Type Driven Development in Idris

Edwin Brady (ecb10@st-andrews.ac.uk)
University of St Andrews, Scotland, UK
@edwinbrady

Kats Workshop, Dublin, 21st May 2016
Welcome to Fife
Idris is a *Pac-man Complete* functional programming language with *dependent types*

- Still undergoing lots of *research*...
Idris is a *Pac-man Complete* functional programming language with *dependent types*

- Still undergoing lots of *research*...
- ...but lots of fun to play (and learn) with!
Idris

Idris is a Pac-man Complete functional programming language with dependent types

- http://idris-lang.org/
- Still undergoing lots of research...
- ...but lots of fun to play (and learn) with!

Workshop materials (code, slides, exercises):

Idris

**Idris** is a *Pac-man Complete* functional programming language with *dependent types*

- Still undergoing lots of *research*...
- ...but lots of fun to play (and learn) with!

Workshop materials (code, slides, exercises):

- Also, I have stickers!
Workshop outline:

- Part 1 (*Fundamentals*): First class types, vectors, defining data types
- Part 2 (*Relationships between data*): equality proofs, expressing assumptions in types
- Part 3 (*Effects*): IO, Side effects, reasoning about state
- Part 4 (*Total Functional Programming*): views, streams, processes and type safe concurrency
Workshop outline:

- Part 1 (*Fundamentals*): First class types, vectors, defining data types
- Part 2 (*Relationships between data*): equality proofs, expressing assumptions in types
- Part 3 (*Effects*): IO, Side effects, reasoning about state
- Part 4 (*Total Functional Programming*): views, streams, processes and type safe concurrency

There are exercises for the breaks!
https://www.manning.com/books/type-driven-development-with-idris

(Ask me about discount codes)
Why types?

We can use **type systems** for:

- **Checking** a program has the intended properties
- **Guiding** a programmer towards a correct program
- Building **expressive** and **generic** libraries
Why types?

We can use *type systems* for:

- **Checking** a program has the intended properties
- **Guiding** a programmer towards a correct program
- Building *expressive* and *generic* libraries

*Type Driven Development* puts *types* first. Three steps:

- **Type**: Write a type for a function
- **Define**: Create a (possibly incomplete) implementation
- **Refine**: Improve/complete the implementation
Idris

Time to write some code!

- Type directed editing
- First-class types, vectors, defining data types
- Proofs and relationships between data (after the break)
So far, the programs we’ve written have been *pure*

- But... real programs need to interact with the outside world!
- Two approaches:
  - The **IO** type (the **Haskell** approach)
  - The **Eff** dependent type (describe allowed side effects precisely)
The IO Type

IO is an abstract type which allows us to describe interactive programs

data IO : Type -> Type where ...
**IO** is an abstract type which allows us to *describe* interactive programs

```ml
data IO : Type -> Type where ...
```

Then we can *execute* interactive programs using the :exec command. For example:

```
Idris> :exec putStrLn (show (47 * 2))
94
```
The IO Type

putStrLn (show (47 * 2)) ← original expression

Evaluator

Write "94\n" to console ← description of interactions

Run Time Environment
Coding time: IO and Effects
State machine example: Door opening

State Machine Diagram:
- **CLOSED**
- **OPEN**

Transitions:
- Knock → CLOSED
- Open Door → OPEN
- Close Door → CLOSED
Coding time: Door Protocol and Effects
A total function is a function which is defined for all possible well-typed inputs. That is:

- It covers all possible well-typed input cases
- For all well-typed inputs it either
  - terminates with a well-typed result
  - produces a new constructor of an infinite data type
A *total* function is a function which is defined for all possible well-typed inputs. That is:

- It *covers* all possible well-typed input cases
- For all well-typed inputs it either
  - *terminates* with a well-typed result
  - *produces* a new constructor of an infinite data type

Idris *always* checks whether a function is guaranteed to be total.

- This is conservative, due to the halting problem...
Why does totality matter?

- **Correctness**: “Well-typed programs don’t go wrong”
- **Efficiency**: optimiser can be more aggressive when functions are guaranteed to terminate
- **Precision**: related to correctness
Why does totality matter?

- **Correctness**: “Well-typed programs don’t go wrong”
- **Efficiency**: optimiser can be more aggressive when functions are guaranteed to terminate
- **Precision**: related to correctness
A function may be total, but why does that help if we don’t know how long it runs for?

- We can still guard against *accidental* non-termination
  - Assertion: *accidental* non-termination is *always* a bug
- A program without a termination proof is *highly suspicious*
Coding time: Streams and Views
Why are we interested in dependent types?

- **Safety**
  - Programs checked against precise specifications

- **Expressivity**
  - Better, more descriptive APIs
  - *Type directed* development
  - Type system should be *helping*, not telling you off!

- **Genericity**
  - e.g. program generation

- **Efficiency**
  - More precise type information should help the compiler
  - *Partial evaluation*, *erasure*
https://www.manning.com/books/type-driven-development-with-idris

(Ask me about discount codes!)