad \mathit{sfAux}\ (\mathit{Cons}\ x\ \mathit{xs}) = f\ x\ (\mathit{sfAux}\ \mathit{xs}) \\
& \mathit{selems} :: \mathit{Stream}\ a \to [a] \\
& \mathit{selems} = \mathit{sfold}\ (:)\ [\,]
\end{aligned}
$$

Fold on Cyclic Streams (1)

$$sfold :: (a \to b \to b) \to b \to Stream\ a \to b$$
$$sfold\ f\ b\ s = sfAux\ (\uparrow\ s)$$

where

$$\begin{aligned} sfAux\ (Var\ v) &= v \\ sfAux\ (Mu\ g) &= sfAux\ (g\ b) \\ sfAux\ (Cons\ x\ xs) &= f\ x\ (sfAux\ xs) \end{aligned}$$

$$selems :: Stream\ a \to [a]$$
$$selems = sfold\ (:)\ [\,]$$

Example:

$$selems\ s_1 \Rightarrow [1, 2, 3]$$

Fold on Cyclic Streams (2)

Another possibility, using g at type $() \to PStream\ a\ ()$:

$$
\begin{array}{l}
sfold :: (a \to b \to b) \to b \to Stream\ a \to b \\
sfold\ f\ b\ s = sfAux\ (\uparrow\ s) \\
\quad \textbf{where} \\
\qquad \begin{array}{lcl}
sfAux\ (Var\ \_) & = & b \\
sfAux\ (Mu\ g) & = & sfAux\ (g\ ()) \\
sfAux\ (Cons\ x\ xs) & = & f\ x\ (sfAux\ xs)
\end{array}
\end{array}
$$

Cyclic Fold on Cyclic Streams

$$
\begin{aligned}
&\mathit{scfold} :: (a \to b \to b) \to \mathit{Stream}\ a \to b \\
&\mathit{scfold}\ f\ s = \mathit{csfAux}\ (\uparrow\ s) \\
&\quad \textbf{where} \\
&\qquad \begin{aligned}
\mathit{csfAux}\ (\mathit{Var}\ v) &= v \\
\mathit{csfAux}\ (\mathit{Mu}\ g) &= \mathit{fix}\ (\mathit{csfAux} \circ g) \\
\mathit{csfAux}\ (\mathit{Cons}\ x\ xs) &= f\ x\ (\mathit{csfAux}\ xs)
\end{aligned} \\
&\mathit{toList} :: \mathit{Stream}\ a \to [a] \\
&\mathit{toList} = \mathit{scfold}\ (:)
\end{aligned}
$$

Cyclic Fold on Cyclic Streams

$$scfold :: (a \to b \to b) \to Stream\ a \to b$$
$$scfold\ f\ s = csfAux\ (\uparrow\ s)$$

where

$$\begin{aligned} csfAux\ (Var\ v) &= v \\ csfAux\ (Mu\ g) &= \mathit{fix}\ (csfAux \circ g) \\ csfAux\ (Cons\ x\ xs) &= f\ x\ (csfAux\ xs) \end{aligned}$$

$$toList :: Stream\ a \to [a]$$
$$toList = scfold\ (:)$$

Example: (Because $\mathit{fix}\ (\lambda x \to 2 : 3 : x) = [2, 3, 2, 3, \ldots)$

$$scfold\ s_1 \Rightarrow [1, 2, 3, 2, 3, 2, 3, \ldots$$

Sharing-preserving Transformation

$$smap :: (a \to b) \to Stream\ a \to Stream\ b$$
$$smap\ f\ s = \downarrow\ (smAux\ (\uparrow\ s))$$

where

$$\begin{aligned}
smAux\ (Var\ x) &= Var\ x \\
smAux\ (Mu\ g) &= Mu\ (smAux \circ g) \\
smAux\ (Cons\ x\ xs) &= Cons\ (f\ x)\ (smAux\ xs)
\end{aligned}$$

Sharing-preserving Transformation

$$smap :: (a \to b) \to Stream\ a \to Stream\ b$$
$$smap\ f\ s = \downarrow\ (smAux\ (\uparrow\ s))$$

where

$$\begin{aligned} smAux\ (Var\ x) &= Var\ x \\ smAux\ (Mu\ g) &= Mu\ (smAux \circ g) \\ smAux\ (Cons\ x\ xs) &= Cons\ (f\ x)\ (smAux\ xs) \end{aligned}$$

Note that standard map on a list that happens to be represented by a cyclic heap structure in a lazy functional language (like $ones = 1 : ones$) will lose the cyclic structure (unless memoization is used).

Tail of a Cyclic Stream (1)

$$
\begin{array}{l}
stail :: Stream\ a \to Stream\ a \\
stail\ s = \downarrow\ (pjoin\ (ptail\ (\uparrow\ s))) \\
\quad \textbf{where} \\
\qquad ptail\ (Cons\ x\ xs) = xs \\
\qquad ptail\ (Mu\ g) \qquad\quad = Mu\ (\lambda x \to \\
\qquad\quad \textbf{let}\ phead\ (Mu\ g) \qquad\quad = phead\ (g\ x) \\
\qquad\qquad\ \ phead\ (Cons\ y\ ys) = y \\
\qquad\quad \textbf{in} \\
\qquad\qquad ptail\ (g\ (Cons\ (phead\ (g\ x))\ x)))
\end{array}
$$

Here, g is used at type
$PStream\ a\ v \to PStream\ a\ (PStream\ a\ v)$

Tail of a Cyclic Stream (2)

$pjoin$ is a monadic-like $join$ operation.

$$\begin{aligned}
&pjoin :: PStream\ a\ (PStream\ a\ v) \to PStream\ a\ v \\
&pjoin\ (Var\ x) \qquad\ \ = x \\
&pjoin\ (Mu\ f) \qquad\ \ = Mu\ (pjoin \circ f \circ Var) \\
&pjoin\ (Cons\ x\ xs) = Cons\ x\ (pjoin\ xs)
\end{aligned}$$

Structural Equality

The nub of the algorithm:

$$
\begin{array}{llll}
\multicolumn{4}{l}{peq :: Eq\ a \Rightarrow Int \to PStream\ a\ Int \to PStream\ a\ Int \to Bool} \\
peq\ n\ (Var\ n_1) & (Var\ n_2) & = & n_1 \equiv n_2 \\
peq\ n\ (Mu\ f) & (Mu\ g) & = & peq\ (n+1)\ (f\ n)\ (g\ n) \\
peq\ n\ (Cons\ x\ xs) & (Cons\ y\ ys) & = & x \equiv y \wedge peq\ n\ xs\ ys \\
peq\ \_\ \_ & \_ & = & False
\end{array}
$$

Generic Structured Graphs

- Parametrize on a functor describing the node structure.
- Employ multi-binder to allow cross-edges in addition to back-edges.

$$
\begin{aligned}
&\textbf{data}\ Rec\ f\ v = \\
&\qquad Var\ v \\
&\quad\ \mid Mu\ ([v] \to [f\ (Rec\ f\ a)]) \\
&\quad\ \mid In\ (f\ (Rec\ f\ a)) \\
&\textbf{newtype}\ Graph\ f = \downarrow \{\uparrow :: \forall v\ .\ Rec\ f\ v\}
\end{aligned}
$$

Cyclic Trees in Terms of Graphs

$$\textbf{data}\ \mathit{TreeF}\ a\ r = \mathit{Empty} \mid \mathit{Fork}\ a\ r\ r$$
$$\quad \textbf{deriving}\ (\mathit{Functor}, \mathit{Foldable}, \mathit{Traversable})$$

$$\textbf{type}\ \mathit{Tree}\ a = \mathit{Graph}\ (\mathit{TreeF}\ a)$$

Example:

$$\begin{aligned}
&\mathit{tree} = \downarrow\ (\mathit{Mu}\ (\lambda(\sim(t_1 : t_2 : t_3 : _)) \rightarrow [\\
&\quad \mathit{Fork}\ 1\ (\mathit{In}\ (\mathit{Fork}\ 4\ (\mathit{Var}\ t_2)\ (\mathit{In}\ \mathit{Empty})))\\
&\qquad\qquad (\mathit{Var}\ t_3),\\
&\quad \mathit{Fork}\ 2\ (\mathit{Var}\ t_1)\ (\mathit{Var}\ t_3),\\
&\quad \mathit{Fork}\ 3\ (\mathit{Var}\ t_2)\ (\mathit{Var}\ t_1)]))
\end{aligned}$$

Some Generic Graph Folds

$$
\begin{aligned}
&\mathit{fold} :: \mathit{Functor}\ f\ (f\ \ a \to a) \to a \to \mathit{Graph}\ f \to a \\
&\mathit{cfold} :: \mathit{Functor}\ f\ (f\ \ a \to a) \to \mathit{Graph}\ f \to a \\
&\mathit{sfold} :: (\mathit{Eq}\ \ a, \mathit{Functor}\ f) \Rightarrow \\
&\qquad\qquad (f\ \ a \to a) \to a \to \mathit{Graph}\ f \to a
\end{aligned}
$$

sfold uses a fixed-point operator that iterates the function until convergence (assuming monotonicity).

An Application: Liveness (1)

A variable v is ***live*** at point p if there exists an execution path from p to a use of v along which v is not updated.

```
1   i := m;
2   n := 1;
3   while (i < 10) do begin
4       n := n * p;
5       i := i + 1
6   end
7   return n;
```

Which of `i`, `m`, `n`, `p` are live immediately before lines 1, 3, 7?

An Application: Liveness (2)

Define a suitable dataflow graph:

$$\mathbf{data}\ \mathit{Expr} = \mathit{Lit}\ \mathit{Int} \mid \mathit{Use}\ \mathit{Id} \mid \mathit{Add}\ \mathit{Expr}\ \mathit{Expr} \mid \ldots$$

$$\mathit{uses} :: \mathit{Expr} \to [\mathit{Id}]$$

$$\mathit{uses} = \ldots$$

$$\begin{aligned}
&\mathbf{data}\ \mathit{CodeF}\ a = \\
&\qquad \mathit{Return}\ \mathit{Id} \\
&\quad\ \mid \mathit{Assign}\ \mathit{Id}\ \mathit{Expr}\ a \\
&\quad\ \mid \mathit{IfZ}\ \mathit{Id}\ a\ a \\
&\quad \mathbf{deriving}\ (\mathit{Functor}, \mathit{Foldable}, \mathit{Traversable})
\end{aligned}$$

An Application: Liveness (3)

Define the analysis algebra:

$$
\begin{array}{l}
liveF :: CodeF\ [Id] \to [Id] \\
liveF\ (Return\ v) \qquad\ \ = [v] \\
liveF\ (Assign\ v\ e\ l) = uses\ e \cup (l \setminus [v]) \\
liveF\ (IfZ\ v\ l_1\ l_2) \quad\ = [v] \cup l_1 \cup l_2
\end{array}
$$

Finally, define the liveness analysis as an $sfold$:

$$
\begin{array}{l}
live :: Graph\ CodeF \to [Id] \\
live = sfold\ liveF\ [\,]
\end{array}
$$

(Returns what's live at whatever block is first.)

Conclusions

- Oliveira's and Cook's method works well for applications where the graph structure is preserved or where computation is by folding over a graph.
- Structure-changing operations is possible, but much more involved (see paper).

References (1)

- Adam Chlipala. Parametric higher-order abstract syntax for mechanized semantics. ICFP’08, 2008.
- Martin Erwig. Inductive graphs and functional graph algorithms. Journal of Functional Programming 11(5), 2001.
- Leonidas Fegaras and Tim Sheard. Revisiting catamorphisms over datatypes with embedded functions (or, programs from outer space). POPL’96, 1996.

References (2)

- Neil Ghani, Makoto Hamana, Tarmo Uustalu, & Varmo Vene. Representing cyclic structures as nested datatypes. TFP’06, 2006.
- Andy Gill. Type-safe observable sharing in Haskell. Haskell’09, 2009.
- John Hughes. Lazy memo-functions. FPCA’85, 1985.
- Thomas Johnsson. Efficient graph algorithms using lazy monolithic arrays. Journal of Functional Programming 8(4), 1998

References (3)

- David J. King & John Launchbury. Lazy depth-first search and linear graph algorithms in Haskell. Glasgow Workshop on Functional Programming, 1994.
- Bruno Oliveira & William Cook. Functional Programming with Structured Graphs. Draft, March 2012.

COMP4075/G54RFP: Lecture 15

Property-based Testing

Henrik Nilsson

University of Nottingham, UK

QuickCheck: What is it? (1)

- Framework for property-based testing
- Flexible language for stating properties
- Random test cases generated automatically based on type of argument(s) to properties.
- Highly configurable:
  - Number, size of test cases can easily be specified
  - Additional types for more fine-grained control of test case generation
  - Customised test case generators

QuickCheck: What is it? (2)

- Support for checking test coverage
- Counterexample produced when test case fails
- Counterexamples automatically shrunk in attempt to find minimal counterexample

Basic Example

$$\textbf{import}\ \mathit{Test.QuickCheck}$$

$$\mathit{prop\_RevRev} :: [\mathit{Int}] \to \mathit{Bool}$$
$$\mathit{prop\_RevRev}\ \mathit{xs} =$$
$$\quad \mathit{reverse}\ (\mathit{reverse}\ \mathit{xs}) \equiv \mathit{xs}$$

$$\mathit{prop\_RevApp} :: [\mathit{Int}] \to [\mathit{Int}] \to \mathit{Bool}$$
$$\mathit{prop\_RevApp}\ \mathit{xs}\ \mathit{ys} =$$
$$\quad \mathit{reverse}\ (\mathit{xs} +\!\!+\ \mathit{ys}) \equiv \mathit{reverse}\ \mathit{ys} +\!\!+\ \mathit{reverse}\ \mathit{xs}$$

$$\mathit{quickCheck}\ (\mathit{prop\_RevRev}\ .\&\&.\ \mathit{prop\_RevApp})$$

Basic Example

$$\textbf{import}\ Test.QuickCheck$$

$$prop\_RevRev :: [Int] \to Bool$$
$$prop\_RevRev\ xs =$$
$$\quad reverse\ (reverse\ xs) \equiv xs$$

$$prop\_RevApp :: [Int] \to [Int] \to Bool$$
$$prop\_RevApp\ xs\ ys =$$
$$\quad reverse\ (xs +\!\!+\ ys) \equiv reverse\ ys +\!\!+\ reverse\ xs$$

$$quickCheck\ (prop\_RevRev\ .\&\&.\ prop\_RevApp)$$

Result: `+++ OK, passed 100 tests`

Class *Testable*

Type of quickCheck:

$$quickCheck :: Testable\ prop \Rightarrow prop \rightarrow IO\ ()$$

Class $\mathit{Testable}$

Type of quickCheck:

$$quickCheck :: Testable\ prop \Rightarrow prop \rightarrow IO\ ()$$

$\mathit{Testable}$ and some instances:

$$
\begin{aligned}
&\textbf{class}\ Testable\ prop\ \textbf{where}\\
&\quad property\quad :: prop \rightarrow Property\\
&\quad exhaustive :: prop \rightarrow Bool\\
&\textbf{instance}\ Testable\ Bool\\
&\textbf{instance}\ Testable\ Property\\
&\textbf{instance}\ (Arbitrary\ a, Show\ a, Testable\ prop) \Rightarrow\\
&\qquad\qquad Testable\ (a \rightarrow prop)
\end{aligned}
$$

Class $Arbitrary$

$$
\begin{array}{l}
\textbf{class } Arbitrary\ a\ \textbf{where} \\
\quad arbitrary :: Gen\ a \\
\quad shrink :: a \to [a]
\end{array}
$$

$$generate :: Gen\ a \to IO\ a$$

$Arbitrary$ instance for all basic types provided. Easy to define additional ones.

Class *Arbitrary*

$$
\begin{array}{l}
\textbf{class } Arbitrary\ a\ \textbf{where} \\
\quad arbitrary :: Gen\ a \\
\quad shrink :: a \to [\,a\,]
\end{array}
$$

$$generate :: Gen\ a \to IO\ a$$

Arbitrary instance for all basic types provided. Easy to define additional ones.

Gen is a *Monad*, *Applicative*, *Functor* (and more).

Class *Arbitrary*

$$\textbf{class}\ Arbitrary\ a\ \textbf{where}$$
$$arbitrary :: Gen\ a$$
$$shrink :: a \rightarrow [a]$$

$$generate :: Gen\ a \rightarrow IO\ a$$

Arbitrary instance for all basic types provided. Easy to define additional ones.

Gen is a *Monad*, *Applicative*, *Functor* (and more).

