Advanced Programming

Final Lecture

Troels Henriksen

Real World Haskell

How to Live the Rest of Your Life

Course Evaluation

The Exam

Leaving the ivory tower

AP	▪ Simple models. ▪ Clear specifications. ▪ No performance concerns.
Real world	▪ Arbitrary complexity. ▪ Unclear requirements. ▪ Performance requirements.

The ivory tower still exists in the real world

- My research is on parallel programming, compiler optimisations, high performance computing, numerical computing, etc.
- **Applied computer science**—I read and write many programs.
  - Haskell is my *pragmatic* tool of choice.

The ivory tower still exists in the real world

- My research is on parallel programming, compiler optimisations, high performance computing, numerical computing, etc.
- **Applied computer science**—I read and write many programs.
  - Haskell is my *pragmatic* tool of choice.

- Main applied project is *Futhark*.
- Almost 100k lines of Haskell.
- Haskell is *not the point*—feels like any other program.

`https://futhark-lang.org`

Artwork by Robert Schenck.

So what is it like to use Haskell for real programs?

`String` considered harmful

```
type String = [Char]

"DIKU" == 'D' : 'I' : 'K' : 'U' : []
```

`String` considered harmful

```
type String = [Char]
```

```
"DIKU" == 'D' : 'I' : 'K' : 'U' : []
```

But consider the memory usage.

- Each `:` (*cons*) cell contains two slots: *value* and *successor*.
  - Each is a pointer to somewhere in memory.

`String` considered harmful

```
type String = [Char]
```

```
"DIKU" == 'D' : 'I' : 'K' : 'U' : []
```

But consider the memory usage.

- Each `:` (*cons*) cell contains two slots: *value* and *successor*.
  - Each is a pointer to somewhere in memory.
- A four-character string needs four cells, meaning eight pointers in total.
  - That is $8 \cdot 8 = 64$ bytes!
    - For a four-character string!
    - This is $16x$ overhead.
  - And in fact, each Haskell value also needs a *header* for garbage collection purposes, so things are even worse!

`String` considered harmful

```
type String = [Char]

"DIKU" == 'D' : 'I' : 'K' : 'U' : []
```

But consider the memory usage.

- Each `:` (*cons*) cell contains two slots: *value* and *successor*.
  - Each is a pointer to somewhere in memory.
- A four-character string needs four cells, meaning eight pointers in total.
  - That is $8 \cdot 8 = 64$ bytes!
    - For a four-character string!
    - This is 16*x* overhead.
  - And in fact, each Haskell value also needs a *header* for garbage collection purposes, so things are even worse!
- Also the characters are scattered all over memory $\Rightarrow$ terrible locality.

Real Haskell programs use `Data.Text.Text`

`https://hackage.haskell.org/package/text-2.1.1/docs/Data-Text.html`

- Built on top of `Data.ByteString`.

Haskell cheat code

```
unsafePerformIO ::  IO a -> a
```

- **Extremely Dangerous**, much more than you think.

Free Monads are Used In Real Programs!

But...

- The representation you saw in AP is inefficient.
  - Each use of »= has to completely rebuild the computation tree.
  - A better one looks like this:

```
newtype F f a = F (forall r.  (a -> r) -> (f r -> r) -> r)
```

Example of free monads in my own work

- Package resolution: `https://github.com/diku-dk/futhark/blob/master/src/Futhark/Pkg/Solve.hs`
- Interpreter with single stepping, breakpoints, and tracing: `https://github.com/diku-dk/futhark/blob/master/src/Language/Futhark/Interpreter.hs`

Alternatives to Free Monads

Free monads are not the only way to decouple the specification of effects from their interpretation.

- Another technique (perhaps more common) is the so-called *tagless final*.
- Idea is to describe effects with a *type class*.

Example based on Haskell's `mtl`

```
class Monad m => MonadState s m | m -> s where
  get :: m s
  put :: s -> m ()

class Monad m => MonadReader r m | m -> r where
  ask :: m r
```

Example based on Haskell's `mtl`

```
class Monad m => MonadState s m | m -> s where
  get :: m s
  put :: s -> m ()

class Monad m => MonadReader r m | m -> r where
  ask :: m r
```

The classes can then be implemented by different concrete monads.

```
newtype RS r s a = RS (r -> s -> (a,s))

instance MonadState s (RS r s) where
  get = RS $ \r s -> (s,s)
  put s = RS $ \r _ -> ((),s)

instance MonadReader r (RS r s) where
  ask = RS $ \r s -> (r,s)
```

Using the type classes

```
stateMax :: MonadState Int m => Int -> m ()
stateMax x = do
  cur <- get
  if cur > x then pure ()
             else put x
```

Real World Haskell

How to Live the Rest of Your Life

Course Evaluation

The Exam

If you liked AP

Unfortunately there is no *Really Advanced Programming* course at DIKU.

