

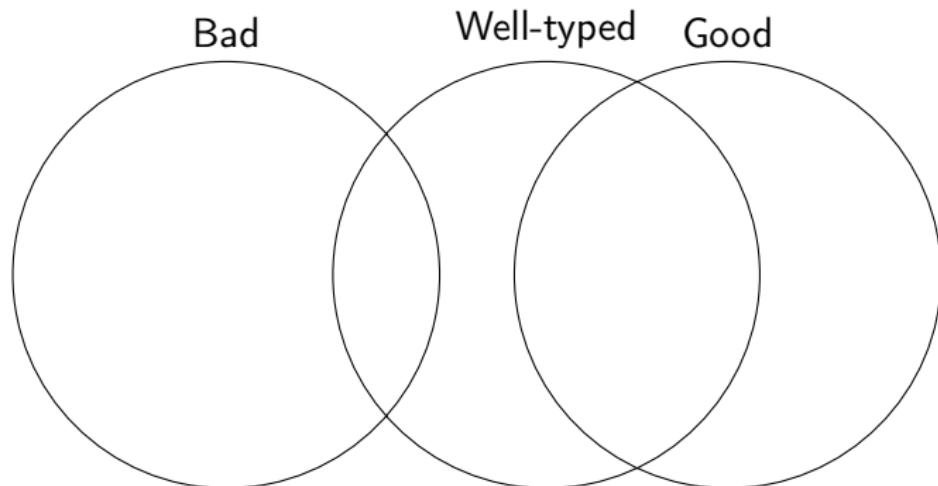
# Dependently Typed Programming in Idris