Example:

$$generate\ (arbitrary :: Gen\ [Int])$$

Result: `[28,-2,-26,6,8,8,1]`

Stating Properties (1)

Implication:

$$(==>) :: Testable\ prop \Rightarrow Bool \to prop \to Property$$

Stating Properties (1)

Implication:

$$(==>) :: Testable\ prop \Rightarrow Bool \to prop \to Property$$

Universal quantification:

$$\begin{aligned} forAll :: (Show\ a, Testable\ prop) \Rightarrow \\ Gen\ a \to (a \to prop) \to Property \end{aligned}$$

Stating Properties (1)

Implication:

$$(==>) :: \mathit{Testable\ prop} \Rightarrow \mathit{Bool} \to \mathit{prop} \to \mathit{Property}$$

Universal quantification:

$$\begin{aligned} \mathit{forAll} :: (&\mathit{Show}\ a, \mathit{Testable\ prop}) \Rightarrow \\ &\mathit{Gen}\ a \to (a \to \mathit{prop}) \to \mathit{Property} \end{aligned}$$

Conjunction and disjunction:

$$\begin{aligned} (.\&\&.) :: &(\mathit{Testable\ prop1}, \mathit{Testable\ prop2}) \\ &\Rightarrow \mathit{prop1} \to \mathit{prop2} \to \mathit{Property} \\ (.||.) :: \quad &(\mathit{Testable\ prop1}, \mathit{Testable\ prop2}) \\ &\Rightarrow \mathit{prop1} \to \mathit{prop2} \to \mathit{Property} \end{aligned}$$

Stating Properties (2)

$$
\begin{array}{l}
prop\_Index :: Eq\ a \Rightarrow [\,a\,] \rightarrow Property \\
prop\_Index\ xs = \\
\quad length\ xs > 0 ==> \\
\qquad forAll\ (choose\ (0, length\ xs - 1))\ \$\ \lambda i \rightarrow \\
\qquad\quad xs\ !!\ i \equiv head\ (drop\ i\ xs)
\end{array}
$$

Modifiers (1)

A number of newtypes with $Arbitrary$ instances. E.g. $NonEmptyList\ a$, $SortedList\ a$, $NonNegative\ a$

Modifiers (1)

A number of newtypes with $Arbitrary$ instances. E.g. $NonEmptyList\ a$, $SortedList\ a$, $NonNegative\ a$

Typical definitions:

$$\begin{aligned} &\mathbf{newtype}\ NonEmptyList\ a = \\ &\qquad NonEmpty\ \{ getNonEmpty :: [a] \} \\ &\mathbf{newtype}\ NonNegative\ a = \\ &\qquad NonNegative\ \{ getNonNegative :: a \} \end{aligned}$$

Modifiers (2)

Example:

$$
\begin{aligned}
&prop\_Index :: \\
&\quad Eq\ a \Rightarrow NonEmptyList\ [a] \to Property \\
&prop\_Index\ (NonEmpty\ xs) = \\
&\quad forAll\ (choose\ (0, length\ xs - 1))\ \$\ \lambda i \to \\
&\qquad xs\ !!\ i \equiv head\ (drop\ i\ xs)
\end{aligned}
$$

Runnnig Tests

Basic function to run tests:

$$quickCheck :: Testable\ prop \Rightarrow prop \rightarrow IO\ ()$$

Runnnig Tests

Basic function to run tests:

$$quickCheck :: Testable\ prop \Rightarrow prop \rightarrow IO\ ()$$

Printing of all test cases:

$$verboseCheck :: Testable\ prop \Rightarrow prop \rightarrow IO\ ()$$

Runnnig Tests

Basic function to run tests:

$$quickCheck :: Testable\ prop \Rightarrow prop \rightarrow IO\ ()$$

Printing of all test cases:

$$verboseCheck :: Testable\ prop \Rightarrow prop \rightarrow IO\ ()$$

Controlling e.g. number and size of test cases:

$$quickCheckWith ::$$
$$\quad Testable\ prop \Rightarrow Args \rightarrow prop \rightarrow IO\ ()$$

$$quickCheckWith$$
$$\quad (stdArgs\ \{ maxSize = 10, maxSuccess = 1000 \})$$
$$\quad prop\_XXX$$

Labelling and Coverage (1)

$label$ attaches a label to a test case:

$$label :: Testable\ prop \Rightarrow String \to prop \to Property$$

Example:

$$
\begin{aligned}
&prop\_RevRev :: [Int] \to Property\\
&prop\_RevRev\ xs =\\
&\quad label\ (\texttt{"length is "} +\!\!+\ show\ (length\ xs))\ \$\\
&\qquad reverse\ (reverse\ xs) === xs
\end{aligned}
$$

Labelling and Coverage (2)

Result:

```
+++ OK, passed 100 tests:
7% length is 7
6% length is 3
5% length is 4
4% length is 6
```

There are also $cover$ and $checkCover$ for checking/enforcingig specific coverage requirements.

A Cautionary Tale (1)

$$
\begin{array}{l}
prop\_Sqrt :: Double \rightarrow Bool \\
prop\_Sqrt\ x \\
\quad \begin{array}{l|ll}
 & x < 0 & = isNaN\ sqrtX \\
 & x \equiv 0 \vee x \equiv 1 & = sqrtX \equiv x \\
 & x < 1 & = sqrtX > x \\
 & x > 1 & = sqrtX > 0 \wedge sqrtX < x
\end{array} \\
\quad \mathbf{where} \\
\qquad sqrtX = sqrt\ x
\end{array}
$$

$$main = quickCheck\ propSqrt$$

A Cautionary Tale (1)

$$
\begin{aligned}
&prop\_Sqrt :: Double \to Bool\\
&prop\_Sqrt\ x\\
&\quad \mid x < 0 \qquad\qquad\quad\ = isNaN\ \ sqrtX\\
&\quad \mid x \equiv 0 \lor x \equiv 1 = sqrtX \equiv x\\
&\quad \mid x < 1 \qquad\qquad\quad\ = sqrtX > x\\
&\quad \mid x > 1 \qquad\qquad\quad\ = sqrtX > 0 \land sqrtX < x\\
&\quad \mathbf{where}\\
&\qquad sqrtX = sqrt\ x
\end{aligned}
$$

$$main = quickCheck\ propSqrt$$

Result: `+++ OK, passed 100 tests`

A Cautionary Tale (2)

$$
\begin{aligned}
&prop\_Sqrt :: Double \to Bool \\
&prop\_Sqrt\ x \\
&\qquad \ldots \\
&\quad \textbf{where} \\
&\qquad sqrtX = flawedSqrt\ x \\
&\qquad flawedSqrt\ x \mid x \equiv 1 \qquad\quad = 0 \\
&\qquad\qquad\qquad\qquad \mid otherwise = sqrt\ x \\
&main = quickCheck\ propSqrt
\end{aligned}
$$

A Cautionary Tale (2)

$$
\begin{array}{l}
prop\_Sqrt :: Double \to Bool \\
prop\_Sqrt\ x \\
\quad \ldots \\
\quad \mathbf{where} \\
\quad\quad sqrtX = flawedSqrt\ x \\
\quad\quad flawedSqrt\ x \mid x \equiv 1 \quad\quad = 0 \\
\quad\quad\quad\quad\quad\quad\quad \mid otherwise = sqrt\ x \\
main = quickCheck\ propSqrt
\end{array}
$$

Result: `+++ OK, passed 100 tests`

A Cautionary Tale (2)

$$
\begin{aligned}
&prop\_Sqrt :: Double \to Bool\\
&prop\_Sqrt\ x\\
&\qquad \ldots\\
&\quad \mathbf{where}\\
&\qquad sqrtX = flawedSqrt\ x\\
&\qquad flawedSqrt\ x \mid x \equiv 1 \qquad = 0\\
&\qquad\qquad\qquad\quad \mid otherwise = sqrt\ x\\
&main = quickCheck\ propSqrt
\end{aligned}
$$

Result: `+++ OK, passed 100 tests`

Errr ...

A Cautionary Tale (3)

$$
\begin{array}{l}
\mathit{prop\_Sqrt} :: \mathit{Double} \rightarrow \mathit{Bool} \\
\mathit{prop\_Sqrt}\ x \\
\qquad \ldots \\
\quad \textbf{where} \\
\qquad \mathit{sqrtX} = \mathit{flawedSqrt}\ x \\
\qquad \ldots \\
\mathit{main} = \mathit{quickCheckWith} \\
\qquad\qquad (\mathit{stdArgs}\ \{\mathit{maxSuccess} = 1000000\}) \\
\qquad\qquad \mathit{propSqrt}
\end{array}
$$

A Cautionary Tale (3)

$$
\begin{array}{l}
prop\_Sqrt :: Double \to Bool \\
prop\_Sqrt\ x \\
\qquad \ldots \\
\quad \textbf{where} \\
\qquad sqrtX = flawedSqrt\ x \\
\qquad \ldots \\
main = quickCheckWith \\
\qquad\qquad (stdArgs\ \{\ maxSuccess = 1000000\ \}) \\
\qquad\qquad propSqrt
\end{array}
$$

Result: `+++ OK, passed 1000000 tests`

A Cautionary Tale (3)

$$
\begin{array}{l}
\mathit{prop\_Sqrt} :: \mathit{Double} \to \mathit{Bool} \\
\mathit{prop\_Sqrt}\ x \\
\qquad \ldots \\
\quad \mathbf{where} \\
\qquad \mathit{sqrtX} = \mathit{flawedSqrt}\ x \\
\qquad \ldots \\
\mathit{main} = \mathit{quickCheckWith} \\
\qquad\qquad (\mathit{stdArgs}\ \{\mathit{maxSuccess} = 1000000\}) \\
\qquad\qquad \mathit{propSqrt}
\end{array}
$$

Result: `+++ OK, passed 1000000 tests`

Oops.

A Cautionary Tale (4)

Simply test specific cases when needed:

$$
\begin{aligned}
&prop\_Sqrt0 :: Bool \\
&prop\_Sqrt0 = mySqrt\ 0 \equiv 0
\end{aligned}
$$

$$
\begin{aligned}
&prop\_Sqrt1 :: Bool \\
&prop\_Sqrt1 = mySqrt\ 1 \equiv 1
\end{aligned}
$$

A Cautionary Tale (5)

$$
\begin{array}{l}
prop\_SqrtX :: Double \to Bool \\
prop\_SqrtX\ x \\
\quad \mid x < 0 \ = isNaN\ \ sqrtX \\
\quad \mid x \leqslant 1 = sqrtX \geqslant x \\
\quad \mid x > 1 \ = sqrtX > 0 \wedge sqrtX < x \\
\quad \mathbf{where} \\
\qquad sqrtX = mySqrt\ x
\end{array}
$$

A Cautionary Tale (6)

```
prop_Sqrt :: Property
prop_Sqrt = counterexample
              "sqrt 0 failed"
              prop_Sqrt0
      .&&. counterexample
              "sqrt 1 failed"
              prop_Sqrt1
      .&&. prop_SqrtX
```

Testing Interval Arithmetic (1)

Lifting a unary operator $\ominus$ to an operator $\hat{\ominus}$ working on intervals is defined as follows, assuming $\ominus$ is defined on the entire interval:

$$\hat{\ominus} i = [\min_{\forall x \in i} \ominus x, \ \max_{\forall x \in i} \ominus x]$$

Testing Interval Arithmetic (1)

Lifting a unary operator $\ominus$ to an operator $\hat{\ominus}$ working on intervals is defined as follows, assuming $\ominus$ is defined on the entire interval:

$$\hat{\ominus} i = [\min_{\forall x \in i} \ominus x,\ \max_{\forall x \in i} \ominus x]$$

And for binary operators:

$$i_1 \hat{\otimes} i_2 = [\min_{\forall x \in i_1, y \in i_2} x \otimes y,\ \max_{\forall x \in i_1, y \in i_2} x \otimes y]$$

Testing Interval Arithmetic (2)

But how can we test that? In general, very difficult to find the global minimum/maximum of a function over an interval without further information e.g. about its derivatives.

Testing Interval Arithmetic (2)

But how can we test that? In general, very difficult to find the global minimum/maximum of a function over an interval without further information e.g. about its derivatives.

However, for a given interval i, it follows that:

$$\forall x \in i.\ominus x \in \hat{\ominus} i$$

Testing Interval Arithmetic (3)

Unfortunately, $\hat{\ominus} i = [-\infty, \ +\infty]$ satisfies

$$\forall x \in i . \ominus x \in \hat{\ominus} i$$

Testing Interval Arithmetic (3)

Unfortunately, $\hat{\ominus} i = [-\infty,\ +\infty]$ satisfies

$$\forall x \in i.\ \ominus\ x \in \hat{\ominus} i$$

We should ideally test that the result interval is not larger than necessary. But that is hard too.

Testing Interval Arithmetic (3)

Unfortunately, $\hat{\ominus} i = [-\infty, \ +\infty]$ satisfies

$$\forall x \in i . \ominus x \in \hat{\ominus} i$$

We should ideally test that the result interval is not larger than necessary. But that is hard too.

However, the definition does imply that a 1-point interval must be mapped to a 1-point interval:

$$\hat{\ominus}[x, x] = [\ominus x, \ \ominus x]$$

While not perfect, does rule out trivial implementations and it is easy to test.

Testing Interval Arithmetic (4)

For binary operators:

- For given intervals i_1 and i_2:

$$\forall x \in i_1, y \in i_2.\ x \otimes y \in i_1 \hat{\otimes} i_2$$

- For given x and y:

$$[x, x] \mathbin{\hat{\otimes}} [y, y] = [x \otimes y,\ x \otimes y]$$

Let us turn the above into QuickCheck test cases interactively.

COMP4075/G54RFP: Lecture 16
Optics

Henrik Nilsson

University of Nottingham, UK

Guest Tutorial: Preparations (1)

- Ben Clifford: Build a RESTful Room-Booking Server Using Servant and Aeson
  Fri. 6 Dec 2019, 11:00–13:00, CS A32

Guest Tutorial: Preparations (1)

- Ben Clifford: Build a RESTful Room-Booking Server Using Servant and Aeson
  Fri. 6 Dec 2019, 11:00–13:00, CS A32
- Goal: Building simple booking system accessible through a JSON+HTTP API using established Haskell libraries.

Guest Tutorial: Preparations (1)

- Ben Clifford: Build a RESTful Room-Booking Server Using Servant and Aeson
  Fri. 6 Dec 2019, 11:00–13:00, CS A32
- Goal: Building simple booking system accessible through a JSON+HTTP API using established Haskell libraries.
- Hands on tutorial! Preferably, bring laptop with:
  - Stack (cross-platform Haskell dev. system)
  - Tutorial prerequisites installed
  - WiFi connectivity

Guest Tutorial: Preparations (2)

- See link off guest lecture webpage for details: `https://github.com/benclifford/2019-nottingham-prereq`

Guest Tutorial: Preparations (2)

- See link off guest lecture webpage for details: `https://github.com/benclifford/2019-nottingham-prereq`
- To get most out of the tutorial, it is essential to:
  - Bring a laptop with prerequisites installed
  - Resolve issues ***before*** the tutorial

Optics: What?

- ***Optics*** are ***functional references***: focusing on one part of a structure for access an update.

Optics: What?

- ***Optics*** are ***functional references***: focusing on one part of a structure for access an update.
- Examples of “optics” include ***Lens***, ***Prism***. ***Iso***, ***Traversable***.

Optics: What?

- ***Optics*** are ***functional references***: focusing on one part of a structure for access an update.
- Examples of “optics” include ***Lens***, ***Prism***. ***Iso***, ***Traversable***.
- Different kinds of “optics” allow different number of focal points and may or may not be invertible.

Optics: What?

- ***Optics*** are ***functional references***: focusing on one part of a structure for access an update.
- Examples of "optics" include ***Lens***, ***Prism***. ***Iso***, ***Traversable***.
- Different kinds of "optics" allow different number of focal points and may or may not be invertible.
- Today, we'll look at lenses. Lenses ***compose*** very nicely, allowing focusing on the target step-by-step.

Motivating Example (1)

- Haskell’s “records” often get critisized.

Motivating Example (1)

- Haskell’s “records” often get critisized.
- Somewhat undeserved:
  - Merit of simplicity
  - Disciplined field naming conventions can mitigate some of the drawbacks

Motivating Example (1)

- Haskell's "records" often get critisized.
- Somewhat undeserved:
  - Merit of simplicity
  - Disciplined field naming conventions can mitigate some of the drawbacks
- Lenses go a long way to address other criticisms.

