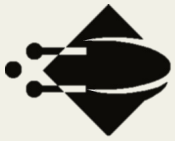
The Practical Guide to Levitation

By Ahmad Salim Al-Sibahi
Supervisors:
Dr. Peter Sestoft
David R. Christiansen



IT University
of Copenhagen

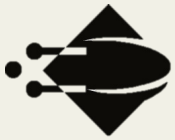
A Practical Guide to Levitation

http://wordswithoutborders.org/article/a-practical-guide-to-levitation

- Excellent, but unrelated short story by José Agualusa

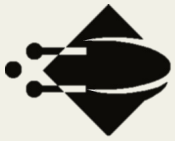
Disposition

- Introduction
- Idris overview
- Tutorial on described types
- Generic algorithm examples
- Overhead and specialisation
- Discussion



IT University
of Copenhagen

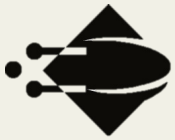
INTRODUCTION



IT University
of Copenhagen

Background

- Generic programming allows writing algorithms that work on the structure of datatypes
- This saves time and helps avoiding programmer errors
- It is possible to have first-class descriptions of datatypes in dependently typed programming languages
- Work is mostly theoretically focused



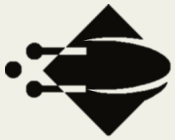
IT University of Copenhagen

Motivation

To investigate how described types can be used in a practical language, what challenges arise, and how to achieve acceptable performance

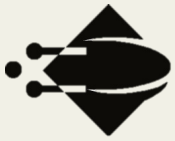
Key Contributions

- Example-based tutorial for understanding described types
- Procedure generating descriptions from ordinary datatype declarations
- Generic implementation of commonly needed functionality, notably decidable equality
- Discussion of the challenges of implementing a generic traversal library in a dependently-typed programming language
- Design of a specialisation algorithm for datatypes based on partial evaluation techniques, suited for optimising described types



IT University of Copenhagen

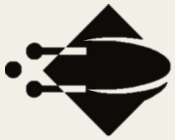
IDRIS OVERVIEW



IT University
of Copenhagen

Description

- Haskell and ML inspired programming language with full dependent types
- Support for some automation using a small set of tactics
- Practically oriented, with features such as partial functions, codata, type classes, and aggressive erasure



IT University of Copenhagen

Core Features

- Indexed datatypes

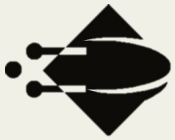
```
data Vec : (a : Type) -> Nat -> Type where
    Nil  : Vec a Z
    Cons : {n : Nat} ->
           a -> Vec a n -> Vec a (S n)
```

- Type-level functions

```
So : Bool -> Type
So True  = ()
So False = _|_
```

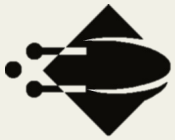
- Built-in notion of propositional equality

```
stillnot1984 : 2 + 2 = 4
Stillnot1984 = Refl
```