## A Demo

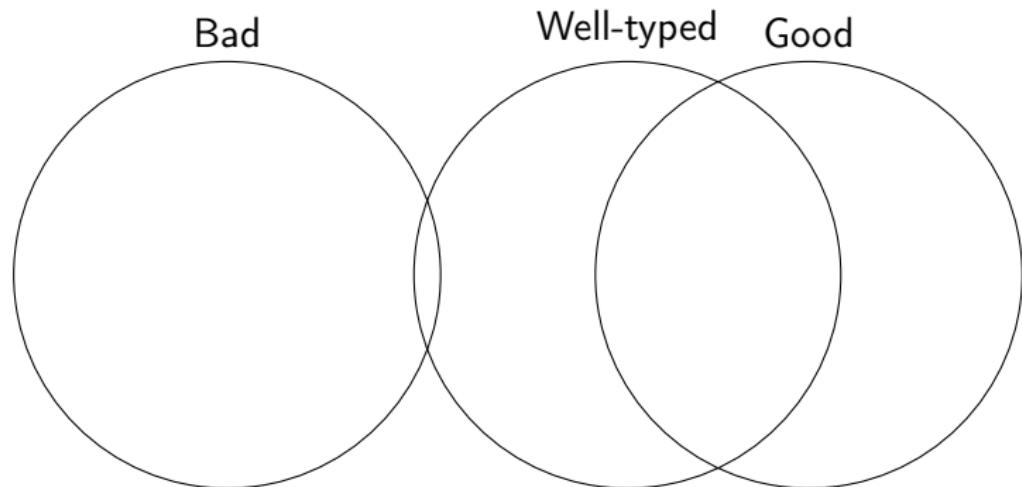
David Raymond Christiansen

March 26, 2014

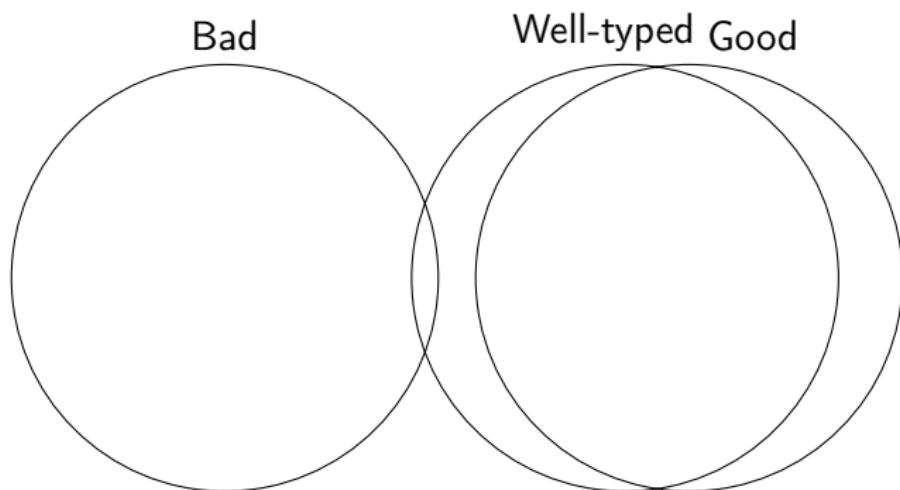
# Types in programming



## Stronger types, safer code



Write more programs



# Type Computation is Computation

- ▶ Non-trivial invariants require non-trivial types

## Type Computation is Computation

- ▶ Non-trivial invariants require non-trivial types
- ▶ Type systems achieve greater and greater expressiveness

## Type Computation is Computation

- ▶ Non-trivial invariants require non-trivial types
- ▶ Type systems achieve greater and greater expressiveness
- ▶ More and more, we have *interesting computation* in the type system

## Type Computation is Computation

- ▶ Non-trivial invariants require non-trivial types
- ▶ Type systems achieve greater and greater expressiveness
- ▶ More and more, we have *interesting computation* in the type system
- ▶ Scala's type system is Turing-complete!

## Type Computation is Computation

- ▶ Non-trivial invariants require non-trivial types
- ▶ Type systems achieve greater and greater expressiveness
- ▶ More and more, we have *interesting computation* in the type system
- ▶ Scala's type system is Turing-complete!
- ▶ How do these non-trivial types look?

## Scala type computation

### Scala type computation

- ▶ Anonymous type functions:

```
({type F[A] = Either[A, String]})#F
```

# Scala type computation

## Scala type computation

- ▶ Anonymous type functions:

```
({type F[A] = Either[A, String]})#F
```

- ▶ Case distinction performed with type members:

```
sealed trait Nat { type Plus[N<:Nat] <: Nat }
trait Zero extends Nat {
  type Plus[N <: Nat] = N
}
trait Succ[N <: Nat] extends Nat {
  type Plus[M <: Nat] = Succ[N#Plus[M]]
}
```

# Scala type computation

## Scala type computation

- ▶ Anonymous type functions:

```
({type F[A] = Either[A, String]})#F
```

- ▶ Case distinction performed with type members:

```
sealed trait Nat { type Plus[N<:Nat] <: Nat }
trait Zero extends Nat {
  type Plus[N <: Nat] = N
}
trait Succ[N <: Nat] extends Nat {
  type Plus[M <: Nat] = Succ[N#Plus[M]]
}
```

- ▶ We can even define a general type-level fold!

# Scala type computation

Scala

- ▶ A
- ▶ C
- ▶ C
- ▶ s
- ▶ t
- ▶ }
- ▶ t
- ▶ }
- ▶ }
- ▶ V



## Various Haskell extensions

- ▶ Generalized algebraic datatypes



## -XKitchenSink

### Various Haskell extensions

- ▶ Generalized algebraic datatypes
- ▶ Type families



## Various Haskell extensions

- ▶ Generalized algebraic datatypes
- ▶ Type families
- ▶ Data kinds



## Various Haskell extensions

- ▶ Generalized algebraic datatypes
- ▶ Type families
- ▶ Data kinds
- ▶ Multi-parameter type classes with functional dependencies and undecidable instances

## Various Haskell extensions

- ▶ Generalized algebraic datatypes
- ▶ Type families
- ▶ Data kinds
- ▶ Multi-parameter type classes with functional dependencies and undecidable instances
- ▶ Polymorphic kinds

