# PROGRAMMING WITH DEPENDENT TYPES

Kontonis Vasilis

October 17, 2015

NTUA

# TABLE OF CONTENTS

# WHY DEPENDENT TYPES?

# A SIMPLE PROBLEM

Consider a simple function which returns the first element of a non-empty array.
Here is a naive implementation in python:

```
def head(arr):
  return arr[0]
```

Some fail senarios for this code:

1. You pass value that is not an array
2. You pass null
3. You pass an empty array

These problems derive from the dynamic type system of python (also ruby etc.)

Let's try java:

```java
public int head(int[] list) {
  return list[0];
}
```

Now we are sure that the argument passed to first will be an int list. Java handles the first flaw at compile time itself but cannot know if list is Null or empty so the other 2 points still remain.

Haskell to the rescue !

```haskell
head :: [a] -> a
head xs = xs !! 0
```

Haskell is known to have a very safe static type system and doesn't allow to pass null when calling head.

However the we must be able to make sure, at compile time, that the argument passed to first will always be a **non-empty** array. That is, the length of the array should also be part of the type of the argument.

Those types are called **Dependent Types** because they depend on the value they hold.

We'll use **Idris** to write the definition of first that does not have any of those flaws.

# INTRODUCTION TO IDRIS

In conventional programming languages, there is a clear **distinction** between **types** and **values**.

In a language with dependent types, however, the distinction is less clear. Types are a first class language construct and can be **manipulated** like any other value.

The standard example is the type of lists of a given length, **Vect** n a, where a is the **element type** and n is the **length** of the list and can be an arbitrary term.

**Idris** is a general-purpose purely functional programming language with dependent types.

Note that parametric polymorphism refers to when the type of a value contains one or more (unconstrained) **type variables**, so that the value may adopt any type that results from **substituting** those variables with **concrete types**.

# NATURAL NUMBERS

Compilers don't really know the difference between 0 and 1.

An int is something that doesn't have a decimal value for compilers. 0 and 1 have different values at **runtime**, but at compile time, they are just ints.

We need to **teach** the compiler that there's a difference between them so that we can achieve our ultimate goal of accepting a non-empty array.

We'll recursively define Natural Numbers. So, a natural number is either **zero** or a **successor** of another natural number.

```
data Nat = Z | S Nat
```

```
one = S Z
two = S (S Z)
```

Idris automatically de-sugars the above Natural numbers:

```
Idris> S (S (S Z))
3 : Nat
```

# THE VECTOR DATATYPE

Let's implement Vect recursively.
A vector can be of length **zero** (Vect Z a) or it can be an element **appended** to another Vector.

In that case its length will be exactly 1 more than the other Vector.

So, if the other vector is of length n, its length will be n+1. If n is Nat, it'll be S n.

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

- Nat : length of the Vector
- Type #1 : type of the elements of the Vector.
- Type #2 : the resultant dependent type.

```
zeroVect : Vect 0 Int
zeroVect = Nil

fourVect : Vect 4 String
fourVect = "A" :: "Neat" :: "String" :: "Vector" :: Nil
```

# A SOLUTION TO OUR PROBLEM

Now that we have our Vector Datatype, writing a safe head is easy. It's just a function that takes any Vect with length greater than zero, i.e. S n where n is any Nat. Even if n is Z, S n is 1.

```
head : Vect (S k) a -> a
head (x::xs) = x
```

We can always guarantee that first always receives an array that is neither null nor empty.

We can observe now that Dependent Types allow us to encode any runtime possibility into the type and have it checked at compile time.

This allows for having **no runtime errors at all**.

# A TYPE SAFE PRINTF

# UNSAFE PRINTF

This is the case with printf from Haskell's Text.Printf

```
> printf "Hello %s, %d is the best number!\n" "There" 7
Hello There, 7 is the best number!
```

Throws a exception if we don't provide an Int for %d

```
> printf "Hello %s, %d is the best number!\n" "There" "yo"
Hello There, *** Exception: printf: bad formatting char 'd'
```

the same if we don't supply enough arguments ...

```
> printf "Hello %s, %d is the best number!\n" "There"
Hello There, *** Exception: printf: argument list ended prematurely
```

or too many.

```
> printf "Hello %s, %d is the best number!\n" "There" 7 13
Hello There, 7 is the best number!
*** Exception: printf: formatting string ended prematurely
```

# TOTALITY CHECKING

The Halting Problem states that there are programs that cannot be proven to **terminate**. That does **not** mean that it is impossible to prove that any program terminates.

Idris and other languages with totality checking put some restrictions on the forms that functions are allowed to take so that totality checking is possible.