Motivating Example (2)

$$
\begin{aligned}
&\mathbf{data}\ \mathit{Point} = \mathit{Point}\ \{\\
&\quad \mathit{positionX} :: \mathit{Double},\\
&\quad \mathit{positionY} :: \mathit{Double}\\
&\}\\
&\mathbf{data}\ \mathit{Segment} = \mathit{Segment}\ \{\\
&\quad \mathit{segmentStart} :: \mathit{Point},\\
&\quad \mathit{segmentEnd}\ \ :: \mathit{Point}\\
&\}
\end{aligned}
$$

Motivating Example (3)

Field access is straightforward.
For example, given $seg :: Segment$:

$$end\_y = positionY \circ segmentEnd \;\$\; seg$$

Motivating Example (3)

Field access is straightforward.
For example, given $seg :: Segment$:

$$end\_y = positionY \circ segmentEnd \mathbin{\$} seg$$

Field update is much clunkier:

$$\begin{array}{l}
\mathbf{let}\ end = segmentEnd\ seg \\
\mathbf{in}\ seg\ \{\,segmentEnd = \\
\qquad\qquad end\ \{\,positionY = 2 * positionY\ \ end\,\} \\
\quad\ \}
\end{array}$$

Lenses to the rescue! (1)

Lenses for focusing on specific fields can be defined manually, but there is support for automating the process which is convenient if there are many fields.

Field names must then start by an underscore.

Lenses to the rescue! (2)

```haskell
import Control.Lens

data Point = Point {
    _positionX :: Double,
    _positionY :: Double
}
makeLenses ''Point

data Segment = Segment {
    _segmentStart :: Point,
    _segmentEnd   :: Point
}
makeLenses ''Segment
```

Lenses to the rescue! (3)

This gives us lenses for the fields:

$$
\begin{array}{ll}
\mathit{positionX} & :: \mathit{Lens'}\ \mathit{Point}\ \mathit{Double} \\
\mathit{positionY} & :: \mathit{Lens'}\ \mathit{Point}\ \mathit{Double} \\
\mathit{segmentStart} & :: \mathit{Lens'}\ \mathit{Segment}\ \mathit{Point} \\
\mathit{segmentEnd} & :: \mathit{Lens'}\ \mathit{Segment}\ \mathit{Point}
\end{array}
$$

Lenses to the rescue! (3)

This gives us lenses for the fields:

$$\begin{aligned}
&positionX &&:: Lens'\ Point\ Double\\
&positionY &&:: Lens'\ Point\ Double\\
&segmentStart &&:: Lens'\ Segment\ Point\\
&segmentEnd &&:: Lens'\ Segment\ Point
\end{aligned}$$

Individual fields can now be accessed and updated:

$$\begin{aligned}
&view\ segmentEnd\ seg\\
&set\ segmentEnd\ seg
\end{aligned}$$

Lenses to the rescue! (4)

But what is really cool is that lenses compose!

Ordinary function composition, but note the order: from "large" to "small":

$$view\ (segmentEnd \circ positionY)\ seg$$
$$over\ (segmentEnd \circ positionY)\ (2*)\ seg$$

How does this work? (1)

$Lens'\ a\ b$ is a type synonym:

$$\begin{aligned}&\mathbf{type}\ Lens'\ s\ a =\\&\quad Functor\ f \Rightarrow (a \rightarrow f\ a) \rightarrow (s \rightarrow f\ s)\end{aligned}$$

How does this work? (1)

$Lens'\ a\ b$ is a type synonym:

$$\textbf{type}\ Lens'\ s\ a = \\ \quad Functor\ f \Rightarrow (a \to f\ a) \to (s \to f\ s)$$

This is a function that transforms an operation on a part of type a of a structure of type s to an operation on the whole structure.

How does this work? (2)

In particular:

$$
\begin{aligned}
&\mathit{positionY} :: \\
&\quad \mathit{Functor}\ f \Rightarrow \\
&\qquad (\mathit{Double} \to f\ \mathit{Double}) \to (\mathit{Point} \to f\ \mathit{Point}) \\
&\mathit{segmentEnd} :: \\
&\quad \mathit{Functor}\ f \Rightarrow \\
&\qquad (\mathit{Point} \to f\ \mathit{Point}) \to (\mathit{Segment} \to f\ \mathit{Segment})
\end{aligned}
$$

And thus:

$$
\begin{aligned}
&\mathit{segmentEnd} \circ \mathit{positionY} :: \mathit{Functor}\ f \Rightarrow \\
&\quad (\mathit{Double} \to f\ \mathit{Double}) \to (\mathit{Segment} \to f\ \mathit{Segment})
\end{aligned}
$$

How does this work? (3)

Combinators like $view$, set, $over$ instantiate the functor to something suitable to achieve the desired effect:

$$
\begin{aligned}
&set \quad :: ASetter\ s\ t\ a\ b \to b \to s \to t \\
&over \ :: ASetter\ s\ t\ a\ b \to (a \to b) \to s \to t \\
&\mathbf{type}\ ASetter\ s\ t\ a\ b = \\
&\qquad\qquad (a \to Identity\ b) \to s \to Identity\ t
\end{aligned}
$$

Consequently, e.g.:

$$
\begin{aligned}
&over\ (segmentEnd \circ positionY) :: \\
&\quad (Double \to Double) \to Segment \to Segment
\end{aligned}
$$

How does this work? (4)

$$\mathit{view} :: \mathit{MonadReader}\ s\ m \Rightarrow \mathit{Getting}\ a\ s\ a \to m\ a$$
$$\mathbf{type}\ \mathit{Getting}\ r\ s\ a =$$
$$(a \to \mathit{Const}\ r\ a) \to s \to \mathit{Const}\ r\ s$$

Const is the constant functor:

$$\mathbf{newtype}\ \mathit{Const}\ a\ b = \mathit{Const}\ \{\mathit{getConst} :: a\}$$

Consequently, e.g.:

$$\mathit{view}\ (\mathit{segmentEnd} \circ \mathit{positionY}) ::$$
$$\mathit{MonadReader}\ \mathit{Segment}\ m \Rightarrow m\ \mathit{Double}$$

How does this work? (5)

As $(\rightarrow)$ $Segment$ is a reader monad,

$$
\begin{aligned}
&view\ (segmentEnd \circ positionY) :: \\
&\quad MonadReader\ Segment\ m \Rightarrow m\ Double
\end{aligned}
$$

is just a function $Segment \rightarrow Double$.

Some other useful lenses

The *Lens* package defines lots of optics for standard types. In particular, it defines lenses for all field of tuples up to size 19. For example:

$> view\ \_2\ (3, 4, 5)$

4

$> view\ \_2\ (5, 6, 7, 8)$

6

$> set\ \_3\ 9\ (5, 6, 7, 8)$

$(5, 6, 9, 8)$

$> over\ \_3\ (*2)\ (3, 4, 5)$

$(3, 4, 10)$

COMP4075/G54RFP: Lecture 17

Arrows, FRP, and Games

Henrik Nilsson

University of Nottingham, UK

Program Games Declaratively?

Video games is not a major application area for declarative programming . . .

Program Games Declaratively?

Video games is not a major application area for declarative programming . . . or even a niche one.

Perhaps not so surprising:

Program Games Declaratively?

Video games is not a major application area for declarative programming . . . or even a niche one.

Perhaps not so surprising:

- Many pragmatical reasons: performance, legacy issues, . . .

Program Games Declaratively?

Video games is not a major application area for declarative programming . . . or even a niche one.

Perhaps not so surprising:

- Many pragmatical reasons: performance, legacy issues, . . .
- State and effects are pervasive in video games: Is declarative programming even a conceptually good fit?

But Why NOT, Really?

Many eloquent and compelling cases for functional programming in general:

But Why NOT, Really?

Many eloquent and compelling cases for functional programming in general:

- John Backus, 1977 ACM Turing Award Lecture: Can Programming Be Liberated from the von Neumann Style?

But Why NOT, Really?

Many eloquent and compelling cases for functional programming in general:

- John Backus, 1977 ACM Turing Award Lecture: Can Programming Be Liberated from the von Neumann Style?
- John Hughes, recent retrospective: Why Functional Programming Matters (on YouTube, recommended)

But Why NOT, Really?

Many eloquent and compelling cases for functional programming in general:

- John Backus, 1977 ACM Turing Award Lecture: Can Programming Be Liberated from the von Neumann Style?
- John Hughes, recent retrospective: Why Functional Programming Matters (on YouTube, recommended)

One key point: Program with whole values, not a word-at-a-time. (Will come back to this.)

Possible Gains

High profile people in the games industry have pointed out potential benefits:

Possible Gains

High profile people in the games industry have pointed out potential benefits:

- John D. Carmack, id Software:
  Wolfenstein 3D, Doom, Quake

Possible Gains

High profile people in the games industry have pointed out potential benefits:

- John D. Carmack, id Software: Wolfenstein 3D, Doom, Quake
- Tim Sweeney, Epic Games: The Unreal Engine

Possible Gains

High profile people in the games industry have pointed out potential benefits:

- John D. Carmack, id Software:
  Wolfenstein 3D, Doom, Quake
- Tim Sweeney, Epic Games:
  The Unreal Engine

E.g. pure, declarative code:

- promotes parallelism
- eliminates many sources of errors

“Whole Values” for Games?

How should we go about writing video games “declaratively”?

“Whole Values” for Games?

How should we go about writing video games “declaratively”?

In particular, what should those “whole values” be?

“Whole Values” for Games?

How should we go about writing video games “declaratively”?

In particular, what should those “whole values” be?

- Could be conventional entities like vectors, arrays, lists and aggregates of such.

“Whole Values” for Games?

How should we go about writing video games “declaratively”?

In particular, what should those “whole values” be?

- Could be conventional entities like vectors, arrays, lists and aggregates of such.
- Could even be things like pictures.

“Whole Values” for Games?

How should we go about writing video games “declaratively”?

In particular, what should those “whole values” be?

- Could be conventional entities like vectors, arrays, lists and aggregates of such.
- Could even be things like pictures.

But we are going to go one step further and consider programming with ***time-varying entities***.

Take-home Message # 1

Take-home Message # 1

Video games can be programmed declaratively by describing ***what*** entities are ***over*** time.

Take-home Message # 1

Video games can be programmed declaratively by describing ***what*** entities are ***over*** time.

Our whole values are things like:

- The totality of input from the player
- The animated graphics output
- The entire life of a game object

Take-home Message # 1

Video games can be programmed declaratively by describing ***what*** entities are ***over*** time.

Our whole values are things like:

- The totality of input from the player
- The animated graphics output
- The entire life of a game object

We construct and work with ***pure*** functions on these:

- The game: function from input to animation
- In the game: fixed point of function on collection of game objects

Take-home Message # 1 (cont.)

- That said, we focus on the core game logic in the following: there will often be code around the “edges” (e.g., rendering, interfacing to input devices) that may not be very declarative, at least not in the sense above.
- See Perez & Nilsson (2015) for one approach.

Take-home Message # 2

You too can program games declaratively . . .

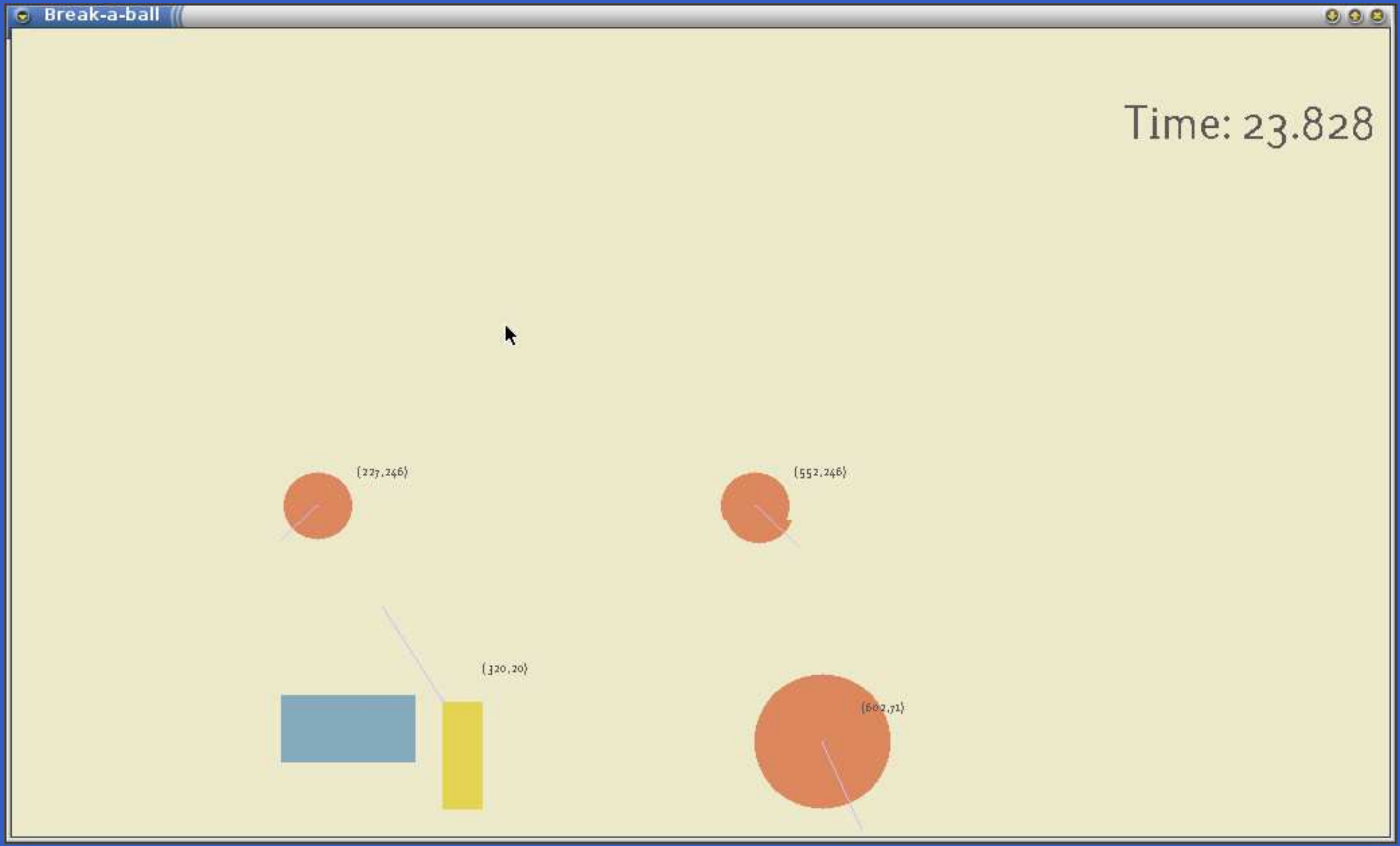
Take-home Message # 2

You too can program games declaratively . . . today!

Play Store: Keera Breakout (Keera Studios)

Take-home Game!

Or download one for free to your Android device!



Play Store: Pang-a-lambda (Keera Studios)

This Tutorial

We will implement a Breakout-like game using:

- Functional Reactive Programming (FRP): a paradigm for describing time-varying entities
- Simple DirectMedia Layer (SDL) for rendering etc.

Focus on FRP as that is what we need for the game logic. We will use Yampa:

`http://hackage.haskell.org/package/Yampa-0.9.6`

Functional Reactive Programming

Functional Reactive Programming

- Key idea: Don’t program one-time-step-at-a-time, but describe an evolving entity as whole.

Functional Reactive Programming

- Key idea: Don't program one-time-step-at-a-time, but describe an evolving entity as whole.
- FRP originated in Conal Elliott and Paul Hudak's work on Functional Reactive Animation (Fran).

Functional Reactive Programming

- Key idea: Don’t program one-time-step-at-a-time, but describe an evolving entity as whole.
- FRP originated in Conal Elliott and Paul Hudak’s work on Functional Reactive Animation (Fran). Highly cited 1997 ICFP paper; ICFP award for most influential paper in 2007.

Functional Reactive Programming

- Key idea: Don't program one-time-step-at-a-time, but describe an evolving entity as whole.
- FRP originated in Conal Elliott and Paul Hudak's work on Functional Reactive Animation (Fran). Highly cited 1997 ICFP paper; ICFP award for most influential paper in 2007.
- FRP has evolved in a number of directions and into different concrete implementations.

Functional Reactive Programming

- Key idea: Don’t program one-time-step-at-a-time, but describe an evolving entity as whole.
- FRP originated in Conal Elliott and Paul Hudak’s work on Functional Reactive Animation (Fran). Highly cited 1997 ICFP paper; ICFP award for most influential paper in 2007.
- FRP has evolved in a number of directions and into different concrete implementations.
- We will use Yampa: an arrows-based FRP system embedded in Haskell.