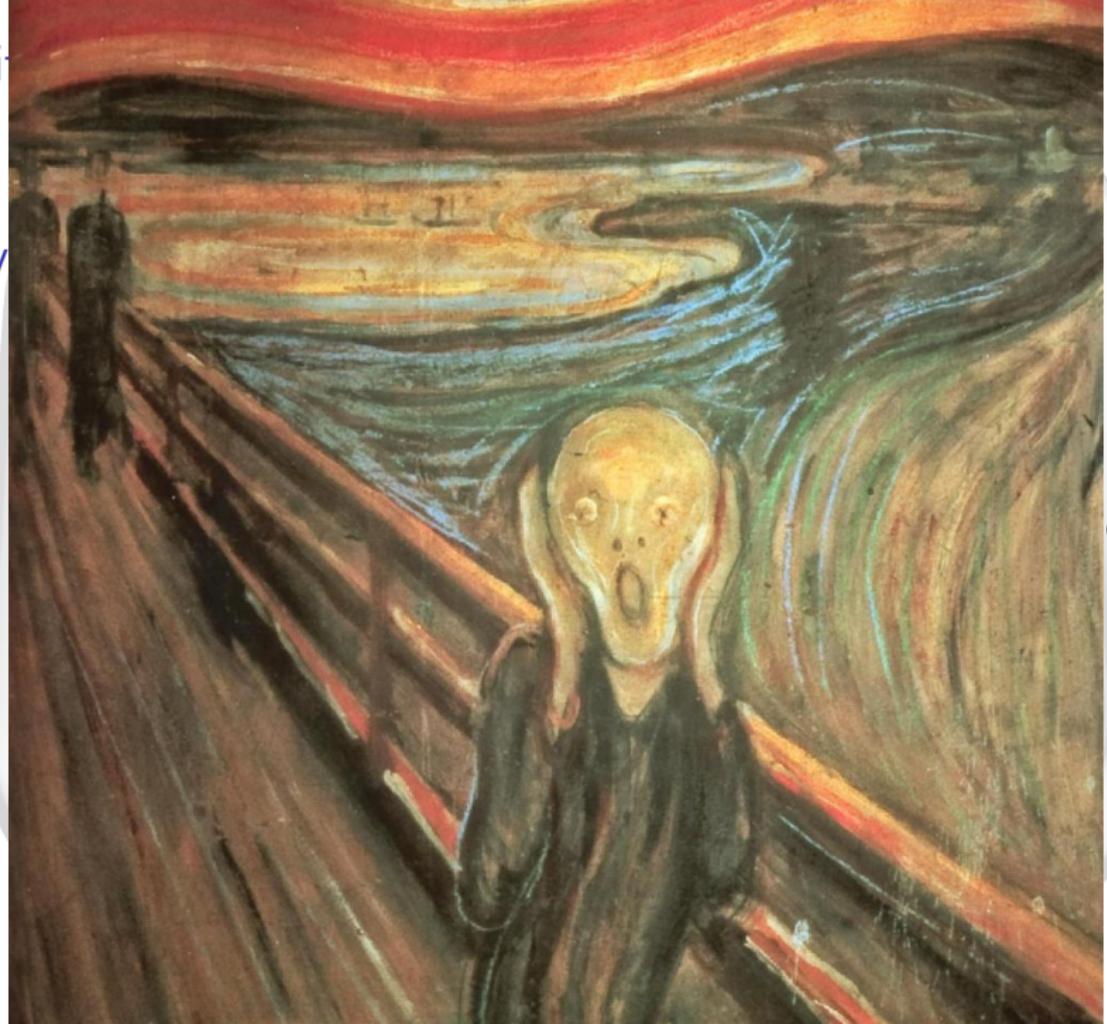
## Various Haskell extensions

- ▶ Generalized algebraic datatypes
- ▶ Type families
- ▶ Data kinds
- ▶ Multi-parameter type classes with functional dependencies and undecidable instances
- ▶ Polymorphic kinds
- ▶ ...

