IDRIS, a general-purpose dependently typed programming language: Design and implementation – Edwin Brady

Daniela Ferreiro, Jorge Blázquez, Daniel Trujillo, Rafael Fernández

January 18, 2022

Table of Contents

Introduction

- Idris is a general purpose functional programming language
- Influenced by Haskell
  - Especially in the part of syntax and types
- Has full dependent types
  - No restriction on which values may appear in types
  - Allow a programmer to give a program more precise type

Dependent Types - Examples

Lists

```
data List : Type -> Type where
         Nil   : List a
         (::) : a -> List a -> List a
```

Dependent Types - Examples

Lists

```
data List : Type -> Type where
        Nil   : List a
        (::) : a -> List a -> List a
```

Vectors - Lists with length

```
data Vect : Nat -> Type -> Type where
        Nil   : Vect Z a
        (::) : a -> Vect k a -> Vect (S k) a
```

Dependent Types - Examples

takeList

```
takeList : (n : Nat) -> List a -> List a
takeList Z     list       = []
takeList (S k) []         = []
takeList (S k) (x :: xs) = x :: takeList k xs
```

Dependent Types - Examples

takeList

```
takeList : (n : Nat) -> List a -> List a
takeList Z     list       = []
takeList (S k) []         = []
takeList (S k) (x :: xs) = x :: takeList k xs
```

```
*Take> takeList 2 [1,2,3,4]
[1, 2] : List Integer

*Take> takeList 5 [1,2,3,4]
[1, 2, 3, 4] : List Integer
```

Dependent Types - Examples

takeList

```
takeList : (n : Nat) -> List a -> List a
takeList Z     list        = []
takeList (S k) []          = []
takeList (S k) (x :: xs) = x :: takeList k xs
```

takeVect

```
takeVect : (n : Nat) -> Vect (n + m) elem -> Vect n elem
takeVect  Z     xs         = []
takeVect (S k) (x :: xs) = x :: takeVect k xs
```

Dependent Types - Examples

takeVect

```
takeVect : (n : Nat) -> Vect (n + m) elem -> Vect n elem
takeVect  Z     xs        = []
takeVect (S k) (x :: xs) = x :: takeVect k xs
```

```
*Take> takeVect 2 [1,2,3,4]
[1, 2] : Vect 2 Integer
*Take> takeVect 5 [1,2,3,4]
(input):1:13:When checking argument xs to constructor Data.Vect.:::
                Type mismatch between
                                Vect 0 elem1 (Type of [])
                and
                                Vect (S m) elem (Expected type)

                Specifically:
                                Type mismatch between
                                        0
                                and
                                        S m
```

Dependent Types - Examples

vAdd

```
vAdd : Num a => Vect n a -> Vect n a -> Vect n a
vAdd Nil       Nil       = Nil
vAdd (x :: xs) (y :: ys) = x + y :: vAdd xs ys
```

```
*vAdd> vAdd [1,2,3] [1,2,3]
[2, 4, 6] : Vect 3 Integer

*vAdd> vAdd ["a",2,3] [1,2,3]
String is not a numeric type
```

Type theory: syntax

$$
\begin{array}{rcll}
\text{Terms}, t ::= & & c & \text{(constant)} \\
& \mid & x & \text{(variable)} \\
& \mid & \lambda x : t.\ t & \text{(abstraction)} \\
& \mid & t\ t & \text{(application)} \\
& \mid & (x : t) \to t & \text{(function space)} \\
& \mid & \mathtt{T} & \text{(type constructor)} \\
& \mid & \mathtt{D} & \text{(data constructor)}
\end{array}
$$

$$
\begin{array}{rcll}
\text{Constants}, c ::= & & \mathtt{Type} & \text{(type universe)} \\
& \mid & i & \text{(integer literal)} \\
& \mid & str & \text{(string literal)}
\end{array}
$$