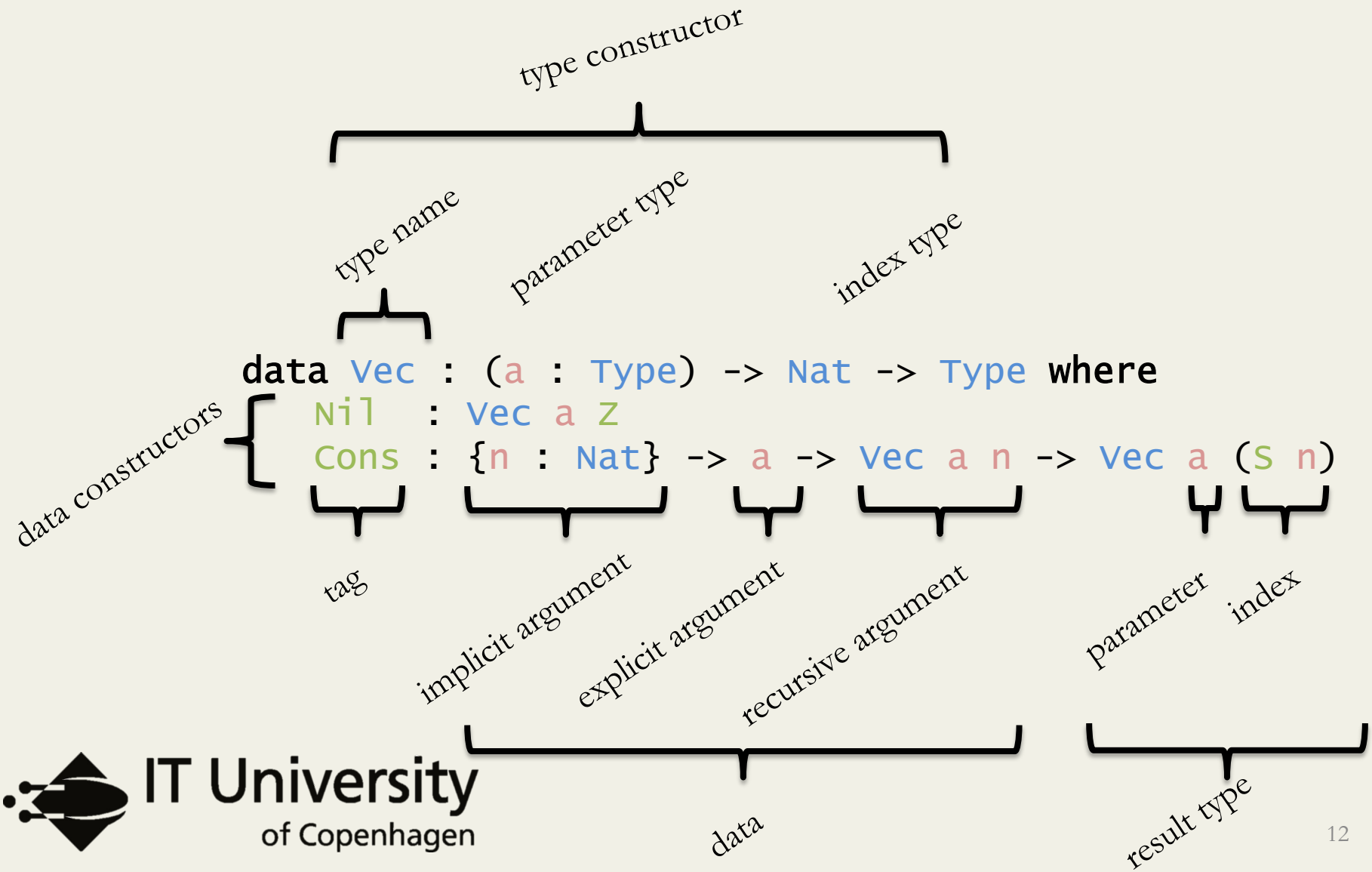
IT University of Copenhagen

TUTORIAL ON DESCRIBED TYPES



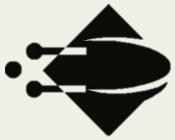
IT University
of Copenhagen

Anatomy of a datatype



Describing datatypes

```
data Desc : Type where
   Ret  : Desc
   Arg  : (a : Type) -> (a -> Desc) -> Desc
   Rec  : Desc -> Desc
```

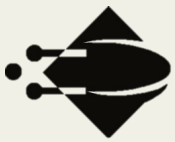


IT University of Copenhagen

Describing natural numbers

```
data Nat : Type where
    Zero : Nat
    Succ : Nat -> Nat
```

```
NatDesc : Desc
NatDesc = Arg Bool
  (\isZero =>
          if isZero
          then Ret
          else Rec Ret)
```

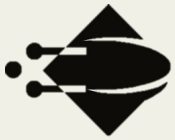


IT University of Copenhagen

Describing list

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

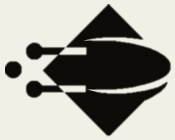
```
ListDesc :
  (a : Type) -> Desc
ListDesc a =
Arg Bool (\isNil =>
  if isNil
  then Ret
  else Arg a (\x => Rec Ret))
```