-XKit

V

d



# Why two languages?

## Phase distinction

Compile-time computation happens in types and type checking

Run-time computation happens with terms

# Why two languages?

## Phase distinction

Compile-time computation happens in types and type checking

Run-time computation happens with terms

## Totality and decidability

- ▶ We want our compiler to terminate, but not necessarily our programs

## Why two languages?

### Phase distinction

Compile-time computation happens in types and type checking

Run-time computation happens with terms

### Totality and decidability

- ▶ We want our compiler to terminate, but not necessarily our programs

### Erasure

Types should disappear at run-time, creating no overhead

## Dependent types

- ▶ One language at compile-time and run-time

## Dependent types

- ▶ One language at compile-time and run-time
- ▶ One language for both types and terms

## Dependent types

- ▶ One language at compile-time and run-time
- ▶ One language for both types and terms
- ▶ Types can contain values, functions can return types

# Overview

Dependent Types

Dependent Pattern Matching  
The with rule

AVL Trees

Universes

Type Providers

Not Covered Today

Resources

# Idris

- ▶ Pure functional programming language
- ▶ Full dependent types
- ▶ Compiles to C, LLVM, Java, Javascript (browser and Node)
- ▶ Syntax inspired by Haskell
- ▶ Strict evaluation order
- ▶ Extensible syntax
- ▶ Integrated theorem proving
- ▶ Free software - everything is on Github

## Your presenter

- ▶ PhD student at ITU with Peter Sestoft
- ▶ Currently visiting Chalmers in Gothenburg
- ▶ Interests include dependently typed programming, DSLs, and programming tools more generally
- ▶ Working on the Actulus project, making a DSL for actuaries
- ▶ Contributor to Idris, esp. type providers + usability work
- ▶ Grew up in Moscow, Idaho, USA, but lived in Copenhagen since 2005
- ▶ <http://www.itu.dk/people/drc>

## Demonstration

Idris basics: Hello, world!, natural numbers and lists; vectors; first dependent types

## Dependent Pattern Matching

- ▶ Pattern matching one argument can affect other arguments (through unification)

## Dependent Pattern Matching

- ▶ Pattern matching one argument can affect other arguments (through unification)
- ▶ Some patterns can be ruled out

## Dependent Pattern Matching

- ▶ Pattern matching one argument can affect other arguments (through unification)
- ▶ Some patterns can be ruled out by others
- ▶ The form of one pattern can be fixed by another pattern

## Dependent Pattern Matching

- ▶ Pattern matching one argument can affect other arguments (through unification)
- ▶ Some patterns can be ruled out by others
- ▶ The form of one pattern can be fixed by another pattern
- ▶ We learn something about related arguments by matching one of them

## Dependent Pattern Matching

- ▶ Pattern matching one argument can affect other arguments (through unification)
- ▶ Some patterns can be rule each other out
- ▶ The form of one pattern can be fixed by another pattern
- ▶ We learn something about related arguments by matching one of them

Demo: zip reconsidered; take and drop on vectors

## The with rule

- ▶ Sometimes recursive calls should affect our arguments
- ▶ The with rule lets us express this instead of using case
- ▶ This can implement custom views

## Demonstration

Snoc lists as a view of regular lists

## AVL Trees

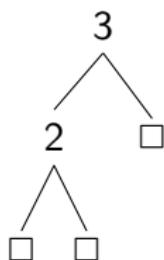
AVL trees are balanced binary trees where the heights of the child branches differ by at most 1

## AVL Trees

AVL trees are balanced binary trees where the heights of the child branches differ by at most 1

Tree rotations are used to recover the invariant when inserting

Insert 1

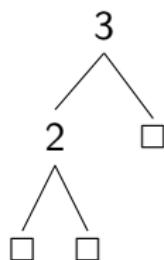


## AVL Trees

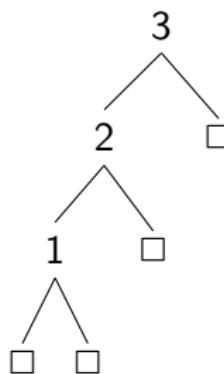
AVL trees are balanced binary trees where the heights of the child branches differ by at most 1