FRP Applications

Some domains where FRP or FRP-inspired approaches have been used:

- Graphical Animation
- Robotics
- Vision
- Sound synthesis
- GUIs
- Virtual Reality Environments
- Games

FRP Applications

Some domains where FRP or FRP-inspired approaches have been used:

- Graphical Animation
- Robotics
- Vision
- Sound synthesis
- GUIs
- Virtual Reality Environments
- ***GAMES***

Key FRP Features

Combines conceptual simplicity of the synchronous data flow approach with the flexibility and abstraction power of higher-order functional programming:

Key FRP Features

Combines conceptual simplicity of the synchronous data flow approach with the flexibility and abstraction power of higher-order functional programming:

- Synchronous

Key FRP Features

Combines conceptual simplicity of the synchronous data flow approach with the flexibility and abstraction power of higher-order functional programming:

- Synchronous
- First class temporal abstractions

Key FRP Features

Combines conceptual simplicity of the synchronous data flow approach with the flexibility and abstraction power of higher-order functional programming:

- Synchronous
- First class temporal abstractions
- Hybrid: mixed continuous and discrete time

Key FRP Features

Combines conceptual simplicity of the synchronous data flow approach with the flexibility and abstraction power of higher-order functional programming:

- Synchronous
- First class temporal abstractions
- Hybrid: mixed continuous and discrete time
- Dynamic system structure

Key FRP Features

Combines conceptual simplicity of the synchronous data flow approach with the flexibility and abstraction power of higher-order functional programming:

- Synchronous
- First class temporal abstractions
- Hybrid: mixed continuous and discrete time
- Dynamic system structure

Good fit for typical video games

Key FRP Features

Combines conceptual simplicity of the synchronous data flow approach with the flexibility and abstraction power of higher-order functional programming:

- Synchronous
- First class temporal abstractions
- Hybrid: mixed continuous and discrete time
- Dynamic system structure

Good fit for typical video games
(but not everything labelled “FRP” supports them all).

Yampa

Yampa

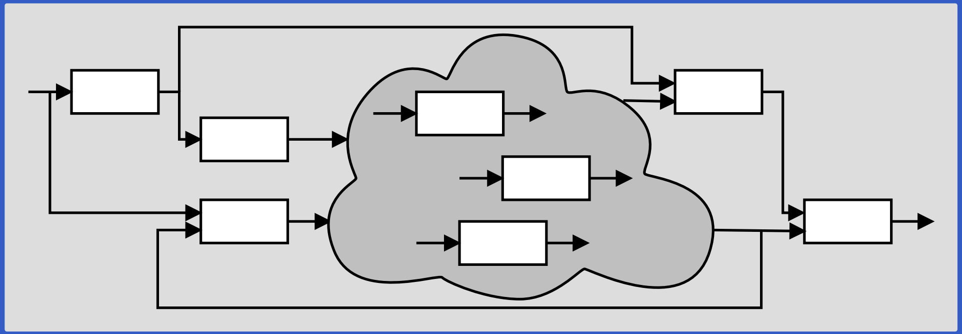
- FRP implemenattion embedded in Haskell

Yampa

- FRP implemenattion embedded in Haskell
- Key concepts:
  - ***Signals***: time-varying values
  - ***Signal Functions***: functions on signals
  - ***Switching*** between signal functions

Yampa

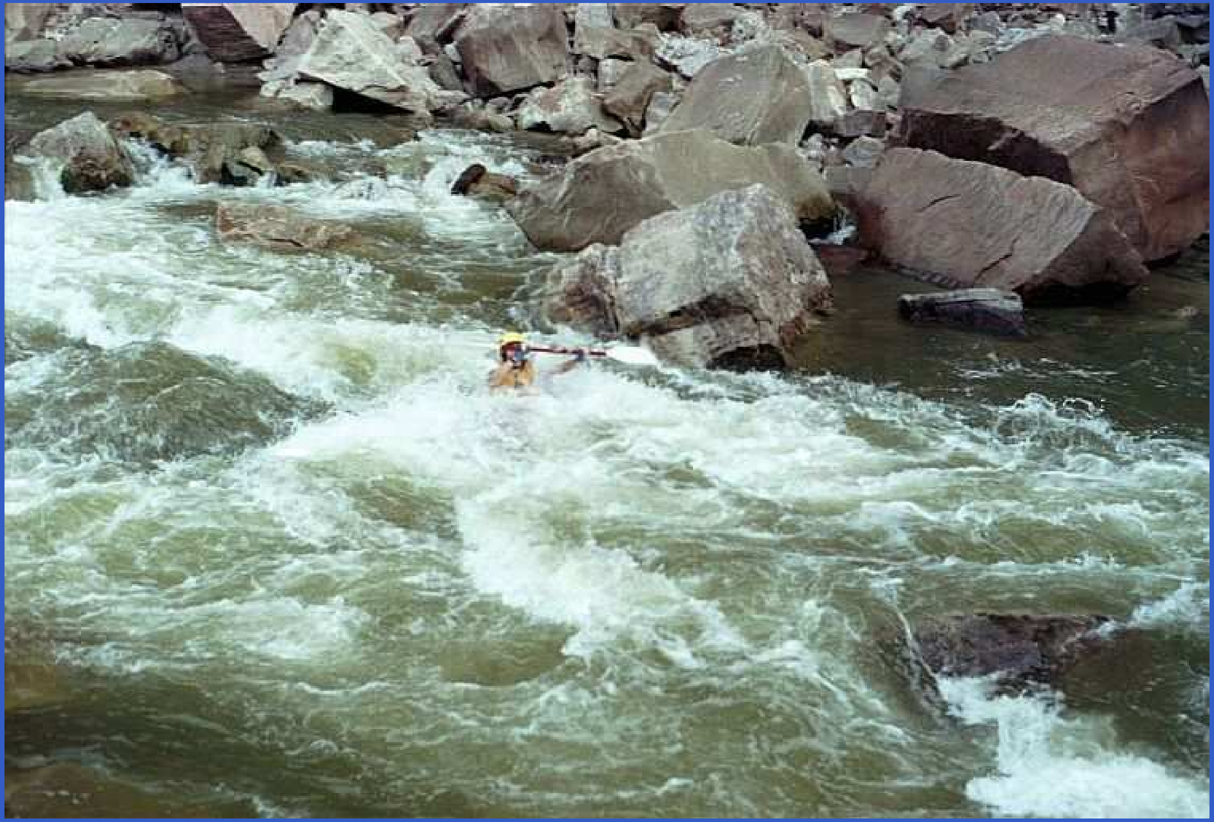
- FRP implemenattion embedded in Haskell
- Key concepts:
  - ***Signals***: time-varying values
  - ***Signal Functions***: functions on signals
  - ***Switching*** between signal functions
- Programming model:



Yampa?

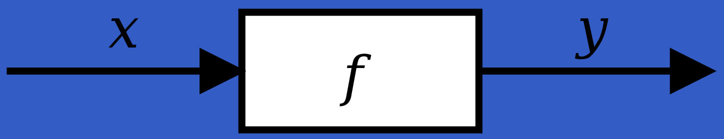
Yampa?

Yampa is a river with long calmly flowing sections and abrupt whitewater transitions in between.

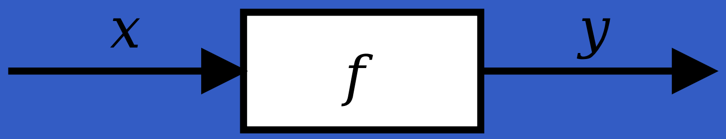


A good metaphor for hybrid systems!

Signal Functions

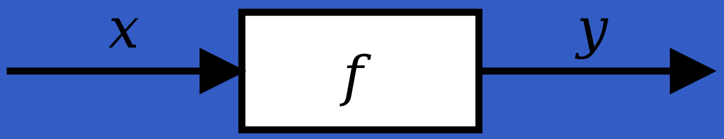


Signal Functions



Intuition:

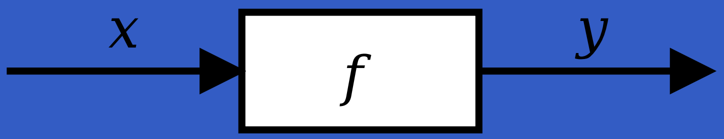
Signal Functions



Intuition:

$$Time \approx \mathbb{R}$$

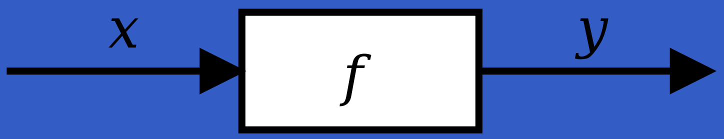
Signal Functions



Intuition:

$$Time \approx \mathbb{R}$$
$$Signal\ a \approx Time \rightarrow a$$
$$x :: Signal\ T1$$
$$y :: Signal\ T2$$

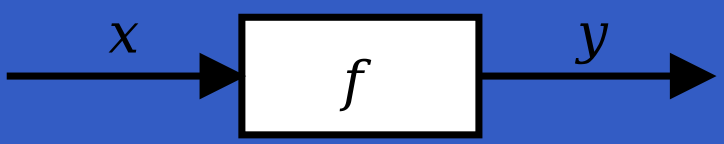
Signal Functions



Intuition:

$$Time \approx \mathbb{R}$$
$$Signal\ a \approx Time \rightarrow a$$
$$x :: Signal\ T1$$
$$y :: Signal\ T2$$
$$SF\ a\ b \approx Signal\ a \rightarrow Signal\ b$$
$$f :: SF\ T1\ T2$$

Signal Functions



Intuition:

$$
\begin{aligned}
&\mathit{Time} \approx \mathbb{R} \\
&\mathit{Signal}\ a \approx \mathit{Time} \rightarrow a \\
&x :: \mathit{Signal}\ \mathit{T1} \\
&y :: \mathit{Signal}\ \mathit{T2} \\
&\mathit{SF}\ a\ b \approx \mathit{Signal}\ a \rightarrow \mathit{Signal}\ b \\
&f :: \mathit{SF}\ \mathit{T1}\ \mathit{T2}
\end{aligned}
$$

Additionally, ***causality*** required: output at time t must be determined by input on interval $[0, t]$.

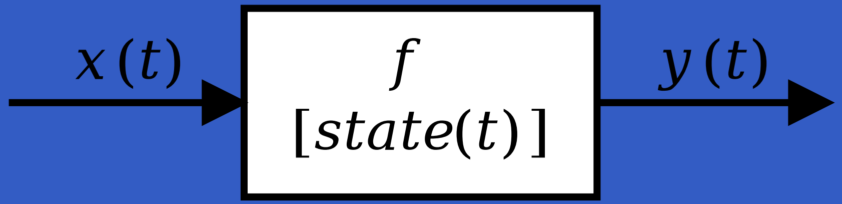
Signal Functions and State

Alternative view:

Signal Functions and State

Alternative view:

Signal functions can encapsulate ***state***.

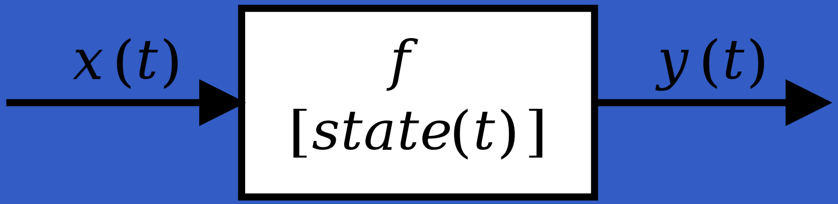


$state(t)$ summarizes input history $x(t')$, $t' \in [0, t]$.

Signal Functions and State

Alternative view:

Signal functions can encapsulate ***state***.



$state(t)$ summarizes input history $x(t')$, $t' \in [0, t]$.

From this perspective, signal functions are:

- ***stateful*** if $y(t)$ depends on $x(t)$ and $state(t)$
- ***stateless*** if $y(t)$ depends only on $x(t)$

Some Basic Signal Functions

$$identity :: SF\ a\ a$$

Some Basic Signal Functions

$$identity :: SF\ a\ a$$

$$constant :: b \rightarrow SF\ a\ b$$

Some Basic Signal Functions

$$identity :: SF\ a\ a$$

$$constant :: b \rightarrow SF\ a\ b$$

$$iPre :: a \rightarrow SF\ a\ a$$

Some Basic Signal Functions

$$identity :: SF\ a\ a$$

$$constant :: b \rightarrow SF\ a\ b$$

$$iPre :: a \rightarrow SF\ a\ a$$

$$integral :: VectorSpace\ a\ s \Rightarrow SF\ a\ a$$

$$y(t) = \int_0^t x(\tau)\,\mathrm{d}\tau$$

Some Basic Signal Functions

$$identity :: SF\ a\ a$$

$$constant :: b \rightarrow SF\ a\ b$$

$$iPre :: a \rightarrow SF\ a\ a$$

$$integral :: VectorSpace\ a\ s \Rightarrow SF\ a\ a$$

$$y(t) = \int_{0}^{t} x(\tau)\,\mathrm{d}\tau$$

Which are stateless and which are stateful?

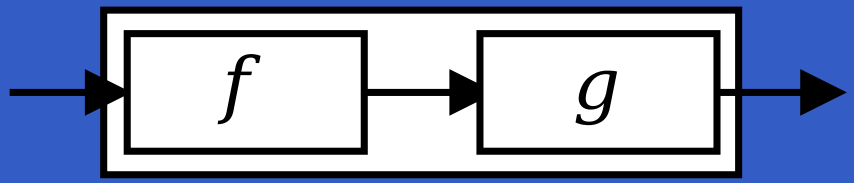
Composition

In Yampa, systems are described by combining signal functions (forming new signal functions).

Composition

In Yampa, systems are described by combining signal functions (forming new signal functions).

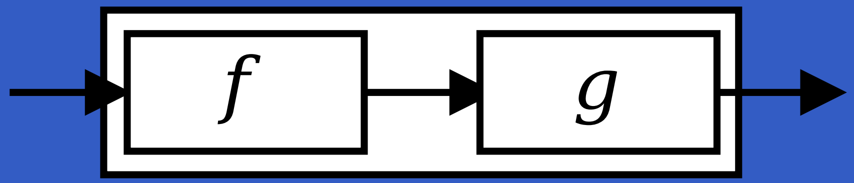
For example, serial composition:



Composition

In Yampa, systems are described by combining signal functions (forming new signal functions).

For example, serial composition:



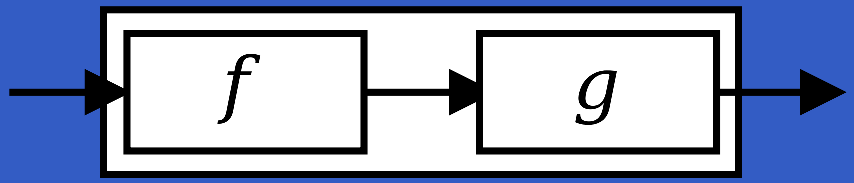
A *combinator* that captures this idea:

$$(\ggg) :: SF\ a\ b \rightarrow SF\ b\ c \rightarrow SF\ a\ c$$

Composition

In Yampa, systems are described by combining signal functions (forming new signal functions).

For example, serial composition:



A *combinator* that captures this idea:

$$(\ggg) :: SF\ a\ b \rightarrow SF\ b\ c \rightarrow SF\ a\ c$$

Signal functions are the primary notion; signals a secondary one, only existing indirectly.

Time

Quick exercise: Define time!

$$time :: SF\ a\ Time$$

Time

Quick exercise: Define time!

$$time :: SF\ a\ Time$$

$$time = constant\ 1.0 \ggg integral$$

Time

Quick exercise: Define time!