But there are courses that have the same spirit.

If you liked AP

Unfortunately there is no *Really Advanced Programming* course at DIKU.

But there are courses that have the same spirit.

- *Programming Massively Parallel Hardware* (PMPH), block 1, taught by Cosmin Oancea. GPU programming. Only course at DIKU where you are specifically graded on writing fast programs.

If you liked AP

Unfortunately there is no *Really Advanced Programming* course at DIKU.

But there are courses that have the same spirit.

- *Programming Massively Parallel Hardware* (PMPH), block 1, taught by Cosmin Oancea. GPU programming. Only course at DIKU where you are specifically graded on writing fast programs.
- *Data Parallel Programming* (DPP), block 2, taught by Cosmin Oancea and myself. High level functional programming and performance. Strongly based on local research.

If you liked AP

Unfortunately there is no *Really Advanced Programming* course at DIKU.

But there are courses that have the same spirit.

- *Programming Massively Parallel Hardware* (PMPH), block 1, taught by Cosmin Oancea. GPU programming. Only course at DIKU where you are specifically graded on writing fast programs.
- *Data Parallel Programming* (DPP), block 2, taught by Cosmin Oancea and myself. High level functional programming and performance. Strongly based on local research.
- *Semantics and Types* (SaT), block 3, taught by Andrzej Filinski. Programming language theory and semantics. **Theoretical.**

Also consider projects

I work in the *Programming Languages and Theory of Computation* section, and there are many researchers willing to supervise projects in functional programming and other PL topics.

- 30 ECTS master thesis.
- 7.5 ECTS thesis preparation project.
- 7.5 ECTS or 15 ECTS “project outside course scope”.

Potential supervisors:

- Troels Henriksen
- Martin Elsman
- Torben Mogensen
- Ken Friis Larsen
- Andrzej Filinski
- Cosmin Oancea
- The others: `https://di.ku.dk/english/research/pltc/`

Real World Haskell

How to Live the Rest of Your Life

Course Evaluation

The Exam

Preliminary context

- **The MSc at DIKU is supposed to be challenging.**
  - *Intended* to be the highest academic level of computer science in Denmark.

Preliminary context

- **The MSc at DIKU is supposed to be challenging.**
  - *Intended* to be the highest academic level of computer science in Denmark.
- **AP is supposed to be hard.**
  - Most ambitious *mandatory* programming course in Denmark (that I could find).

Preliminary context

- **The MSc at DIKU is supposed to be challenging.**
  - *Intended* to be the highest academic level of computer science in Denmark.
- **AP is supposed to be hard.**
  - Most ambitious *mandatory* programming course in Denmark (that I could find).
- **Suffering is not the same as learning.**
  - Students don't learn more when they are stressed.
  - Teachers don't benefit from tormenting students.

AP has been notorious for excessive workload

2023 course evaluation — hours spent by students per week

Min gennemsnitlige ugentlige arbejdsindsats på dette kursus har været på (inkl. undervisningstimer, forberedelse, skriftligt arbejde osv.):

Under 10 timer	0 / 66	0,0 %
10-15 timer	1 / 66	1,5 %
16-25 timer	19 / 66	28,8 %
26-30 timer	15 / 66	22,7 %
Over 30 timer	31 / 66	47,0 %
Ved ikke	0 / 66	0,0 %

2023 course evaluation — perceived student workload

Jeg oplever arbejdsbyrden på kurset som:

Alt for lille	0 / 66	0,0 %
For lille	1 / 66	1,5 %
Tilpas	10 / 66	15,2 %
For stor	16 / 66	24,2 %
Alt for stor	39 / 66	59,1 %

What we tried to do

What we tried to do

1. *Streamline* the course to minimise how much time is spent on nonproductive activities.
   - E.g. less analysis of requirements.
   - Got rid of Erlang for concurrency.

What we tried to do

1. *Streamline* the course to minimise how much time is spent on nonproductive activities.
   - E.g. less analysis of requirements.
   - Got rid of Erlang for concurrency.
2. Strong *cohesion* between course elements.
   - The notes/lecture $\Rightarrow$ exercise $\Rightarrow$ assignment progression.