Type theory: syntax

$$
\begin{array}{rcll}
\text{Terms}, t & ::= & c & \text{(constant)} \\
& | & x & \text{(variable)} \\
& | & \lambda x : t.\ t & \text{(abstraction)} \\
& | & t\ t & \text{(application)} \\
& | & (x : t) \to t & \text{(function space)} \\
& | & \mathtt{T} & \text{(type constructor)} \\
& | & \mathtt{D} & \text{(data constructor)}
\end{array}
$$

$$
\begin{array}{rcll}
\text{Constants}, c & ::= & \mathtt{Type} & \text{(type universe)} \\
& | & i & \text{(integer literal)} \\
& | & str & \text{(string literal)}
\end{array}
$$

Type theory: typing

$$\Gamma \vdash (\lambda x : S.\ t)\ s \rightsquigarrow_\beta t$$

Type theory: typing

$$\Gamma \vdash (\lambda x : S.\ t)\ s \rightsquigarrow_\beta t$$

$$\frac{}{\Gamma \vdash i : \texttt{Int}}\ \text{Const}_1 \qquad \frac{}{\Gamma \vdash str : \texttt{String}}\ \text{Const}_2$$

$$\frac{}{\Gamma \vdash \texttt{Int} : \texttt{Type}}\ \text{Const}_3 \qquad \frac{}{\Gamma \vdash \texttt{String} : \texttt{Type}}\ \text{Const}_4$$

Type theory: typing

$$\Gamma \vdash (\lambda x : S.\ t)\ s \rightsquigarrow_\beta t$$

$$\frac{}{\Gamma \vdash i : \texttt{Int}}\ \text{Const}_1 \qquad \frac{}{\Gamma \vdash str : \texttt{String}}\ \text{Const}_2$$

$$\frac{}{\Gamma \vdash \texttt{Int} : \texttt{Type}}\ \text{Const}_3 \qquad \frac{}{\Gamma \vdash \texttt{String} : \texttt{Type}}\ \text{Const}_4$$

$$\frac{(x : S) \in \Gamma}{\Gamma \vdash x : S}\ \text{Var}$$

Type theory: typing

$$\Gamma \vdash (\lambda x : S.\ t)\ s \rightsquigarrow_\beta t$$

$$\frac{}{\Gamma \vdash i : \texttt{Int}}\ \text{Const}_1 \qquad \frac{}{\Gamma \vdash str : \texttt{String}}\ \text{Const}_2$$

$$\frac{}{\Gamma \vdash \texttt{Int} : \texttt{Type}}\ \text{Const}_3 \qquad \frac{}{\Gamma \vdash \texttt{String} : \texttt{Type}}\ \text{Const}_4$$

$$\frac{(x : S) \in \Gamma}{\Gamma \vdash x : S}\ \text{Var}$$

$$\frac{\Gamma \vdash f : (x : S) \to T \qquad \Gamma \vdash s : S}{\Gamma \vdash f\ \ s : T[s/x]}\ \text{App}$$

Type theory: typing

$$\Gamma \vdash (\lambda x : S.\ t)\ s \rightsquigarrow_\beta t$$

$$\frac{}{\Gamma \vdash i : \texttt{Int}}\ \text{Const}_1 \qquad \frac{}{\Gamma \vdash str : \texttt{String}}\ \text{Const}_2$$

$$\frac{}{\Gamma \vdash \texttt{Int} : \texttt{Type}}\ \text{Const}_3 \qquad \frac{}{\Gamma \vdash \texttt{String} : \texttt{Type}}\ \text{Const}_4$$

$$\frac{(x : S) \in \Gamma}{\Gamma \vdash x : S}\ \text{Var}$$

$$\frac{\Gamma \vdash f : (x : S) \to T \qquad \Gamma \vdash s : S}{\Gamma \vdash f\ s : T[s/x]}\ \text{App}$$

$$\frac{\Gamma; x : S \vdash e : T \qquad \Gamma \vdash (x : S) \to T : \texttt{Type}}{\Gamma \vdash \lambda x : S.\ e : (x : S) \to T}\ \text{Abs}$$