Tree rotations are used to recover the invariant when inserting

Insert 1



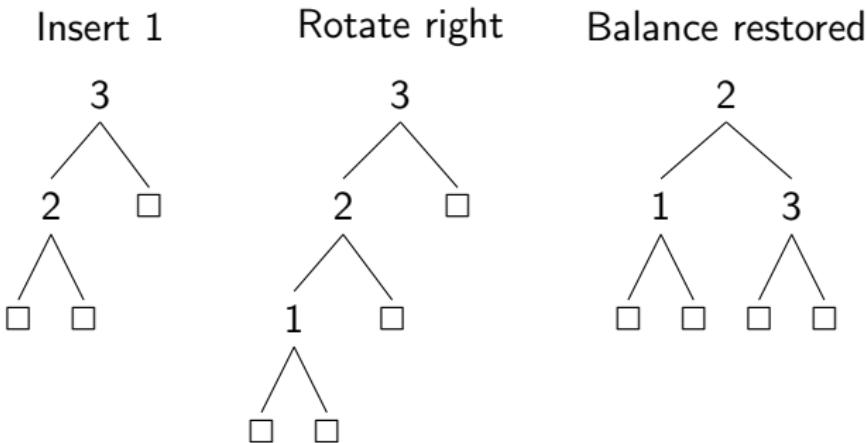
Rotate right



## AVL Trees

AVL trees are balanced binary trees where the heights of the child branches differ by at most 1

Tree rotations are used to recover the invariant when inserting



Demonstration: AVL trees with static balance invariant

# Universes

A universe is:

- ▶ A datatype whose elements describe a collection of types (*codes*)
- ▶ A function that interprets the codes as real types
- ▶ Examples: types that are serializable by some algorithm

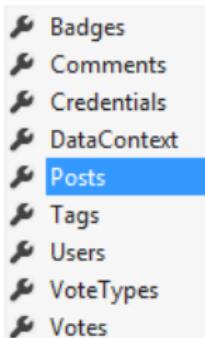
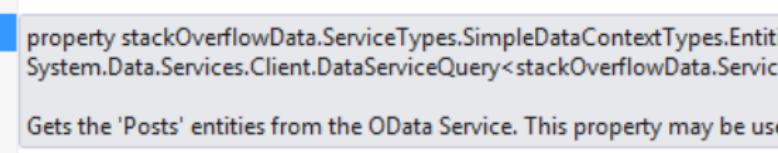
Demo: Universes.idr

## Well-Typed Interpreter

Using a universe of simple types, get Idris to show the type-correctness of an interpreter for the lambda calculus.

# F# type providers

```
module SO =
    type stackOverflowData =
        ODataService< @"http://data.stackexchange.com/stackoverflow/atom">
    let context = stackOverflowData.GetDataContext()

    context.
        
        Posts
        
        Gets the 'Posts' entities from the OData Service. This property may be used to query the posts in the service.
    
```

```
let tpQuestionCount = query {
    for post in context.Posts do
    where (post.Tags.Contains("f#") && post.Body.Contains("type provider"))
```



## Limitations

- ▶ Breaks the abstractions of F#

## Limitations

- ▶ Breaks the abstractions of F#
- ▶ Ill-typed terms and malformed types can result

## Limitations

- ▶ Breaks the abstractions of F#
- ▶ Ill-typed terms and malformed types can result
- ▶ Type providers must be developed in separate modules

## Idris Type Providers

- ▶ Types are terms like any other

## Idris Type Providers

- ▶ Types are terms like any other
- ▶ We have a means of producing terms from I/O: **execution**.