What we tried to do

1. *Streamline* the course to minimise how much time is spent on nonproductive activities.
   - E.g. less analysis of requirements.
   - Got rid of Erlang for concurrency.
2. Strong *cohesion* between course elements.
   - The notes/lecture $\Rightarrow$ exercise $\Rightarrow$ assignment progression.
3. Very applied lectures.
   - Inspired by my experiences in other courses.
   - Perhaps we took it too far this year.

What we tried to do

1. *Streamline* the course to minimise how much time is spent on nonproductive activities.
   - E.g. less analysis of requirements.
   - Got rid of Erlang for concurrency.
2. Strong *cohesion* between course elements.
   - The notes/lecture $\Rightarrow$ exercise $\Rightarrow$ assignment progression.
3. Very applied lectures.
   - Inspired by my experiences in other courses.
   - Perhaps we took it too far this year.
4. Made assignments a bit smaller.
   - But not as much as you'd think.

Overall, tried to ensure that weaker students had a good chance to pass by putting in a reasonable amount of effort (20-25 hours per week).

Notes I wrote to guide my work:
`https://sigkill.dk/writings/teaching.html`

Looking at graphs

Selected comments

- Some of these have been rephrased, paraphrased, or translated from Danish.
- I consider them all *constructive* and *useful*, even if I do not always agree.

It was really good that you got the next submission a few days before the deadline for the previous one. It gave a good flow and you had a lot of flexibility with the working hours. It should be a requirement for all courses to do it this way.

The exercises was a great tool for the assignmentss, but it was a little confusing when the assignments was supposed to be made.

It would be nice if you could just 'collect points' in the mandatory assignments, instead of having to get at least one point in each assignment. Especially having an assignment that has to be handed in the day before the exam seems a bit pressured.

Assignment A4 was of lower quality than the others because the intended solution went against the rules we had otherwise been given about which parts of the code we were allowed to change.

Some of the submissions were a bit unclear about where you could find things sometimes, and where you could/couldn't change the files, but overall it wasn't a big enough problem to be disruptive.

It would be better to clarify what is and is not allowed to be changed in the handout code for the assignment. It was only after a formal clarification in the middle of the course that it was made obvious. This led to some frustration with solving tasks.

The exercise classes are very late in terms of getting relevant help for the submissions.
More exercise classes, and also spread out during the week in case of questions before Thursday

Live coding (with Troels) was really useful.

Also worth mentioning is that this is the first time I've witnessed really good live-coding sessions!

live coding gave a lot

The live coding was nice to get a feel and understanding of the Haskell code.

Live coding worked extremely well.

More live-coding!

But...

I personally didn't get much out of the live coding lectures.

I can confirm. livecoding doesn't work for me.

Personally, I'm not a fan of the live coding format.

I wasn't that happy with the lectures which were just looking at code without using slides

The lectures sometimes felt redundant once you had read the course notes

It is not the task of the University to offer what society asks for, but to give what society needs.

— Edsger Dijkstra, EWD1305 (2000)

Real World Haskell

How to Live the Rest of Your Life

Course Evaluation

The Exam

The exam is about simulating game strategies

Prisoner's Dilemma

Two persons are in solitary confinement and cannot communicate. They are asked to testify against the other.

- If both testify, then both go to jail for 2 years each.
- If one testifies and the other does not, the former goes free and the latter goes to jail for 3 years.
- If none of them testify, both go to jail for 1 year.

From a game-theoretical standpoint, the optimal strategy is to not testify.

This changes if the game is played over multiple rounds and the players have memory of past interactions.

Payoff matrix

a / *b*	*Testify*	*Don't*
Testify	2 / 2	0 / 3
Don't	3 / 0	1 / 1

- Players a, b repeatedly make simultaneous choices whether to testify, and have their score adjusted as by the payoff matrix.
- (In this case the goal is to minimise score, but in other games it may be to maximise it.)

Payoff matrix

a / *b*	*Testify*	*Don't*
Testify	2 / 2	0 / 3
Don't	3 / 0	1 / 1

- Players *a*, *b* repeatedly make simultaneous choices whether to testify, and have their score adjusted as by the payoff matrix.
- (In this case the goal is to minimise score, but in other games it may be to maximise it.)

Now consider if we have multiple prisoners, interacting with randomly chosen other prisoners, over multiple rounds, with memory, and also the prisoners can die and reproduce into empty cells… what is the optimal strategy for when to testify and when not to?