In Idris, partial functions are allowed by default. A totality requirement can be specified per-function. This line enforces totality checking by default for functions in this module.

```
%default total
```

# PARSE INPUT

We create a simple recursive datatype to distinguish the formatting modifiers.

```
data Format = FInt Format -- %d
            | FString Format -- %s
            | FOther Char Format -- [a-zA-Z0-9]
            | FEnd
```

format parses the input string.

```
format : List Char -> Format
format ('%'::'d'::cs) = FInt (format cs)
format ('%'::'s'::cs) = FString (format cs)
format (c::cs) = FOther c (format cs)
format [] = FEnd
```

# FORMAT EXAMPLE

In idris String is not a List of Char (like Haskell) so we have to unpack the input string to provide format with a Char List.

```
formatString : String -> Format
formatString s = format (unpack s)
```

```
> formatString "Hello %s, %d"
FOther 'H'
     (FOther 'e'
           (FOther 'l'
                 (FOther 'l'
                       (FOther 'o'
                             (FOther ' '
                                   (FString (FOther ','
                                                 (FOther ' '
                                                       (FInt FEnd)))))))))
                                                             : Format
```

# TYPES DEPEND ON VALUES

interpFormat returns the type that our input Format should have. This is possible due to the dependent type system of idris.

```
interpFormat : Format -> Type
interpFormat (FInt f) = Int -> interpFormat f
interpFormat (FString f) = String -> interpFormat f
interpFormat (FOther _ f) = interpFormat f
interpFormat FEnd = String
```

```
> interpFormat $ formatString "Hello %s, %d is the best number!"
String -> Int -> String : Type
```

```
> interpFormat $ formatString "asdf %d asdf %s %d "
Int -> String -> Int -> String : Type
```

# SAFE PRINTF

For the last step we need to create a function of the type we constructed from the input Format.

toFunction takes fmt which is of Type Format, an accumulator String and returns a curried function of the type that interpFormat returns for fmt.

```
toFunction : (fmt : Format) -> String -> interpFormat fmt
```

We use pattern matching to append recursively every element to the final string.

```
toFunction (FInt f) a = \i => toFunction f (a ++ show i)
toFunction (FString f) a = \s => toFunction f (a ++ s)
toFunction (FOther c f) a = toFunction f (a ++ singleton c)
toFunction FEnd a = a
```

The notation \x => val constructs an anonymous function which takes one argument, x and returns the expression val.

```
printf : (s : String) -> interpFormat (formatString s)
printf s = toFunction (formatString s) ""
```

# MUCH SAFE SO DEPENDENT WOW

```
> printf "Hello %s, %d is the best number!" "there" 13
"Hello there, 13 is the best number!" : String
```

If we provide less arguments it simply returns a function instead of crashing

```
> printf "test %d"
\i => prim__concat "test " (prim__toStrInt i) : Int -> String
```

With more arguments it finds the mismatch at compile time

```
> printf "test %d" 10 10
builtin:Type mismatch between
      String (Type of printf "test %d" 10)
and
      argTy -> retTy (Expected type)
```

and so does if don't provide an argument of the correct type

```
> printf "test %s" 10
String is not a numeric type
```

# THEOREM PROVING

# CURRY-HOWARD AND PROOFS

According to the Curry-Howard correspondence, mathematical propositions can be **represented** in a program **as types**.

An implementation that satisfies a given type serves as a proof of the corresponding proposition. In other words, inhabited types represent true propositions.

The Curry-Howard correspondence applies to **every** language with type checking.

The type systems in most languages are **not expressive** enough to build very interesting propositions. On the other hand, dependent types can express quantification (i.e., the mathematical concepts of **universal** quantification and **existential** quantification).

This makes it possible to translate a lot of interesting math into machine-verified code.

Because a partial function can introduce a **logical contradiction**, which would make proofs unreliable, totality checking is useful for theorem proving.

We have already seen that a type can be indexed by another type. Here is a constructor for indexed types from the Idris standard library:

```
data LTE : (n, m : Nat) -> Type where
  lteZero : LTE Z right
  lteSucc : LTE left right -> LTE (S left) (S right)
```

This declares that LTE is a **constructor** that takes two Nat values as parameters, and **produces** a concrete Type. The types that LTE constructs also happen to be propositions which state that:

“the natural number n is less than or equal to the natural number m”.

# A PROPOSITION AS A FUNCTION

lteZero is a singleton value - it is a constructor that takes no arguments. But its type contains a variable; so it is **polymorphic**. lteZero can satisfy any type of the form, LTE Z n.

lteZero is effectively an axiom, stating a fundamental property of natural numbers.

Given the definition of LTE it is possible to write a proposition, such as, “zero is less than or equal to every natural number”.

```
nonNegative : (n : Nat) -> LTE Z n
```

The proposition is written as the **type** of a **function** that takes a number as input. The value that is given is assigned to the variable n, which is used to specify the return type. Thus the return type of nonNegative depends on the input value.