Type theory: pattern matching definitions

$$
\begin{aligned}
&\texttt{f} : t \\
&\underline{\text{var}}\ \vec{x}_1 : \vec{t}_1.\ \texttt{f}\ \vec{t}_1 = t_1 \\
&\ \ \vdots \\
&\underline{\text{var}}\ \vec{x}_n : \vec{t}_n.\ \texttt{f}\ \vec{t}_n = t_n
\end{aligned}
$$

Type theory: pattern matching definitions

$$\begin{aligned}
&\texttt{f} : t \\
&\underline{\text{var}}\ \vec{x}_1 : \vec{t}_1.\ \texttt{f}\ \vec{t}_1 = t_1 \\
&\ \ \vdots \\
&\underline{\text{var}}\ \vec{x}_n : \vec{t}_n.\ \texttt{f}\ \vec{t}_n = t_n
\end{aligned}$$

$$\begin{aligned}
&\texttt{add} : \texttt{Nat} \to \texttt{Nat} \to \texttt{Nat} \\
&\underline{\text{var}}\ m : \texttt{Nat}. \qquad\qquad\quad \texttt{add Z}\ m = m \\
&\underline{\text{var}}\ n : \texttt{Nat}, m : \texttt{Nat}.\ \texttt{add}\ (\texttt{S}\ n)\ m = \texttt{S}\ (\texttt{add}\ n\ m)
\end{aligned}$$

Type theory: from IDRIS to TT

$$\text{IDRIS} \xrightarrow{\text{(desugaring)}} \text{IDRIS}^{-} \xrightarrow{\text{(elaboration)}} \mathsf{TT} \xrightarrow{\text{(compilation)}} \text{Executable}$$

Type theory: from IDRIS to TT

IDRIS

```
vAdd : Num a => Vect n a -> Vect n a -> Vect n a
vAdd Nil Nil = Nil
vAdd (x :: xs) (y :: ys) = x + y :: vAdd xs ys
```

Type theory: from Idris to TT

Idris

```
vAdd : Num a => Vect n a -> Vect n a -> Vect n a
vAdd Nil Nil = Nil
vAdd (x :: xs) (y :: ys) = x + y :: vAdd xs ys
```

Idris⁻

```
vAdd : (a : _) -> (n : _) ->
           Num a -> Vect n a -> Vect n a -> Vect n a
vAdd _ _ c (Nil _) (Nil _) = Nil _
vAdd _ _ c ((::) _ _ x xs) ((::) _ _ y ys)
              = (::) _ _ ((+) _ x y) (vAdd _ _ _ xs ys)
```

Type theory: from IDRIS to TT

IDRIS⁻

```
vAdd : (a : _) -> (n : _) ->
           Num a -> Vect n a -> Vect n a -> Vect n a
vAdd _ _ c (Nil _) (Nil _) = Nil _
vAdd _ _ c ((::) _ _ x xs) ((::) _ _ y ys)
                 = (::) _ _ ((+) _ x y) (vAdd _ _ _ xs ys)
```

TT

$$\mathtt{vAdd} : (a : \mathtt{Type}) \to (n : \mathtt{Nat}) \to \mathtt{Num}\ a \to \mathtt{Vect}\ n\ a \to \mathtt{Vect}\ n\ a \to \mathtt{Vect}\ n\ a$$

$\underline{\text{var}}\ a : \mathtt{Type}, c : \mathtt{Num}\ a.$

$$\mathtt{vAdd}\ a\ \mathtt{Z}\ c\ (\mathtt{Nil}\ a)\ (\mathtt{Nil}\ a) = \mathtt{Nil}\ a$$