(Perhaps we need a better analogy.)

It is time to touch grass

This is a Galápagos finch.

Photo by Peter Wilton
https://commons.wikimedia.org/wiki/File:Large_ground_finch_(4229035966).jpg

The finch predicament

- Finches are afflicted by **parasitic mites**.
- A finch cannot clean itself of mites, **but can expend energy helping groom another finch**.
- When a finch **meets** another finch, it can decide to **groom** or **ignore** it.
- A finch has some number of **health points** (HP). The clean/ignore choice gives rise to a cost matrix, which might look like this:

Finch *a* / Finch *b*	*Groom*	*Ignore*
Groom	1 / 1	−4 / 3
Ignore	3 / −4	−2 / −2

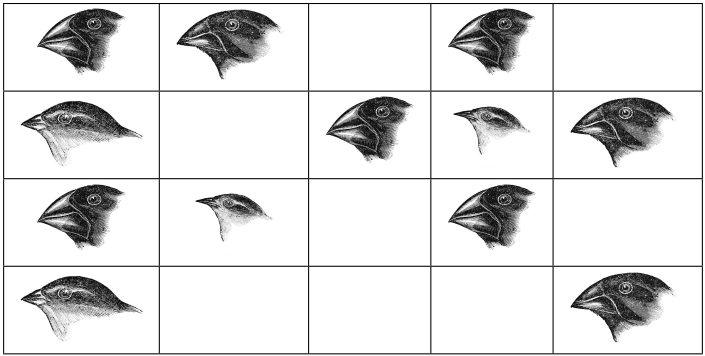
The outcome is added to the HP of each finch.

Further fascinating insights into the life of Galápagos finches

- The Galápagos islands are a rectangular grid of cells joined at the edges, like a torus or donut.
- A finch can meet only its **immediate neighbours** (and a finch never moves).
  - Similar to the model in the Week 7 exercises.
- A finch has a **lifespan**, denoting how many rounds it has left to live.
- When a finch reaches 0 HP or 0 lifespan, it is **dead** and removed.
- When an empty cell is next to a finch, it has a **chance** of reproducing into that cell.
- Different **species** of finches exist, each of which follow a **strategy** for when to groom/ignore anothyer finch.

The exam problem is thus about constructing a simulation of various finch strategies competing against each other, and seeing which ones work best.

A grid of finches



- Many ways to evaluate meetings between finches (or with empty cells).
- You will use a **block automaton using Margolus neighbourhoods**.
  - See Week 7 exercises.
- You will implement this as a **concurrent simulation**, where the amount of concurrency must be proportional to the grid size.

The expression of finch strategies

Strategies are written in a slightly extended version of APL.

Samaritan

```
loop x = 0 for i < 100 do
let unused = meet in groom
```

Cheater

```
loop x = 0 for i < 100 do
let unused = meet in ignore
```

Flipflop

```
loop x = 0 for i < 100 do
let unused = meet
in if i / 2 * 2 == i
   then ignore else groom
```

The expression of finch strategies

Strategies are written in a slightly extended version of APL.

Samaritan

```
loop x = 0 for i < 100 do
let unused = meet in groom
```

Cheater

```
loop x = 0 for i < 100 do
let unused = meet in ignore
```

Flipflop

```
loop x = 0 for i < 100 do
let unused = meet
in if i / 2 * 2 == i
   then ignore else groom
```

Grudger

```
loop jerks = {} for i < 100 do
  let other = meet in
  let outcome =
    if other elem jerks
    then ignore else groom
  in if outcome then jerks
        else jerks union {other}
```

Exam tasks, overall

- Extend APL with set operations.
- Implement concurrent simulation of block-cellular automatons.
- Implement Galápagos finch logic, which involves interpreting the strategy programs.

…factored into multiple parts for ease of correction, and with a few extra bits.

More exam advice

Most important rule

Do not share your solution.

- This includes putting it in public Git repositories.
- This is a disciplinary infraction and you will be suspended from the university.

More exam advice

Most important rule

Do not share your solution.

- This includes putting it in public Git repositories.
- This is a disciplinary infraction and you will be suspended from the university.

- Page limit is long, this does not mean you are expected to fill it. **Short is good.**
- Don’t waste yours and our time with generic ChatGPT-answers to questions — you will receive no credit.
- Take breaks.
- You can pass without solving every task.
- The above information is non-normative.

See you next year

Questions?