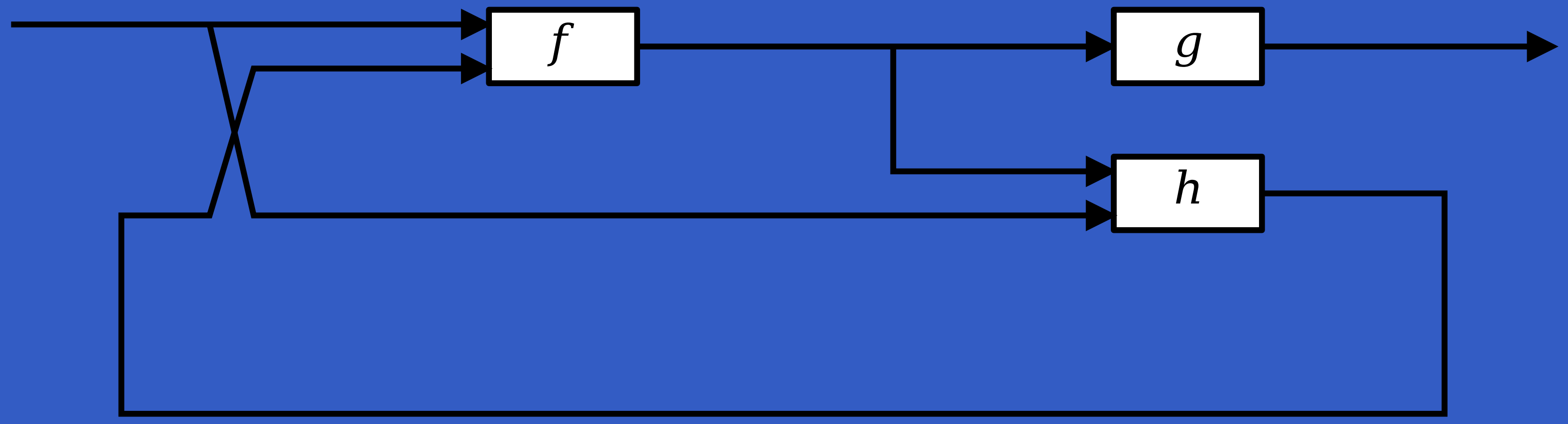
$$time :: SF\ a\ Time$$

$$time = constant\ 1.0 \ggg integral$$

Note: there is ***no*** built-in notion of global time in Yampa: time is always ***local***, measured from when a signal function started.

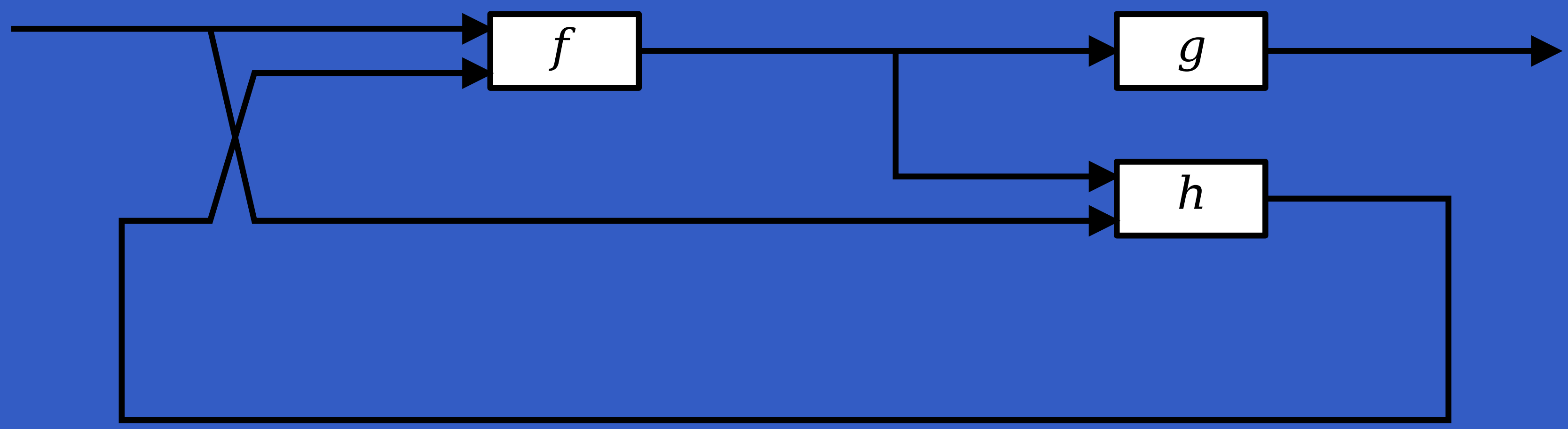
Systems

What about larger networks?
How many combinators are needed?



Systems

What about larger networks?
How many combinators are needed?



John Hughes's ***Arrow*** framework provides a good answer!

The Arrow framework (1)

John Hughes' ***arrow*** framework:

- Abstract data type interface for ***function-like objects*** (or "blocks") with ***effects***.

The Arrow framework (1)

John Hughes' ***arrow*** framework:

- Abstract data type interface for ***function-like objects*** (or "blocks") with ***effects***.
- Particularly suitable for types representing ***process-like computations***.

The Arrow framework (1)

John Hughes' ***arrow*** framework:

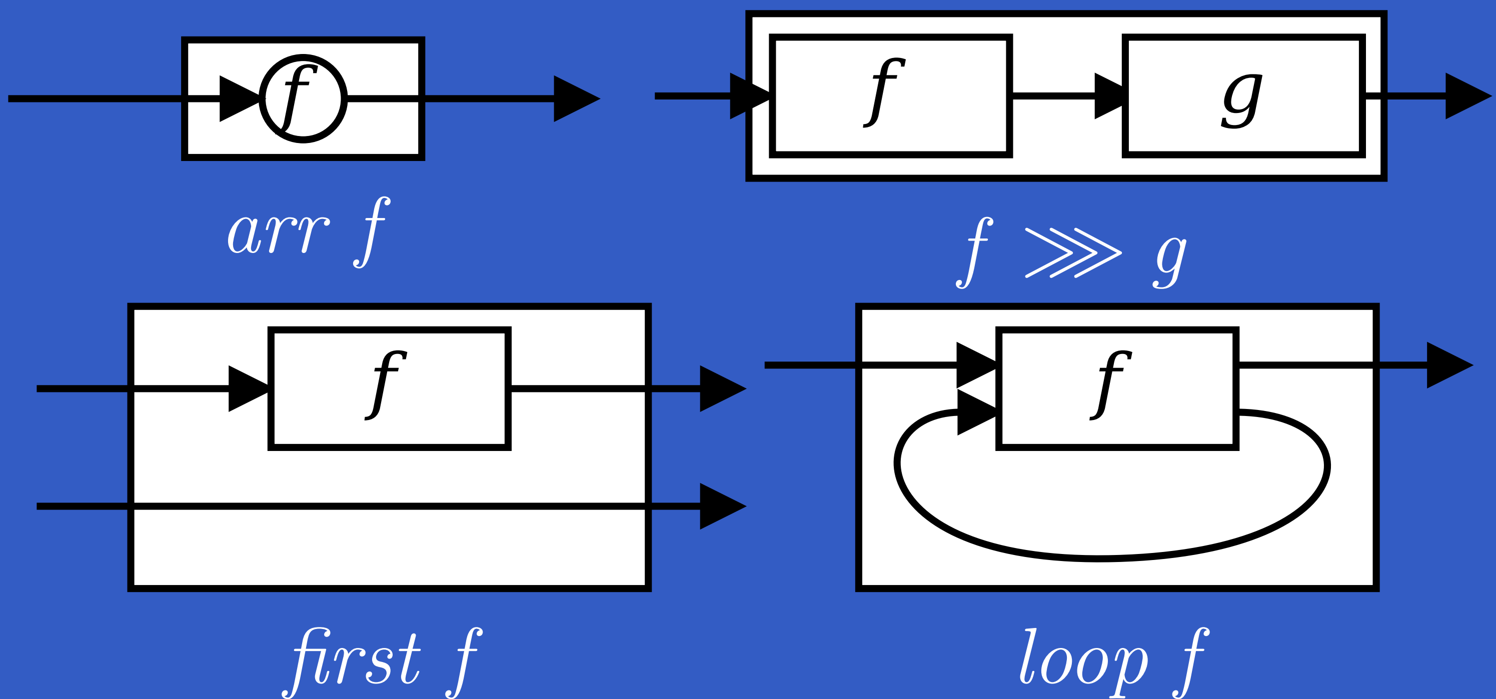
- Abstract data type interface for ***function-like objects*** (or "blocks") with ***effects***.
- Particularly suitable for types representing ***process-like computations***.
- Related to ***monads***, since arrows are computations, but more general.

The Arrow framework (1)

John Hughes' ***arrow*** framework:

- Abstract data type interface for ***function-like objects*** (or "blocks") with ***effects***.
- Particularly suitable for types representing ***process-like computations***.
- Related to ***monads***, since arrows are computations, but more general.
- Provides a minimal set of "wiring" combinators.

The Arrow framework (2)



$$\begin{aligned}
arr &:: (a \rightarrow b) \rightarrow SF\ a\ b \\
(\ggg) &:: SF\ a\ b \rightarrow SF\ b\ c \rightarrow SF\ a\ c \\
first &:: SF\ a\ b \rightarrow SF\ (a, c)\ (b, c) \\
loop &:: SF\ (a, c)\ (b, c) \rightarrow SF\ a\ b
\end{aligned}$$

The Arrow framework (3)

Examples:

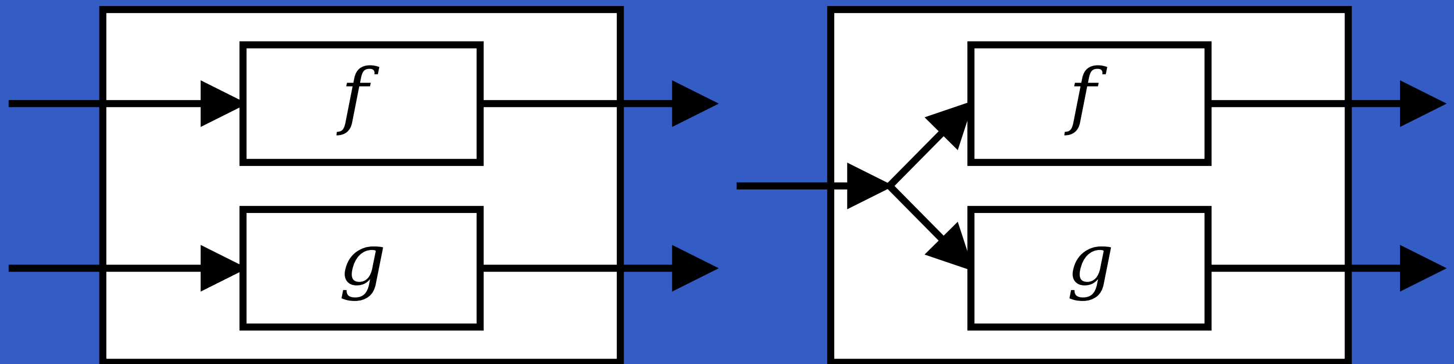
$$identity :: SF\ a\ a$$
$$identity = arr\ id$$

$$constant :: b \rightarrow SF\ a\ b$$
$$constant\ b = arr\ (const\ b)$$

$$\hat{}\!\ll :: (b \rightarrow c) \rightarrow SF\ a\ b \rightarrow SF\ a\ c$$
$$f\ \hat{}\!\ll\ sf = sf \ggg arr\ f$$

The Arrow framework (4)

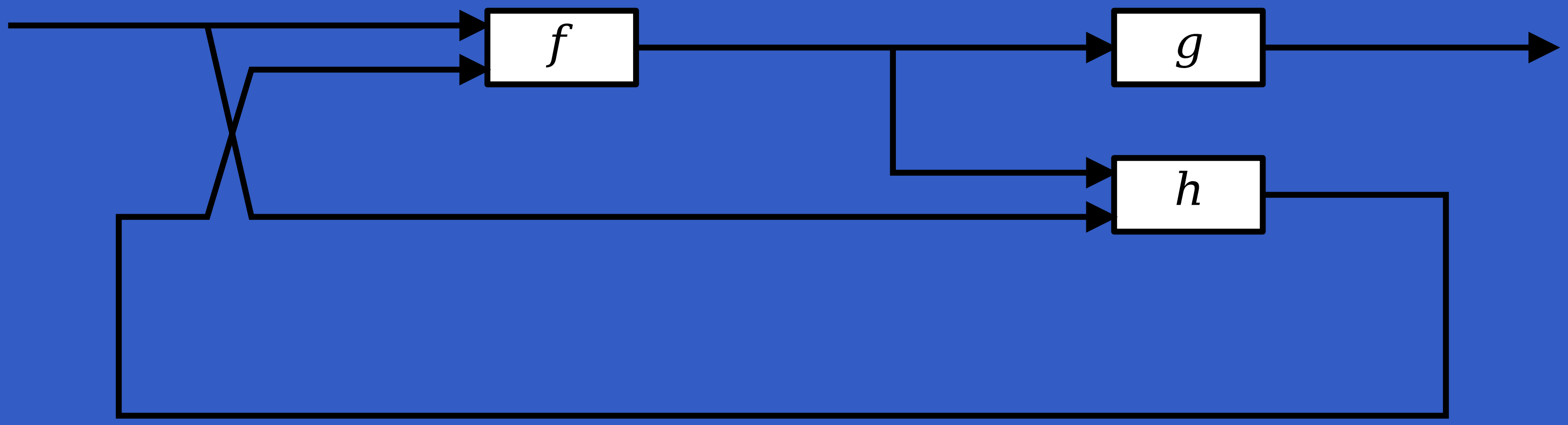
Some derived combinators:



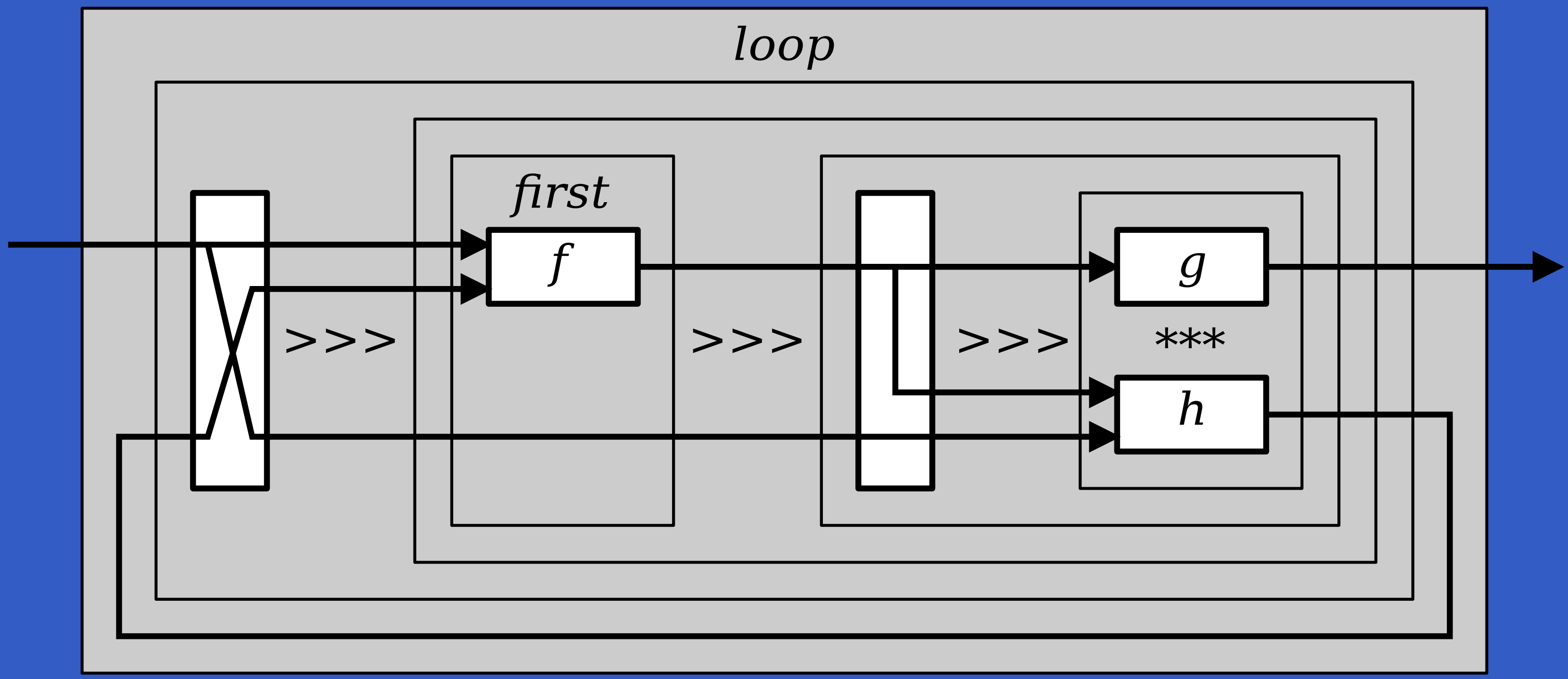
$f \ast\!\ast\!\ast\; g$ $\qquad$ $f \;\&\&\&\; g$

$$(\ast\!\ast\!\ast) :: SF\ a\ b \rightarrow SF\ c\ d \rightarrow SF\ (a, c)\ (b, d)$$
$$(\&\&\&) :: SF\ a\ b \rightarrow SF\ a\ c \rightarrow SF\ a\ (b, c)$$