## Simple Example

```
fromFile : String -> IO (Provider Type)
```

## Simple Example

```
fromFile : String -> IO (Provider Type)
fromFile fname =
  do str <- readFile fname
    let type = if (trim str) == "Int"
               then Int else Nat
    return (Provide type)
```

## Simple Example

```
fromFile : String -> IO (Provider Type)
fromFile fname =
  do str <- readFile fname
    let type = if (trim str) == "Int"
               then Int else Nat
    return (Provide type)

%provide (T1 : Type) with fromFile "theType"
foo : T1
foo = 3
```

## Errors

```
data Provider a = Error String  
                | Provide a
```

## Errors

```
confirmAge : IO Bool
confirmAge =
  do putStrLn "How old are you?"
     input <- getLine
     let age = parseInt (trim input)
     case age of
       Nothing => do putStrLn "Didn't understand"
                      confirmAge
       Just x => return (x >= 18)
```

## Keep off my lawn!

```
adultsOnly : IO (Provider Bool)
adultsOnly =
  do oldEnough <- confirmAge
    if oldEnough
      then do putStrLn "ok"
              return (Provide True)
      else return (Error ("Only adults may " ++
                           "compile this program")))
%provide (ok : Bool) with adultsOnly
```

# Demo

## How does it work?

```
data SQLiteType = TEXT | INTEGER | REAL
                | NULLABLE SQLiteType

interpSql : SQLiteType -> Type
interpSql TEXT = String
interpSql INTEGER = Integer
interpSql REAL = Float
interpSql (NULLABLE x) = Maybe (interpSql x)
```

## How does it work?

```
data Attribute = (::::) String SQLiteType
```

## How does it work?

```
data Attribute = (:::) String SQLiteType  
  
data Schema = Nil | (:) Attribute Schema
```

## How does it work?

```
data Attribute = (:::) String SQLiteType

data Schema = Nil | (:) Attribute Schema

data Row : Schema -> Type where
  Nil : Row []
  (:) : (interpSql t) -> Row s -> Row ((col:::t) :: s)
```

## How does it work?

```
data DB : String -> Type where
  MkDB : (dbFile : String) ->
    (dbTables : List (String, Schema)) ->
    DB dbFile
```

## How does it work?

```
data Query : DB f -> Schema -> Type where
  Select : {db : DB f} -> Tables db s ->
    Expr s INTEGER ->
    (s' : Schema) ->
    {ok : SubSchema s' s} ->
    {auto solveIt : decSubSchema s' s = Yes ok} ->
    Query db s'
```

## How does it work?

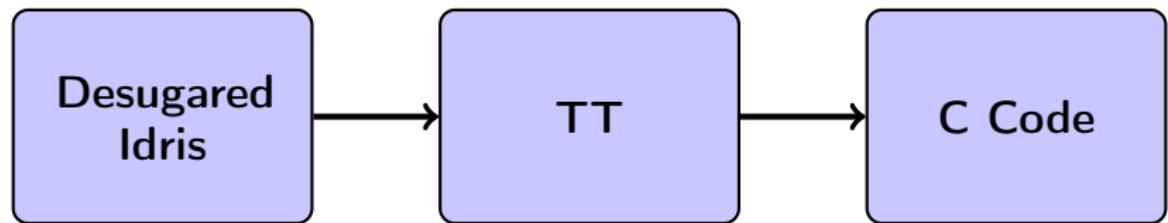
```
data Query : DB f -> Schema -> Type where
  Select : {db : DB f} -> Tables db s ->
    Expr s INTEGER ->
    (s' : Schema) ->
    {ok : SubSchema s' s} ->
    {auto solveIt : decSubSchema s' s = Yes ok} ->
    Query db s'

syntax SELECT [schema] FROM [tables] WHERE [expr] =
  Select tables expr schema
```

## The Type Provider

- ▶ Query the DB for the schemas
- ▶ Parse the DDL
- ▶ Return a **DB**

## Elaboration



# Elaborating Type Providers

To elaborate

**%provide** (**x** : **T**) **with** *p*

# Elaborating Type Providers

To elaborate

**%provide** (**x** : **T**) **with** *p*

1. Elaborate goal type T to  $\tau$ . Check that  $\tau : \text{Type}$ .

# Elaborating Type Providers

To elaborate

**%provide** (**x** : **T**) **with** *p*

1. Elaborate goal type T to  $\tau$ . Check that  $\tau : \text{Type}$ .
2. Elaborate *p* to  $\pi$ . Check that  $\pi : \text{IO}(\text{Provider } \tau)$ .