$\underline{\text{var}}\ a : \mathtt{Type}, k : \mathtt{Nat}, c : \mathtt{Num}\ a,$

$x : a, xs : \mathtt{Vect}\ k\ a, y : a, ys : \mathtt{Vect}\ k\ a.$

$$\mathtt{vAdd}\ a\ (\mathtt{S}\ k)\ c\ ((::)\ a\ k\ x\ xs)((::)\ a\ k\ y\ ys)$$

$$= ((::)\ a\ k\ ((+)\ c\ x\ y)\ (\mathtt{vAdd}\ a\ k\ c\ xs\ ys))$$

Type theory: from IDRIS to TT

IDRIS⁻

```
vAdd : (a : _) -> (n : _) ->
           Num a -> Vect n a -> Vect n a -> Vect n a
vAdd _ _ c (Nil _) (Nil _) = Nil _
vAdd _ _ c ((::) _ _ x xs) ((::) _ _ y ys)
               = (::) _ _ ((+) _ x y) (vAdd _ _ _ xs ys)
```

TT

$$\texttt{vAdd} : (a : \texttt{Type}) \to (n : \texttt{Nat}) \to \texttt{Num}\ a \to \texttt{Vect}\ n\ a \to \texttt{Vect}\ n\ a \to \texttt{Vect}\ n\ a$$

$\underline{\text{var}}\ a : \texttt{Type}, c : \texttt{Num}\ a.$

$\quad \texttt{vAdd}\ a\ \texttt{Z}\ c\ (\texttt{Nil}\ a)\ (\texttt{Nil}\ a) = \texttt{Nil}\ a$

$\underline{\text{var}}\ a : \texttt{Type}, k : \texttt{Nat}, c : \texttt{Num}\ a,$

$\qquad x : a, xs : \texttt{Vect}\ k\ a, y : a, ys : \texttt{Vect}\ k\ a.$

$\quad \texttt{vAdd}\ a\ (\texttt{S}\ k)\ c\ ((::)\ a\ k\ x\ xs)((::)\ a\ k\ y\ ys)$

$\qquad = ((::)\ a\ k\ ((+)\ c\ x\ y)\ (\texttt{vAdd}\ a\ k\ c\ xs\ ys))$

Type theory: from Idris to TT

Idris⁻

```
vAdd : (a : _) -> (n : _) ->
           Num a -> Vect n a -> Vect n a -> Vect n a
vAdd _ _ c (Nil _) (Nil _) = Nil _
vAdd _ _ c ((::) _ _ x xs) ((::) _ _ y ys)
                = (::) _ _ ((+) _ x y) (vAdd _ _ _ xs ys)
```

TT

$$\mathtt{vAdd} : (a : \mathtt{Type}) \to (n : \mathtt{Nat}) \to \mathtt{Num}\ a \to \mathtt{Vect}\ n\ a \to \mathtt{Vect}\ n\ a \to \mathtt{Vect}\ n\ a$$

$\underline{\text{var}}\ a : \mathtt{Type}, c : \mathtt{Num}\ a.$

$$\mathtt{vAdd}\ a\ \mathtt{Z}\ c\ (\mathtt{Nil}\ a)\ (\mathtt{Nil}\ a) = \mathtt{Nil}\ a$$

$\underline{\text{var}}\ a : \mathtt{Type}, k : \mathtt{Nat}, c : \mathtt{Num}\ a,$

$x : a, xs : \mathtt{Vect}\ k\ a, y : a, ys : \mathtt{Vect}\ k\ a.$

$$\mathtt{vAdd}\ a\ (\mathtt{S}\ k)\ c\ ((::)\ a\ k\ x\ xs)((::)\ a\ k\ y\ ys)$$
$$= ((::)\ a\ k\ ((+)\ c\ x\ y)\ (\mathtt{vAdd}\ a\ k\ c\ xs\ ys))$$