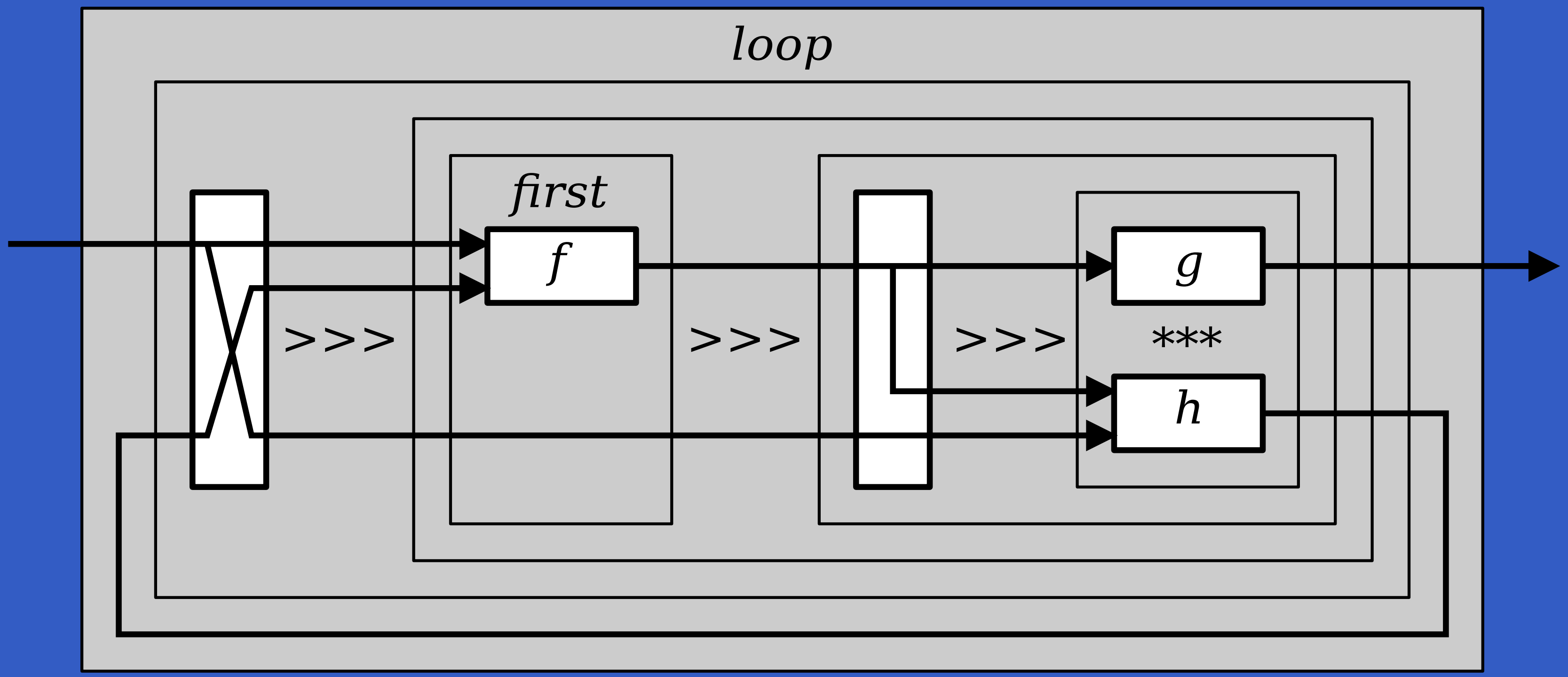
Constructing a network



Constructing a network

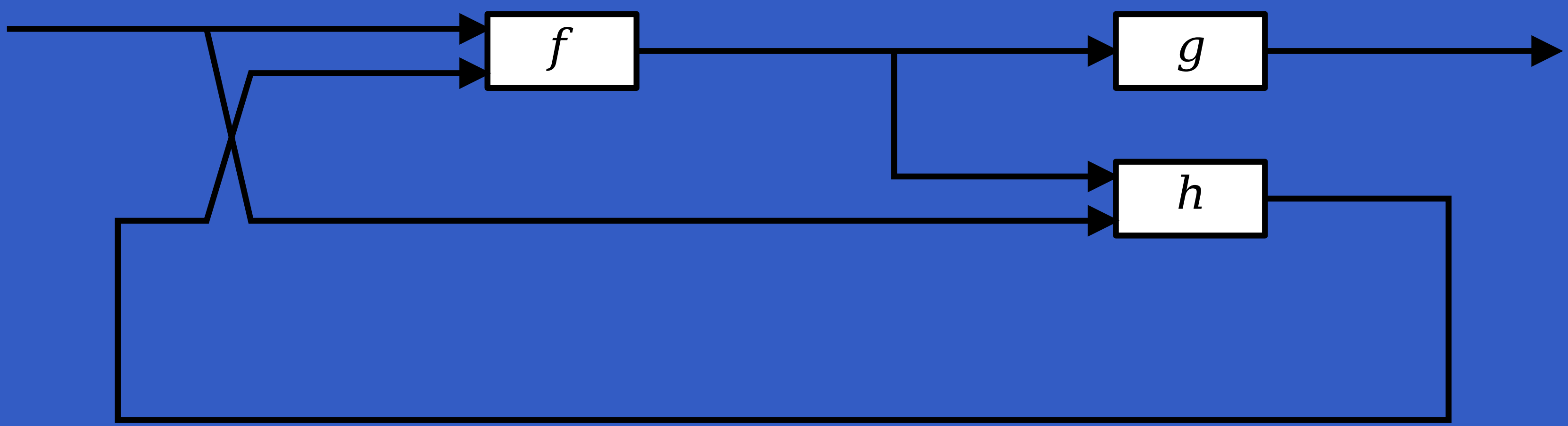


Constructing a network

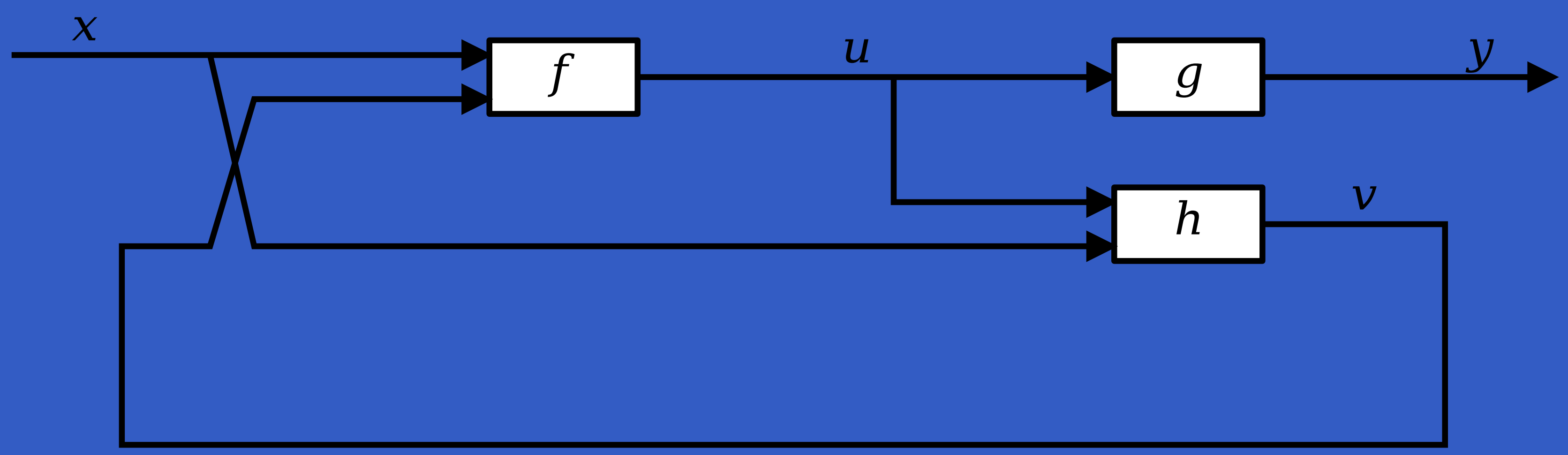


$$loop\ (arr\ (\lambda(x,y) \to ((x,y),x))$$
$$\ggg (first\ f$$
$$\ggg (arr\ (\lambda(x,y) \to (x,(x,y))) \ggg (g *\!\!*\!\!* h))))$$

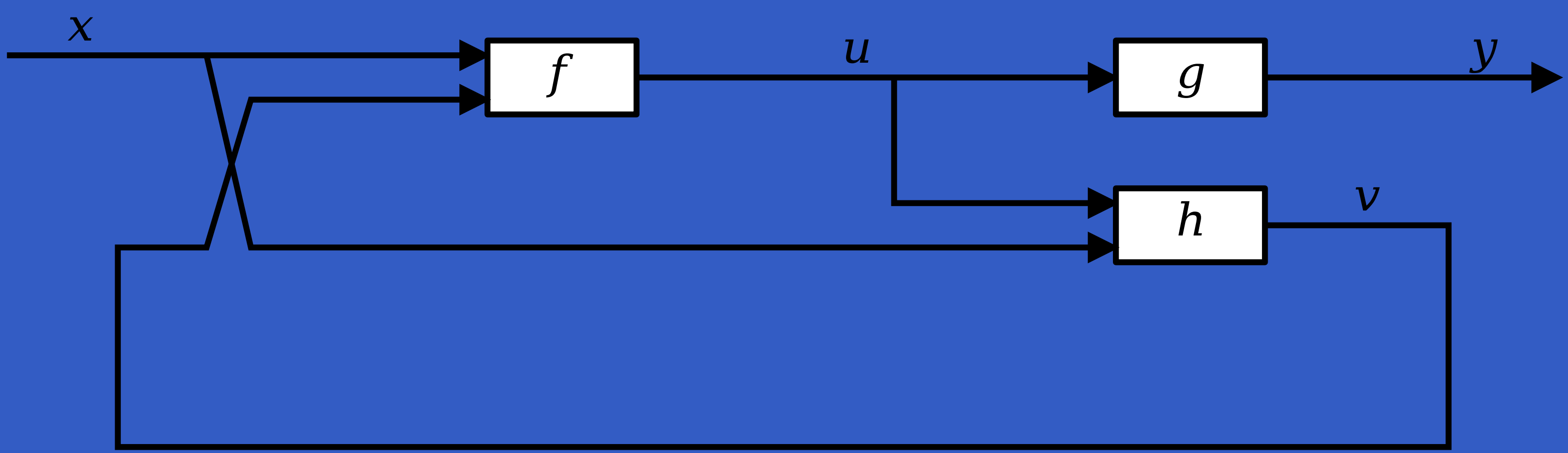
Arrow notation



Arrow notation



Arrow notation



proc $x \rightarrow$ **do**

rec

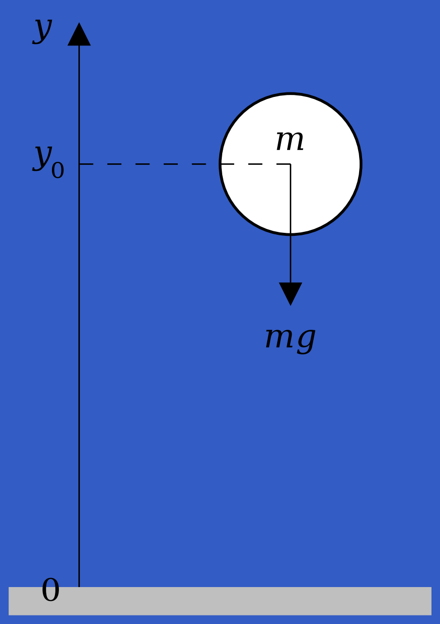
$u \leftarrow f \prec (x, v)$

$y \leftarrow g \prec u$

$v \leftarrow h \prec (u, x)$

$returnA \prec y$

A Bouncing Ball



$$
\begin{aligned}
y &= y_0 + \int v \, \mathrm{d}t \\
v &= v_0 + \int -9.81
\end{aligned}
$$

On impact:

$$v = -v(t-)$$

(fully elastic collision)

Modelling the Bouncing Ball: Part 1

Free-falling ball:

$$
\begin{aligned}
&\textbf{type } \mathit{Pos} = \mathit{Double} \\
&\textbf{type } \mathit{Vel} = \mathit{Double} \\
&\mathit{fallingBall} :: \mathit{Pos} \rightarrow \mathit{Vel} \rightarrow \mathit{SF}\ ()\ (\mathit{Pos}, \mathit{Vel}) \\
&\mathit{fallingBall}\ \mathit{y0}\ \mathit{v0} = \textbf{proc}\ () \rightarrow \textbf{do} \\
&\quad v \leftarrow (\mathit{v0}+)\ \hat{}\!\!\ll \mathit{integral} \prec -9.81 \\
&\quad y \leftarrow (\mathit{y0}+)\ \hat{}\!\!\ll \mathit{integral} \prec v \\
&\quad \mathit{returnA} \prec (y, v)
\end{aligned}
$$

Discrete-time Signals or Events

Yampa's signals are conceptually ***continuous-time*** signals.

Discrete-time Signals or Events

Yampa's signals are conceptually ***continuous-time*** signals.

Discrete-time signals: signals defined at discrete points in time.

Discrete-time Signals or Events

Yampa's signals are conceptually ***continuous-time*** signals.

Discrete-time signals: signals defined at discrete points in time.

Yampa models discrete-time signals by lifting the ***co-domain*** of signals using an option-type:

$$\mathbf{data}\ Event\ a = NoEvent \mid Event\ a$$

Discrete-time Signals or Events

Yampa's signals are conceptually ***continuous-time*** signals.

Discrete-time signals: signals defined at discrete points in time.

Yampa models discrete-time signals by lifting the ***co-domain*** of signals using an option-type:

$$\mathbf{data}\ Event\ a = NoEvent \mid Event\ a$$

Discrete-time signal $= \mathtt{Signal}\,(\mathtt{Event}\,\alpha)$.

Some Event Functions and Sources

$$tag :: Event\ a \rightarrow b \rightarrow Event\ b$$

$$never :: SF\ a\ (Event\ b)$$

$$now :: b \rightarrow SF\ a\ (Event\ b)$$

$$after :: Time \rightarrow b \rightarrow SF\ a\ (Event\ b)$$

$$repeatedly :: Time \rightarrow b \rightarrow SF\ a\ (Event\ b)$$

$$edge :: SF\ Bool\ (Event\ ())$$

$$notYet :: SF\ (Event\ a)\ (Event\ a)$$

$$once :: SF\ (Event\ a)\ (Event\ a)$$

Modelling the Bouncing Ball: Part 2

Detecting when the ball goes through the floor:

$$
\begin{array}{l}
\mathit{fallingBall'} :: \\
\quad \mathit{Pos} \to \mathit{Vel} \to \mathit{SF}\ ()\ ((\mathit{Pos}, \mathit{Vel}), \mathit{Event}\ (\mathit{Pos}, \mathit{Vel})) \\
\mathit{fallingBall'}\ \mathit{y0}\ \mathit{v0} = \mathbf{proc}\ () \to \mathbf{do} \\
\quad \mathit{yv}@(y, \_) \leftarrow \mathit{fallingBall}\ \mathit{y0}\ \mathit{v0} \prec ()\\
\quad \mathit{hit} \qquad\qquad \leftarrow \mathit{edge} \qquad\qquad\qquad\quad \prec y \leqslant 0 \\
\quad \mathit{returnA} \prec (\mathit{yv}, \mathit{hit}\ `\mathit{tag}`\ \mathit{yv})
\end{array}
$$

Switching

Q: How and when do signal functions “start”?

Switching

Q: How and when do signal functions "start"?

A:
- ***Switchers*** "apply" a signal functions to its input signal at some point in time.

Switching

Q: How and when do signal functions “start”?

A:
- ***Switchers*** “apply” a signal functions to its input signal at some point in time.
- This creates a “running” signal function ***instance***.

Switching

Q: How and when do signal functions “start”?

A:
- ***Switchers*** “apply” a signal functions to its input signal at some point in time.
- This creates a “running” signal function ***instance***.
- The new signal function instance often replaces the previously running instance.

Switching

Q: How and when do signal functions "start"?

A:
- ***Switchers*** "apply" a signal functions to its input signal at some point in time.
- This creates a "running" signal function ***instance***.
- The new signal function instance often replaces the previously running instance.