# Elaborating Type Providers

To elaborate

**%provide** (**x** : **T**) **with** *p*

1. Elaborate goal type T to  $\tau$ . Check that  $\tau : \text{Type}$ .
2. Elaborate *p* to  $\pi$ . Check that  $\pi : \text{IO}(\text{Provider } \tau)$ .
3. Execute  $\pi$ :

# Elaborating Type Providers

To elaborate

**%provide** (**x** : **T**) **with** *p*

1. Elaborate goal type T to  $\tau$ . Check that  $\tau : \text{Type}$ .
2. Elaborate *p* to  $\pi$ . Check that  $\pi : \text{IO}(\text{Provider } \tau)$ .
3. Execute  $\pi$ :
  - ▶ If Provide *y*, then result is  $x : \tau ; x = y$ .

# Elaborating Type Providers

To elaborate

**%provide** (**x** : **T**) **with** *p*

1. Elaborate goal type T to  $\tau$ . Check that  $\tau : \text{Type}$ .
2. Elaborate *p* to  $\pi$ . Check that  $\pi : \text{IO}(\text{Provider } \tau)$ .
3. Execute  $\pi$ :
  - ▶ If Provide *y*, then result is  $x : \tau ; x = y$ .
  - ▶ If Error *err*, then show user *err*.

# Elaborating Type Providers

To elaborate

**%provide** (**x** : **T**) **with** *p*

1. Elaborate goal type T to  $\tau$ . Check that  $\tau : \text{Type}$ .
2. Elaborate *p* to  $\pi$ . Check that  $\pi : \text{IO}(\text{Provider } \tau)$ .
3. Execute  $\pi$ :
  - ▶ If **Provide** *y*, then result is  $x : \tau ; x = y$ .
  - ▶ If **Error** *err*, then show user *err*.
  - ▶ If something else, then show user a generic error.

## Executor

- ▶ Interprets IO actions
- ▶ Separate from type checker's evaluator
- ▶ Must be careful about strictness

## Advantages

- ▶ Maintain Idris's abstractions and safety properties
- ▶ Develop type providers as ordinary programs
- ▶ Define and use type providers in same module

## Disadvantages

- ▶ Cannot introduce identifiers — inconvenient!
- ▶ No laziness
- ▶ Record types must be “faked”
- ▶ Terrible error messages

## Next Steps

- ▶ Lazy type providers

## Next Steps

- ▶ Lazy type providers
- ▶ Foreign function interface generation

## Next Steps

- ▶ Lazy type providers
- ▶ Foreign function interface generation
- ▶ Alternate execution times

## More on Idris Type Providers

- ▶ David Raymond Christiansen. *Dependent Type Providers*. WGP '13.
- ▶ David Raymond Christiansen. *Looking Outward: When Dependent Types Meet I/O*. M.Sc. thesis, IT University of Copenhagen, 2013.



## Not Covered (but feel free to ask))

- ▶ Type classes
- ▶ Effects
- ▶ Idiom brackets and bang bindings
- ▶ Proof automation
- ▶ Optional laziness
- ▶ Codata and corecursion

## Resources

- ▶ <http://www.idris-lang.org/>
- ▶ <http://groups.google.com/group/idris-lang>
- ▶ IRC: #idris on freenode
- ▶ <https://github.com/idris-lang/Idris-dev/wiki>
  - ▶ In-progress manual
  - ▶ Installation help
- ▶ <https://github.com/idris-hackers/>
  - ▶ Editor modes
  - ▶ Miscellaneous libraries
  - ▶ Other utilities such as IRC bots

# Get Involved!

Good projects for beginners:

- ▶ Documentation
- ▶ Port your favorite library
- ▶ Polish
- ▶ Document the pain points experienced by new users
- ▶ Improve editor and tool support

# Hackathon!

There will be an Idris developers' meeting in Gothenburg next month.

Tuesday, 29 April Introductions to Idris for Haskell and Agda programmers

Wednesday-Friday Talks, projects, and productivity

Sign up by mailing [drc@itu.dk](mailto:drc@itu.dk), details on wiki on Github