IT University of Copenhagen

Supporting indexing

```
data Desc : (ix : Type) -> Type where
   Ret  : ix -> Desc ix
   Arg  : (a : Type) -> (a -> Desc ix) ->
          Desc ix
   Rec  : ix -> Desc ix -> Desc ix
```



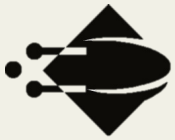
IT University of Copenhagen

Informative encoding of tags

```
CLabel : Type
CLabel = String

CEnum : Type
CEnum = List CLabel


data Tag : CLabel -> CEnum -> Type where
  TZ : Tag l (l :: e)
  TS : Tag l e -> Tag l (l' :: e)
```



IT University of Copenhagen

Describing vectors

```
data Vec :
  (a : Type) -> Nat -> Type
where
    Nil  : Vec a Z
    Cons : {n : Nat} -> a ->
           Vec a n ->
           Vec a (S n)
```

```
VecDesc :
      (a : Type) -> Desc Nat
VecDesc a =
  Arg CLabel (\l=>
    Arg (Tag l
          [ "Nil" , "Cons" ])
      ((switchDesc
          (Ret Z
        , (Arg Nat (\n=>
           Arg a (\x=>
           Rec n
           (Ret (S n))))
             , () ) )) l))
```

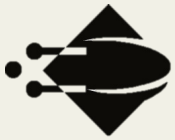
IT University of Copenhagen

Synthesising datatypes

```
Synthesise : Desc ix -> (ix -> Type) ->
             (ix -> Type)
Synthesise (Ret j)       x i = (j = i)
Synthesise (Rec j d)     x i =
  (rec : x j ** Synthesise d x i)
Synthesise (Arg a d)     x i =
  (arg : a ** Synthesise (d arg) x i)
```

Tying-up recursion

```
data Data : {ix : Type} -> Desc ix
                        -> ix -> Type where
  Con : {d : Desc ix} -> {i : ix} ->
        Synthesise d (Data d) i ->
        Data d i
```

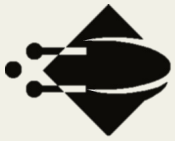


IT University of Copenhagen

Example vector

```
exampleVec : Data (VecDesc Nat) 3
exampleVec =
 Con ("Cons" ** (TS TZ ** (2 ** (1 **
  (Con ("Cons" ** (TS TZ ** (1 ** (2 **
   (Con ("Cons" ** (TS TZ ** (0 ** (3 **
    (Con ("Nil" ** (TZ ** Refl)) ** Refl)
                 )))) ** Refl)
               )))) ** Refl)
             ))))
```

EXAMPLES OF GENERIC ALGORITHM



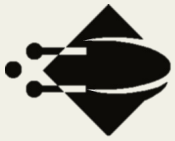
IT University of Copenhagen

Pretty printing

1. Print the name of the constructor
2. Pretty print the arguments
   a. If argument is of the same type, repeat procedure recursively
   b. Otherwise, find the corresponding procedure for the relevant datatype and use that for pretty printing

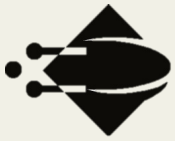
Decidable equality

1. Check if constructor tags are equal, otherwise disprove using difference of constructors lemma
2. Iterate through arguments
   a. Check if current arguments are equal, otherwise disprove using injectivity of current argument lemma
   b. Check if the rest of the arguments are equal, otherwise disprove using injectivity of rest of arguments lemma
3. The resulting types are equal if not disproved



IT University of Copenhagen

OVERHEAD AND SPECIALISATION



IT University of Copenhagen

Overhead comparison

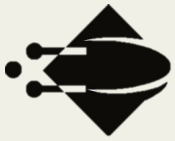
- Generic version of [42]

```
Con ("Cons" ** (TS TZ **
          (0 ** (42 **
                (Con ("Nil" ** (TZ ** Refl))
                             ** Refl)))))
```