Switchers thus allow systems with ***varying structure*** to be described.

The Basic Switch

Idea:

- Allows one signal function to be replaced by another.
- Switching takes place on the first occurrence of the switching event source.

$$\begin{aligned} &\mathit{switch} :: \\ &\quad \mathit{SF}\ a\ (b, \mathit{Event}\ c) \\ &\quad \rightarrow (c \rightarrow \mathit{SF}\ a\ b) \\ &\quad \rightarrow \mathit{SF}\ a\ b \end{aligned}$$

The Basic Switch

Idea:

- Allows one signal function to be replaced by another.
- Switching takes place on the first occurrence of the switching event source.

Initial SF with event source

$$
\begin{aligned}
&switch :: \\
&\quad \boxed{SF\ a\ (b, Event\ c)} \\
&\quad \rightarrow\ (c \rightarrow SF\ a\ b) \\
&\quad \rightarrow SF\ a\ b
\end{aligned}
$$

The Basic Switch

Idea:

- Allows one signal function to be replaced by another.
- Switching takes place on the first occurrence of the switching event source.

$$
\begin{aligned}
&\mathit{switch} :: \\
&\quad SF\ a\ (b, \mathit{Event}\ c) \\
&\quad \rightarrow \boxed{(c \rightarrow SF\ a\ b)} \\
&\quad \rightarrow SF\ a\ b
\end{aligned}
$$

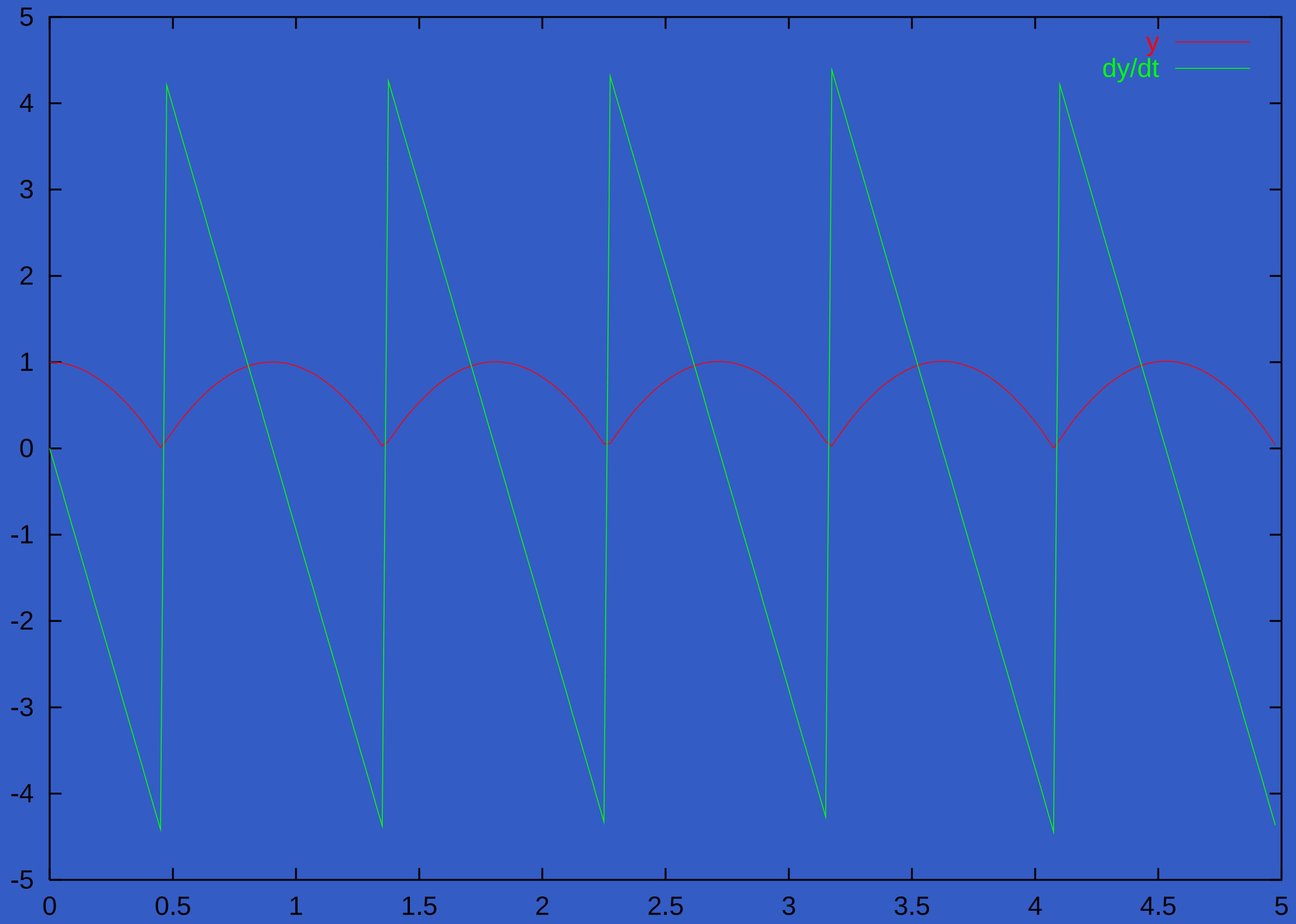
Function yielding SF to switch into

Modelling the Bouncing Ball: Part 3

Making the ball bounce:

$$
\begin{aligned}
&\mathit{bouncingBall} :: \mathit{Pos} \rightarrow \mathit{SF}\ ()\ (\mathit{Pos}, \mathit{Vel})\\
&\mathit{bouncingBall}\ \mathit{y0} = \mathit{bbAux}\ \mathit{y0}\ 0.0\\
&\quad \mathbf{where}\\
&\qquad \mathit{bbAux}\ \mathit{y0}\ \ \mathit{v0} =\\
&\qquad\quad \mathit{switch}\ (\mathit{fallingBall'}\ \mathit{y0}\ \ \mathit{v0})\ \$\ \lambda(y, v) \rightarrow\\
&\qquad\quad \mathit{bbAux}\ y\ (-v)
\end{aligned}
$$

Simulation of the Bouncing Ball



The Decoupled Switch

$$
\begin{aligned}
&dSwitch :: \\
&\quad SF\ a\ (b, Event\ c) \\
&\quad \rightarrow (c \rightarrow SF\ a\ b) \\
&\quad \rightarrow SF\ a\ b
\end{aligned}
$$

The Decoupled Switch

$$
\begin{aligned}
& dSwitch :: \\
& \quad SF\ a\ (b, Event\ c) \\
& \quad \rightarrow (c \rightarrow SF\ a\ b) \\
& \quad \rightarrow SF\ a\ b
\end{aligned}
$$

- Output at the point of switch is taken from the old subordinate signal function, ***not*** the new residual signal function.

The Decoupled Switch

$$\begin{aligned}&dSwitch :: \\ &\quad SF\ a\ (b, Event\ c) \\ &\quad \rightarrow (c \rightarrow SF\ a\ b) \\ &\quad \rightarrow SF\ a\ b\end{aligned}$$

- Output at the point of switch is taken from the old subordinate signal function, ***not*** the new residual signal function.
- ***Output*** at the current point in time thus ***independent*** of whether or not the ***switching event*** occurs at that point. Hence decoupled. Useful e.g. in some feedback scenarios.

Game Objects (1)

Observable aspects of game entities:

$$
\begin{aligned}
&\mathbf{data}\ \mathit{Object} = \mathit{Object}\ \{\\
&\quad \mathit{objectName} :: \mathit{ObjectName},\\
&\quad \mathit{objectKind}\ \ :: \mathit{ObjectKind},\\
&\quad \mathit{objectPos}\ \ \ :: \mathit{Pos2D},\\
&\quad \mathit{objectVel}\ \ \ :: \mathit{Vel2D},\\
&\quad \mathit{objectAcc}\ \ \ :: \mathit{Acc2D},\\
&\quad \mathit{objectDead}\ :: \mathit{Bool},\\
&\quad \mathit{objectHit}\ \ \ :: \mathit{Bool},\\
&\qquad \ldots\\
&\}
\end{aligned}
$$

Game Objects (2)

$$
\begin{aligned}
\textbf{data}\ \mathit{ObjectKind} = \ & \mathit{Ball}\quad \mathit{Double} \\
\mid\ & \mathit{Paddle}\ \mathit{Size2D} \\
\mid\ & \mathit{Block}\quad \mathit{Energy}\ \mathit{Size2D} \\
\mid\ & \mathit{Side}\quad\ \ \mathit{Side}
\end{aligned}
$$

Game Objects (3)

$$\textbf{type}\ ObjectSF = SF\ ObjectInput\ ObjectOutput$$

$$\begin{array}{l}
\textbf{data}\ ObjectInput = ObjectInput\ \{ \\
\quad \begin{array}{ll} userInput & :: Controller, \\ collisions & :: [\,Collision\,], \\ knownObjects & :: [\,Object\,] \end{array} \\
\}
\end{array}$$

$$\begin{array}{l}
\textbf{data}\ ObjectOutput = ObjectOutput\ \{ \\
\quad \begin{array}{ll} outputObject & :: Object, \\ harakiri & :: Event\ () \end{array} \\
\}
\end{array}$$

Observing the Game World

- Note that $[\,Object\,]$ appears in the input type.

Observing the Game World

- Note that $[Object]$ appears in the input type.
- This allows each game object to observe ***all*** live game objects.

Observing the Game World

- Note that $[\,Object\,]$ appears in the input type.
- This allows each game object to observe ***all*** live game objects.
- Similarly, $[\,Collision\,]$ allows interactions ***between*** game objects to be observed.

Observing the Game World

- Note that $[\mathit{Object}]$ appears in the input type.
- This allows each game object to observe ***all*** live game objects.
- Similarly, $[\mathit{Collision}]$ allows interactions ***between*** game objects to be observed.
- Typically achieved through delayed feedback to ensure the feedback is well-defined:

$$
\begin{array}{l}
\mathit{loopPre} :: c \rightarrow \mathit{SF}\ (a, c)\ (b, c) \rightarrow \mathit{SF}\ a\ b \\
\mathit{loopPre}\ \mathit{c\_init}\ \mathit{sf} = \\
\quad \mathit{loop}\ (\mathit{second}\ (\mathit{iPre}\ \mathit{c\_init}) \ggg \mathit{sf})
\end{array}
$$

Paddle, Take 1

$$
\begin{aligned}
&\mathit{objPaddle} :: \mathit{ObjectSF}\\
&\mathit{objPaddle} = \mathbf{proc}\ (\mathit{ObjectInput}\ \mathit{ci}\ \mathit{cs}\ \mathit{os}) \rightarrow \mathbf{do}\\
&\quad \mathbf{let}\ \mathit{name} = \texttt{"paddle"}\\
&\quad \mathbf{let}\ \mathit{isHit}\ = \mathit{inCollision}\ \mathit{name}\ \mathit{cs}\\
&\quad \mathbf{let}\ p \qquad = \mathit{refPosPaddle}\ \mathit{ci}\\
&\quad v \leftarrow \mathit{derivative} \prec p\\
&\quad \mathit{returnA} \prec \mathit{livingObject}\ \$\ \mathit{Object}\ \{\\
&\qquad\qquad\qquad\qquad\qquad\qquad \mathit{objectName} = \mathit{name},\\
&\qquad\qquad\qquad\qquad\qquad\qquad \mathit{objectPos} = p,\\
&\qquad\qquad\qquad\qquad\qquad\qquad \mathit{objectVel}\ = v,\\
&\qquad\qquad\qquad\qquad\qquad\qquad \ldots\}
\end{aligned}
$$

Paddle, Take 2

$$
\begin{array}{l}
\mathit{objPaddle} :: \mathit{ObjectSF} \\
\mathit{objPaddle} = \textbf{proc}\ (\mathit{ObjectInput}\ \mathit{ci}\ \mathit{cs}\ \mathit{os}) \rightarrow \textbf{do} \\
\quad \textbf{let}\ \mathit{name} = \texttt{"paddle"} \\
\quad \textbf{let}\ \mathit{isHit}\ = \mathit{inCollision}\ \mathit{name}\ \mathit{cs} \\
\quad \textbf{rec} \\
\qquad \textbf{let}\ v = \mathit{limitNorm}\ (20.0 \mathbin{*\hat{}}\ (\mathit{refPosPaddle}\ \mathit{ci} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \hat{}{-}\hat{}\ p)) \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathit{maxVNorm} \\
\qquad p \leftarrow (\mathit{initPosPaddle}\ \hat{}{+}\hat{}\ )\hat{}{\ll}\ \mathit{integral} \prec v \\
\quad \mathit{returnA} \prec \mathit{livingObject}\ \$\ \mathit{Object}\ \{\ldots\}
\end{array}
$$

Ball, Take 1

$$
\begin{array}{l}
objBall :: ObjectSF \\
objBall = \\
\quad switch\ followPaddleDetectLaunch \mathbin{\$} \lambda p \rightarrow \\
\quad objBall
\end{array}
$$

$$
\begin{array}{l}
followPaddleDetectLaunch = \textbf{proc}\ oi \rightarrow \textbf{do} \\
\quad o \quad\ \ \leftarrow followPaddle \prec oi \\
\quad click \leftarrow edge \qquad\qquad\quad \prec controllerClick \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad (userInput\ oi) \\
\quad returnA \prec (o, click\ `tag`\ (objectPos \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad (outputObject\ o)))
\end{array}
$$

Ball, Take 2

$$
\begin{array}{l}
objBall :: ObjectSF \\
objBall = \\
\quad switch\ followPaddleDetectLaunch\ \$\ \lambda p \rightarrow \\
\quad switch\ (freeBall\ p\ initBallVel \mathbin{\&\&\&} never)\ \$\ \lambda\_ \rightarrow \\
\quad objBall
\end{array}
$$

$$
\begin{array}{l}
freeBall\ p0\ v0 = \mathbf{proc}\ (ObjectInput\ ci\ cs\ os) \rightarrow \mathbf{do} \\
\quad p \leftarrow (p0\ \hat{}{+}\hat{}\ )\hat{}{\ll}\ integral \prec v0' \\
\quad returnA \prec livingObject\ \$\ \{\ldots\} \\
\quad \mathbf{where} \\
\qquad v0' = limitNorm\ v0\ maxVNorm
\end{array}
$$

Ball, Take 3

$$
\begin{array}{l}
objBall :: ObjectSF \\
objBall = \\
\quad switch\ followPaddleDetectLaunch \qquad \$\ \lambda p \rightarrow \\
\quad switch\ (bounceAroundDetectMiss\ p)\ \$\ \lambda\_ \rightarrow \\
\quad objBall
\end{array}
$$

$$
\begin{array}{l}
bounceAroundDetectMiss\ p = \mathbf{proc}\ oi \rightarrow \mathbf{do} \\
\quad o \qquad \leftarrow bouncingBall\ p\ initBallVel \prec oi \\
\quad miss \leftarrow collisionWithBottom \qquad\quad \prec collisions\ oi \\
\quad returnA \prec (o, miss)
\end{array}
$$

Making the Ball Bounce

$$
\begin{aligned}
&\mathit{bouncingBall}\ p0\ v0 = \\
&\quad \mathit{switch}\ (\mathit{moveFreelyDetBounce}\ p0\ v0)\ \$\ \lambda(p', v') \rightarrow \\
&\quad \mathit{bouncingBall}\ p'\ v'
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{moveFreelyDetBounce}\ p0\ v0 = \\
&\quad \mathbf{proc}\ oi@(\mathit{ObjectInput}\ \_\ cs\ \_) \rightarrow \mathbf{do} \\
&\qquad o \leftarrow \mathit{freeBall}\ p0\ v0 \prec oi \\
&\qquad ev \leftarrow \mathit{edgeJust} \lll \mathit{initially}\ \mathit{Nothing} \\
&\qquad\qquad\quad \prec \mathit{changedVelocity}\ \texttt{"ball"}\ cs \\
&\quad \mathit{returnA} \prec (o, \mathit{fmap}\ (\lambda v \rightarrow (\mathit{objectPos}\ (\ldots o), v)) \\
&\qquad\qquad\qquad\qquad\qquad ev)
\end{aligned}
$$

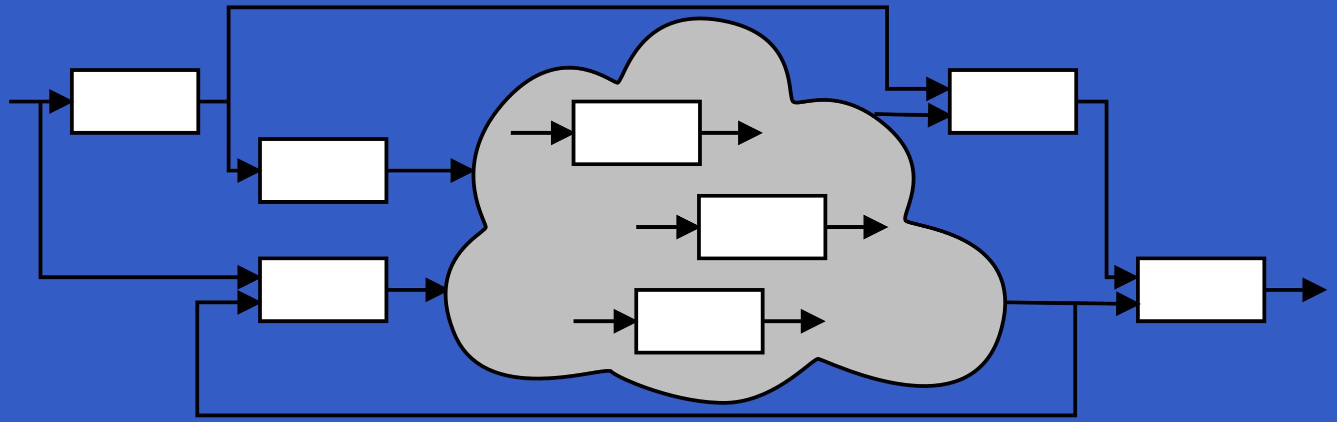
Highly dynamic system structure?