# A SIMPLE PROOF

To write an **implementation** for nonNegative, it is necessary to produce a value of the appropriate LTE type without any information about what input might be given - other than the fact that it will be a natural number.

Totality checking is enabled, so any implementation must be applicable to **every possible input**.

Thus a type of the form, (x : A) -> P x describes universal quantification over the type A.

nonNegative happens to be a restatement of the axiom, lteZero.

So an implementation-proof is trivial:

```
nonNegative : (n : Nat) -> LTE Z n
nonNegative n = lteZero
```

lteSucc maps a given proof to a proof of a related proposition. It is used in **proofs-by-induction**.

For example, a proof that every number is less than or equal to itself:

```
lteReflexive : (n : Nat) -> LTE n n
lteReflexive Z = lteZero
lteReflexive (S n) = lteSucc (lteReflexive n)
```

The proof that zero is equal to itself is given by the axiom.

For every other number, the proof is given as an **inductive step** using a proof for the next-smallest number.

Because the type of lteSucc is that of a function, it can be read as a proposition involving logical implication:

“n <= m implies that n + 1 <= m + 1.”

In general, types of the form P x -> Q y can be read as **logical implication**.

# EXISTENTIAL QUANTIFICATION

We have seen that the input value to a function can be labelled and referenced in the output type.

That is a special property of functions.

For example, it is not permissible to label the value in the first position of a tuple type to reference it in the second position. For this reason there is a special construction, the dependent pair, which does allow this.

Dependent pairs are used to represent existential quantification.

A dependent pair type of the form (x : A ** P x) is read as existential quantification over the type A.

A proof of a proposition with existential quantification can be given as a **pair** of an arbitrary value and a proof that the proposition holds for that value.

# ARCHIMEDEAN PROPERTY

"For every natural number, n, there exists a natural number, m, where m > n":

```
archimedean : (n : Nat) -> (m : Nat ** LTE (S n) m) -- n + 1 <= m
archimedean n = (S n ** lteReflexive (S n))
```

The quantified proposition uses (S n) instead of just n to indicate that m must be **strictly greater** than n - greater-than-or-equal-to is not sufficient.

The Dependent Pair consists of:

- The **witness** S n - a specific value that is used to prove that the quantified proposition holds.
- The **proof** that the witness is greater than or equal to S n - as is required by the type. Since S n and S n are **equal**, lteReflexive suffices.

# TYPE PROVIDERS

# WHAT ARE TYPE PROVIDERS

Idris type providers, inspired by F#'s type providers, are a means of making our types be "about" something in the world outside of Idris.

For example, given a type that represents a **database schema** and a **query** that is checked against it, a type provider could read the schema of a real database during type checking.

Idris type providers use the ordinary execution semantics of Idris to run an IO action and extract the result. This result is then saved as a constant in the **compiled** code.

It can be a type, in which case it is used like any other type, or it can be a value, in which case it can be used as any other value, including as an index in types.

# A SIMPLE PROVIDER

A provider p for some type t is simply an expression of **type IO (Provider t)**. The %provide directive causes the type checker to execute the action and bind the result to a name.

The type provider fromFile reads a text file. If the file consists of the string ”Int”, then the type Int will be provided. Otherwise, it will provide the type Nat.

```
strToType : String -> Type
strToType "Int" = Int
strToType _ = Nat

fromFile : String -> IO (Provider Type)
fromFile fname = do
  str <- readFile fname
  return (Provide (strToType (trim str)))
```

# A SIMPLE PROVIDER

We then use the %provide directive:

```
%provide (T1 : Type) with fromFile "theType"
foo : T1
foo = 2
```

If the file named theType consists of the word Int, then foo will be an Int. Otherwise, it will be a Nat.

When Idris encounters the directive, it first **checks** that the provider expression fromFile "theType" has type IO (Provider Type).

Next, it **executes** the provider. If the result is Provide t, then T1 is defined as t. Otherwise, the result is an error.

# THE PROVIDER DATATYPE

Our datatype Provider t has the following definition:

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

We have already seen the Provide constructor. The Error constructor allows type providers to return useful error messages

# Questions?

# References I


- Ana Bove and Peter Dybjer *Dependent Types at Work*
- Thorsten Altenkirch, Conor McBride, James McKinna *Why Dependent Types Matter*
- Conor McBride *Totality Versus Turing-Completeness*
- David Raymond Christiansen *Dependent Type Providers*
- Idris Documentation and Official Tutorial `https://idris.readthedocs.org/en/v0.9.19.1/index.html`
- Brian McKenna *A Type Safe Printf* `https://gist.github.com/puffnfresh/11202637`
- Haskell Wiki `https://wiki.haskell.org/Haskell`