Example : Words

Data type

```
data WordN : (n : Nat) -> Type where
MkWord : Int -> WordN k
```

Show

```
show : {n : Nat} -> WordN n -> String
show w = "w" ++ (show $ numBits w) ++ "=" ++ (show $ toInt w)
```

Type level

```
incBits : WordN n -> WordN (S n)
incBits (MkWord i) = MkWord i
```

Example : Words

Types and values

```
BitsPerWord : Nat
BitsPerWord = 8

Word : Type
Word = WordN BitsPerWord

accum : Word
accum = MkWord 255 -- w(8)=255

DoubleWord : Type
DoubleWord = WordN (BitsPerWord * 2)

hl : DoubleWord
hl = MkWord 300 -- w(16)=300
```

Example : Words

Functions

```
add : WordN n -> WordN n -> WordN (S n)
add (MkWord a) (MkWord b) = MkWord (a + b)

inc : WordN n -> WordN n
inc (MkWord a) = MkWord (a + 1)

show $ inc accum          -- w(8)=0
show $ inc hl             -- w(16)=301
show $ add accum hl       -- Error
```

Refined types are not dependent types!

```
{-@ type Word3 = {v:Int | v < 8} @-}

{-@ accum :: Word3 @-}
accum :: Int
accum = 7

{-@ inc :: Word3 -> Word3 @-}
inc :: Int -> Int
inc w = (w + 1) `mod` 8

{-@ dec :: Word3 -> Word3 @-}
dec :: Int -> Int
dec 0 = 7
dec w = w - 1
```

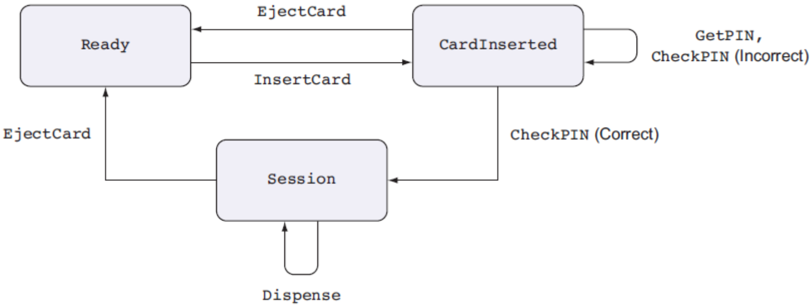
Example : Modelling an ATM DSL in Idris

An ATM should only dispense cash when a user has inserted their card and entered a correct PIN. This is a typical sequence of operations on an ATM:

- A user inserts their bank card.
- The machine prompts the user for their PIN, to check that the user is entitled to use the card.
- If PIN entry is successful, the machine prompts the user for an amount of money, and then dispenses cash. Otherwise, it ejects their bank card.

Example : Modelling an ATM DSL in Idris

A state machine describing the states and operations on an ATM.



Example : Modelling an ATM DSL in Idris

So in our ATM DSL, can be in one of the following states:

- `Ready` — The ATM is ready and waiting for a card to be inserted.
- `CardInserted` — There is a card inside the ATM, but the system has not yet checked a PIN entry against the card.
- `Session` — There is a card inside the ATM and the user has entered a valid PIN for the card, so a validated session is in progress.

```
PIN : Type
PIN = Vect 4 Int
```

```
data ATMState = Ready | CardInserted | InSession
data PINCheck = CorrectPIN | IncorrectPIN
```

Example : Modelling an ATM DSL in Idris

The machine supports the following basic operations:

- `InsertCard` — Waits for a user to insert a card.
- `EjectCard` — Ejects a card from the machine, as long as there's a card in the machine.
- `GetPIN` — Reads a user's PIN, as long as there's a card in the machine.
- `CheckPIN` — Checks whether an entered PIN is valid.
- `Dispense` — Dispenses cash as long as there's a validated card in the machine.
- `Message` — Displays a message to the user.