The basic switch allows one signal function to be replaced by another.

Highly dynamic system structure?

The basic switch allows one signal function to be replaced by another.

- What about more general structural changes?

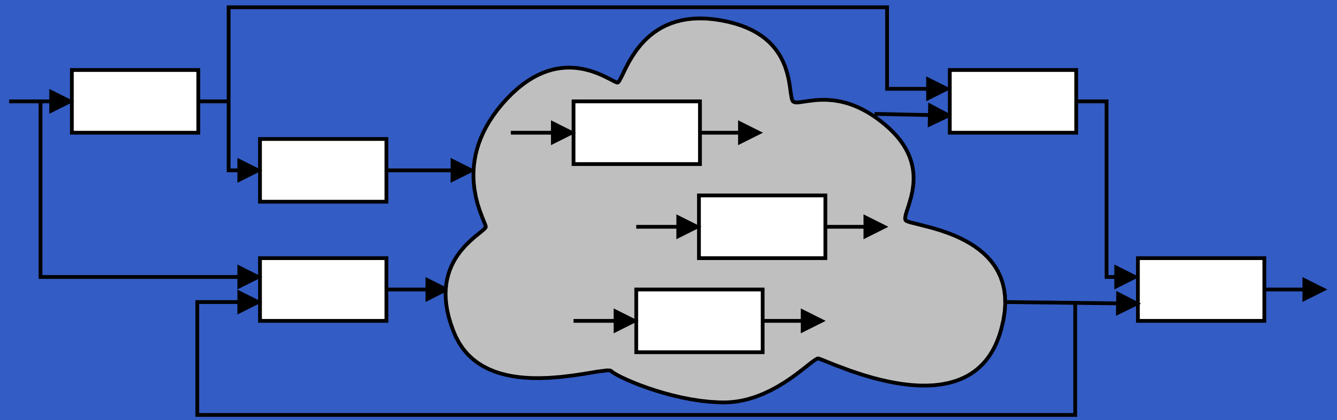


We want blocks to disappear!

Highly dynamic system structure?

The basic switch allows one signal function to be replaced by another.

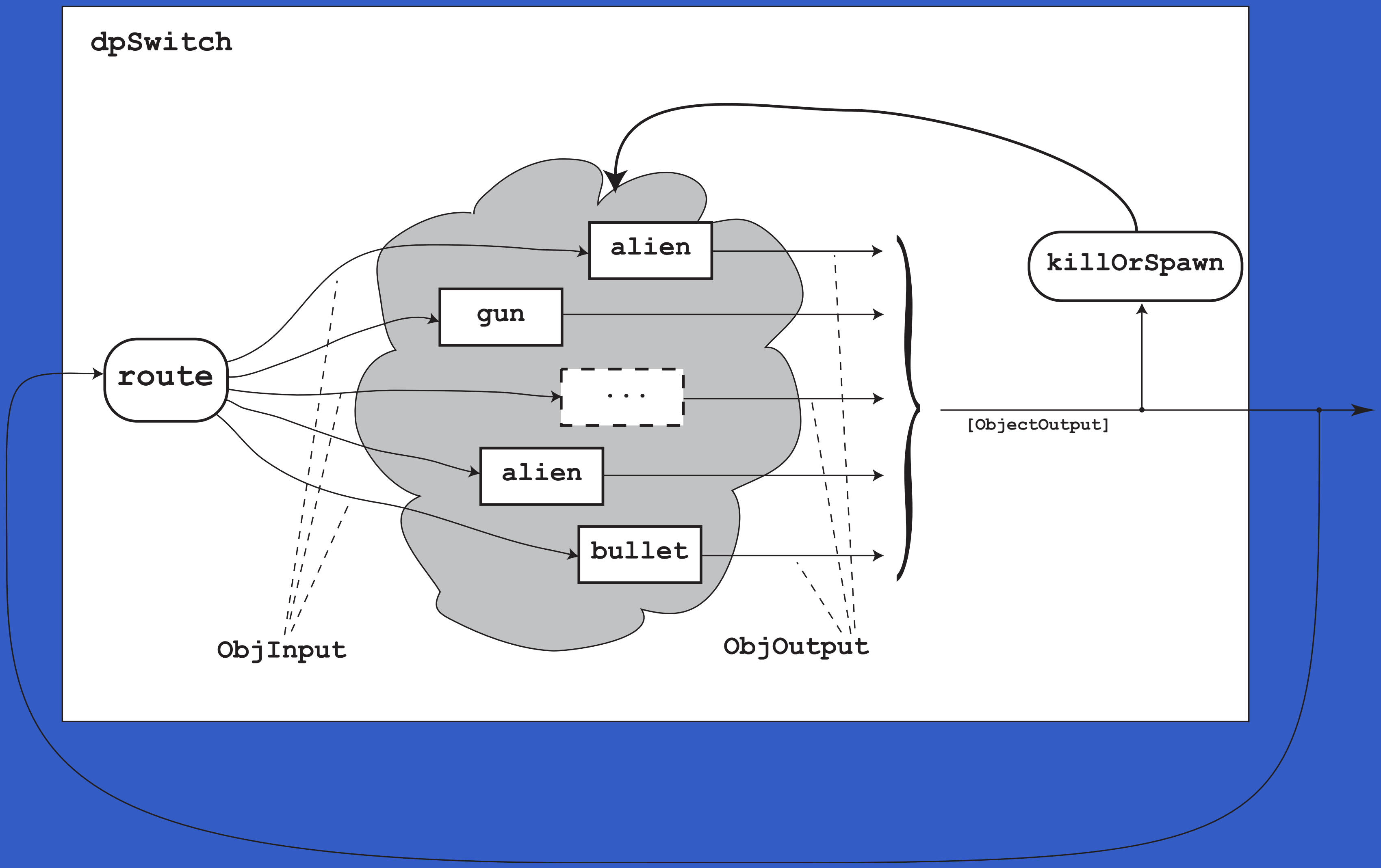
- What about more general structural changes?



We want blocks to disappear!

- What about state?

Typical Overall Game Structure



Dynamic Signal Function Collections

Idea:

Dynamic Signal Function Collections

Idea:

- Switch over ***collections*** of signal functions.

Dynamic Signal Function Collections

Idea:

- Switch over ***collections*** of signal functions.
- On event, “freeze” running signal functions into collection of signal function ***continuations***, preserving encapsulated ***state***.

Dynamic Signal Function Collections

Idea:

- Switch over ***collections*** of signal functions.
- On event, “freeze” running signal functions into collection of signal function ***continuations***, preserving encapsulated ***state***.
- Modify collection as needed and switch back in.

dpSwitch

Need ability to express:

- How input routed to each signal function.
- When collection changes shape.
- How collection changes shape.

```
dpSwitch :: Functor col =>
     (forall sf . (a -> col sf -> col (b,sf)))
   -> col (SF b c)
   -> SF (a, col c) (Event d)
   -> (col (SF b c) -> d -> SF a (col c))
   -> SF a (col c)
```

dpSwitch

Need ability to express:

- How input routed to each signal function.
- When collection changes shape.
- How collection changes shape.

```
dpSwitch :: Functor col =>
    (forall sf . (a -> col sf -> col (b,sf)))
    -> col (SF b c)
    -> SF (a, col c) (Event d)
    -> (col (SF b c) -> d -> SF a (col c))
    -> SF a (col c)
```

Routing function

dpSwitch

Need ability to express:

- How input routed to each signal function.
- When collection changes shape.
- How collection changes shape.

```
dpSwitch :: Functor col =>
    (forall sf . (a -> col sf -> col (b,sf)))
    -> col (SF b c)
    -> SF (a, col c) (Event d)
    -> (col (SF b c) -> d -> SF a (col c))
    -> SF a (col c)
```

Initial collection

dpSwitch

Need ability to express:

- How input routed to each signal function.
- When collection changes shape.
- How collection changes shape.

```
dpSwitch :: Functor col =>
    (forall sf . (a -> col sf -> col (b,sf)))
   -> col (SF b c)
   -> SF (a, col c) (Event d)
   -> (col (SF b c) -> d -> SF a (col c))
   -> SF a (col c)
```

Event source

dpSwitch

Need ability to express:

- How input routed to each signal function.
- When collection changes shape.
- How collection changes shape.

```
dpSwitch :: Functor col =>
    (forall sf . (a -> col sf -> col (b,sf)))
   -> col (SF b c)
   -> SF (a, col c) (Event d)
   -> (col (SF b c) -> d -> SF a (col c))
   -> SF a (col c)
```

Function yielding SF to switch into

Routing

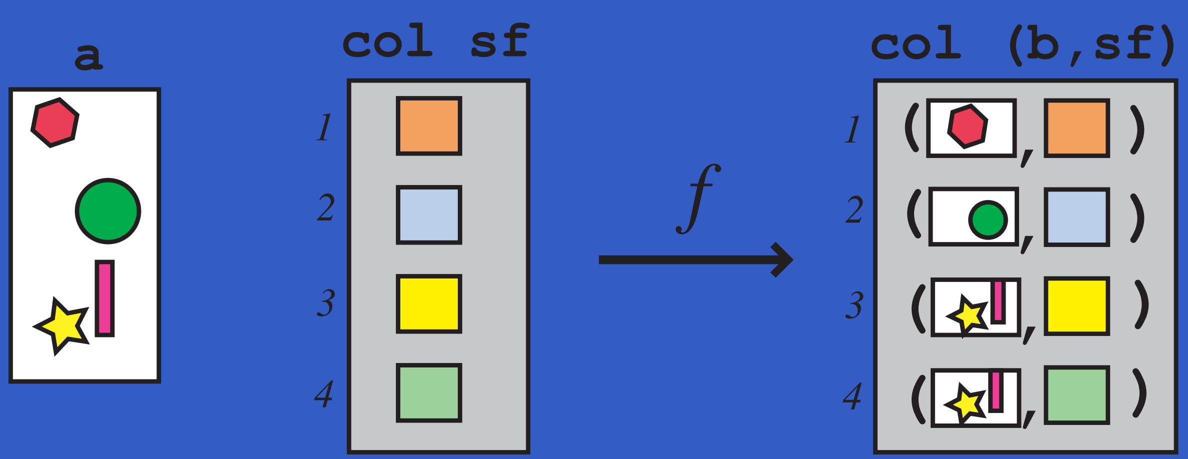
Idea:

- The routing function decides which parts of the input to pass to each running signal function instance.

Routing

Idea:

- The routing function decides which parts of the input to pass to each running signal function instance.
- It achieves this by pairing a projection of the input with each running instance:



The Routing Function Type

Universal quantification over the collection members:

$$\begin{aligned}&\mathit{Functor}\ \mathit{col} \Rightarrow\\&\quad(\mathit{forall}\ \mathit{sf} \circ (a \rightarrow \mathit{col}\ \mathit{sf} \rightarrow \mathit{col}\ (b, \mathit{sf})))\end{aligned}$$

Collection members thus ***opaque***:

- Ensures only signal function instances from argument can be returned.
- Unfortunately, does not prevent duplication or discarding of signal function instances.

Blocks

$$
\begin{array}{l}
objBlockAt\ (x, y)\ (w, h) = \\
\quad \mathbf{proc}\ (ObjectInput\ ci\ cs\ os) \rightarrow \mathbf{do} \\
\qquad \mathbf{let}\ name = \texttt{"blockat"} +\!\!+\ show\ (x, y) \\
\qquad\quad isHit \quad = inCollision\ name\ cs \\
\qquad hit \leftarrow edge \qquad\qquad\qquad\qquad \prec isHit \\
\qquad lives \leftarrow accumHoldBy\ (+)\ 3 \prec (hit\ `tag`\ (-1)) \\
\qquad \mathbf{let}\ isDead = lives \leqslant 0 \\
\qquad dead \leftarrow edge \prec isDead \\
\qquad returnA \prec ObjectOutput \\
\qquad\quad (Object\ \{\ldots\}) \\
\qquad\quad dead
\end{array}
$$

The Game Core

$$
\begin{array}{l}
\mathit{processMovement} :: \\
\quad [\,\mathit{ObjectSF}\,] \rightarrow \mathit{SF}\ \mathit{ObjectInput}\ (\mathit{IL}\ \mathit{ObjectOutput}) \\
\mathit{processMovement}\ \mathit{objs} = \\
\quad \mathit{dpSwitchB}\ \mathit{objs} \\
\qquad\qquad\qquad (\mathit{noEvent} \longrightarrow \mathit{arr}\ \mathit{suicidalSect}) \\
\qquad\qquad\qquad (\lambda \mathit{sfs}'\ f \rightarrow \mathit{processMovement}'\ (f\ \mathit{sfs}'))
\end{array}
$$

$$
\begin{array}{l}
\mathit{loopPre}\ ([\,], [\,], 0)\ \$ \\
\quad \mathit{adaptInput} \\
\quad \ggg \mathit{processMovement}\ \mathit{objs} \\
\quad \ggg (\mathit{arr}\ \mathit{elemsIL} \mathbin{\&\&\&} \mathit{detectCollisions})
\end{array}
$$

Recovering Blocks

$$
\begin{array}{l}
\mathit{objBlockAt}\ (x, y)\ (w, h) = \\
\quad \mathbf{proc}\ (\mathit{ObjectInput}\ \mathit{ci}\ \mathit{cs}\ \mathit{os}) \rightarrow \mathbf{do} \\
\qquad \mathbf{let}\ \mathit{name} = \texttt{"blockat"} +\!\!+\ \mathit{show}\ (x, y) \\
\qquad\quad \mathit{isHit} \ \ = \mathit{inCollision}\ \mathit{name}\ \mathit{cs} \\
\qquad \mathit{hit} \quad\ \ \leftarrow \mathit{edge} \qquad\qquad\quad \prec \mathit{isHit} \\
\qquad \mathit{recover} \leftarrow \mathit{delayEvent}\ 5.0 \prec \mathit{hit} \\
\qquad \mathit{lives} \quad \leftarrow \mathit{accumHoldBy}\ (+)\ 3 \\
\qquad\qquad\qquad \prec (\mathit{hit}\ `\mathit{tag}`\ (-1) \\
\qquad\qquad\qquad\qquad `\mathit{lMerge}`\ \mathit{recover}\ `\mathit{tag}`\ 1) \\
\qquad \ldots
\end{array}
$$

Reading (1)

- John Hughes. Generalising monads to arrows. *Science of Computer Programming*, 37:67–111, May 2000
- John Hughes. Programming with arrows. In *Advanced Functional Programming*, 2004. To be published by Springer Verlag.
- Henrik Nilsson, Antony Courtney, and John Peterson. Functional reactive programming, continued. In *Proceedings of the 2002 Haskell Workshop*, pp. 51–64, October 2002.

Reading (2)

- Antony Courtney and Henrik Nilsson and John Peterson. The Yampa Arcade. In *Proceedings of the 2003 Haskell Workshop*, pp. 7–18, August 2003.
- Ivan Perez and Henrik Nilsson. Bridging the GUI gap with reactive values and relations. In *Proceedings of the 8th ACM SIGPLAN Symposium on Haskell (Haskell'15)*, pages 47–58, 2015.