IT University of Copenhagen

Overhead issues

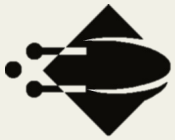
- Large amount of data is purely encoding, and does not add significantly more information
- Index arguments cannot be automatically erased because they are used in the data
- Has implicit arguments which further increases elaboration time



IT University of Copenhagen

Specialisation algorithm

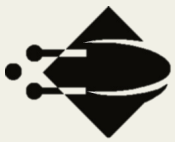
- Specialise static parameters
- Unbox nested types
- Applying the trick
- Substitute indices
- Do corresponding specialisation for dependent functions



IT University
of Copenhagen

Specialising static parameters

```
data Data__Vec_Int : Nat -> Type where
    Con : Synthesise (VecD Int) Data__Vec_Int i ->
          Data d i
```



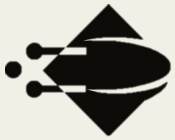
IT University of Copenhagen

Inlining and unboxing

```
data Data__Vec_Int : Nat -> Type where
   Con : (l : CLabel) ->
         (t : Tag l ["Nil", "Cons”]) ->
         (arg : Synthesise (switchDesc (
                  Ret Z
                , Arg Erasable Nat (\n =>
                       Arg None Int (\arg =>
                          Rec n (Ret (S n))))
                       ,()
            ) l t) i) -> Data d i
```

The Trick and index subsitution

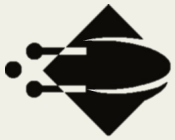
```
data Data__Vec_Int : Nat -> Type where
   Con_Nil : Data_Vec_Int Z
               «l = "Nil", t = TZ, i = Z»
   Con_Cons : .(n : Nat) -> (arg : Int)
                  -> (rec : Data_Vec_Int n)
                  -> Data_Vec_Int (S n)
              «l = "Cons", t = TS TZ, i = S n»
```



IT University of Copenhagen

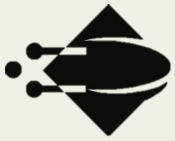
Advantages

- Exploits the added type information in Idris and other dependently-typed languages
- Is not specialised to a particular representation
- Does not require a mapping to existing datatypes
- Works independently of Constructor Specialisation



IT University of Copenhagen

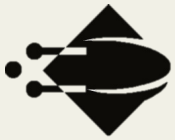
DISCUSSION



IT University
of Copenhagen

Limitations

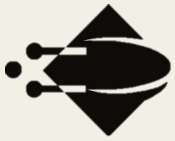
- Constraints for generic algorithms
  - New type of boilerplate
  - Not obvious if correct for nested datatypes
- Implementation and constraints
  - Interaction with advanced datatypes
  - Heuristics for applying the trick
  - Possible issues with unfolding
- Performance
  - Lack of quantitative analysis (benchmarking)



IT University of Copenhagen

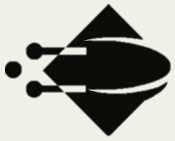
Conclusion

- Provided extensions for Idris to work with dependent types
- Showed how commonly used, and other interesting generic algorithms could be implemented in a practical language
- Designed a suitable specialisation algorithm



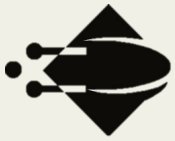
IT University of Copenhagen

QUESTIONS



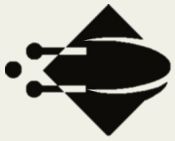
IT University
of Copenhagen

OTHER SLIDES



IT University
of Copenhagen

GENERIC TRAVERSAL AND LIMITATIONS



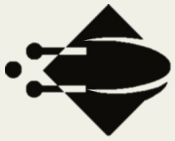
IT University of Copenhagen

Background

- Type-directed querying and transformation
- Automatically derivable using generic programming
- Example: capitalise all titles in a CMS system (with tree-like structure)

Limitations

- Requires type casting, loses parametericity
- A deriving algorithm cannot differentiate between data used for indexing and ordinary data
- Hard to model a flexible type signature without break datatype invariants



IT University of Copenhagen