Example : Modelling an ATM DSL in Idris

The machine supports the following basic operations:

```
data ATMcmd: (ty: Type) -> ATMState -> ATMState -> Type where
    InsertCard : ATMcmd () Ready CardInserted
    EjectCard: ATMcmd () st Ready
    GetPIN: ATMcmd PIN CardInserted CardInserted
    CheckPIN: (p:PIN) -> ATMcmd PINCheck CardInserted (Main.chkPIN (Main.isCorrect p))
    GetAmount: ATMcmd Nat st st
    Dispense: (amount: Nat) -> ATMcmd () InSession InSession
```

Example : Modelling an ATM DSL in Idris

The machine supports the following basic operations:

```
data ATMcmd: (ty: Type) -> ATMState -> (ty -> ATMState) -> Type where
    InsertCard : ATMcmd () Ready (const CardInserted)
    EjectCard: ATMcmd () st (const Ready)
    GetPIN: ATMcmd PIN CardInserted (const CardInserted)
    CheckPIN: PIN -> ATMcmd PINCheck CardInserted Dsl.chkPIN
    GetAmount: ATMcmd Nat st (const st)
    Dispense: (amount: Nat) -> ATMcmd () InSession (const InSession)
    Message : String -> ATMcmd () st (const st)

    -- Monad
    Pure : (res: ty) -> ATMcmd ty (st_final res) st_final
    (>>=) : ATMcmd a st1 st2 -> ((res:a) -> ATMcmd b (st2 res) stf) -> ATMcmd b st1 stf
```

Example : Modelling an ATM DSL in Idris

Its interpreter (part.I)

```
runATM : ATMcmd res sti stf -> IO res
runATM InsertCard = do putStrLn "Please insert your card (press enter)"
                       x <- getLine
                       pure ()

runATM EjectCard = putStrLn "Eject card.."
runATM GetPIN = do putStrLn "Insert PIN: "
                   s <- getLine
                   pure (parsePIN s)

runATM (CheckPIN pin) = if pin == secretPIN
                        then pure CorrectPIN
                        else pure IncorrectPIN
```

Example : Modelling an ATM DSL in Idris

Its interpreter (part.II)

```
runATM GetAmount = do putStrLn "How much would you like?  "
                      amount <- getLine
                      pure (cast amount)

runATM (Dispense amount) = putStrLn ("Here is $" ++ show amount ++ ". Bye!")
runATM (Message msg) = putStrLn msg
runATM (Pure res) = pure res
runATM (x >>= f) = do res <- runATM x
                      runATM (f res)
```

Example : Modelling an ATM DSL in Idris

The program. In order to execute it, we have to type in the promt: `:exec runATM atm`

```
atm : ATMcmd () Ready (const Ready)
atm = do InsertCard
         pin <- GetPIN
         pinOK <- CheckPIN pin
         Message "Checking Card"
         case pinOK of
              CorrectPIN => do cash <- GetAmount
                                Dispense cash
                                EjectCard
              IncorrectPIN => do Message "Incorrect PIN"
                                  EjectCard
```

Example : Other

We can define the following states for our 'mood'

```
data State = Hungry | Coding | Chill

data MOOD : State -> String -> Type where
        ZZUP: MOOD Coding "idris"
        FOO: MOOD Hungry s
        BAR: Nat -> MOOD Chill s
```

Example : Other

Let's see what happens..

```
-- Singleton
zzup : MOOD Coding "idris"
zzup = ZZUP

-- Different dependent types, same constructor
foo : MOOD Hungry "pizza"
foo = FOO

foo' : MOOD Hungry "apple"
foo' = FOO

-- ##################################
-- Same type but not same constructor

paco : MOOD Chill "netflix"
paco = BAR 0

pacoMultiVerse : MOOD Chill "netflix"
pacoMultiVerse = BAR 1

-- Different dependent types, same constructor
juan : MOOD Chill "music"
juan = BAR 0
```