The Practical Guide to Levitation

Ahmad Salim Al-Sibahi

Advisors:
Dr. Peter Sestoft
& David R. Christiansen
Submitted: September 1, 2014

IT University
of Copenhagen

Abstract

Goal: Implementation of levitation in a realistic setting, with practical performance benefits.

DISCLAIMER: This is a draft and as such is incomplete, incorrect and can contain grammatical errors. Although I claim originality of this report, many underlying ideas are based on current work in the scientific community which will be correctly attributed when the work is complete.

Contents

Chapter 1

Introduction

1.1 Context

Algebraic datatypes such as Boolean values, lists or trees form a core part of modern functional programming. Most functions written work directly on such datatypes, but sometimes it is the case that there are functions that don't directly dependent on the datatype itself and only are there to serve as a repetitive exercise; For example, in cases as structural equality or pretty printing (see Example 1). In fact it is possible to write an algorithm over the structural definition of the datatype, which the computer then could use to derive an actual function for particular datatypes.

Example 1. Pretty printing a data for any algebraic data type follows a very simple step of procedures:

1. Print the name of the constructor
2. Iterate through the data elements and pretty print them, with all the data surrounded by parentheses if necessary
    a) If the data element is a recursive reference to the type itself, then call this procedure recursively (starting at point 1.)
    b) If the data element is of another type
        i. Find the correct pretty printing function for that type
        ii. Pretty print the field using the found function
3. Finish printing

Enter the world of *generic programming* where the target datatype is the one describing the structure of other datatypes, often called the *description*. While generic programming sounds promising, it is usually seen as a challenging aspect of Haskell (Peyton Jones et al. 2003) to use by ordinary programmers. To represent the description, it is often required to use special language extensions (Magalhães et al. 2010) and the programming style tends to require different abstractions than when writing ordinary programs.

However, in dependently-typed languages such as Idris (Brady 2013) or Agda (Norell 2009) that it is possible to mitigate such issues. Due to the nature of dependently-typed languages it is possible to create a correct description using ordinary datatype definitions (Benke et al. 2003). Furthermore, Chapman et al. (2010) show that it is possible to build a self-supporting closed type system which is able to convert these descriptions to ordinary types (creating so-called *described types*), while still being powerful enough to describe the description datatype itself. Therefore in such system, generic programming is just a special case of ordinary programming.

1.2 Problem definition

The current work on generic programming in dependently-typed languages presents both elegant and typesafe ways to represent the structural descriptions of datatypes. Furthermore, it allows the programmer to save both time and boilerplate code while reducing mistakes by using ordinary programming techniques to do generic programming.

However, the state of the art is heavily theoretically oriented, which might lead to issues when a system needs to be developed with a practical audience in mind. First of all, multiple description formats are often presented, sometimes even in the same paper, which might not be particularly attractive in a practical setting. Secondly, there has been little work done on how to integrate such descriptions in languages which contain features such as type classes and proof scripts. Finally, datatypes synthesised from descriptions create large canonical terms thus, both type checking and runtime performance are very slow. In summary, if an efficient and easily usable framework for programming with described types that avoids the described issues could be implemented successfully, it would save programmers both the time and effort required to write repetitive functions.

1.3 Aim and scope

The aim of this research is to provide a practical and efficient implementation of described types in Idris. This project has three primary subgoals.

The first subgoal is to find a good definition of the description that supports many common datatypes. I seek to mainly reuse some of the existing work, and not try to improve underlying type theory to support more complex inductive families; neither will I focus on supporting all language features of Idris such as implicit arguments and codata definitions.

The second subgoal is to present realistic examples using generic functions working on described types. This mainly includes implementing functions that can be used to derive type class instances, and a Scrap Your Boilerplate (SYB) library for generic querying and traversal.

The final subgoal is that I seek to use partial evaluation to optimise the generic functions with regards to specific descriptions in order to achieve acceptable performance. This includes using techniques such as polyvariant partial evaluation and constructor specialisation.

1.4 Significance

The main contributions of this thesis are:

- an example-based tutorial for understanding described types in the context of a practical programming language, namely Idris
- an generic implementation of common operations such as decidable equality, pretty printing and functors, which can be used to provide default implementations to type class methods.
- a discussion of the challenges that arise when trying to implement a SYB-style generics library in dependently typed languages
- optimisation techniques based on partial evaluation for reducing runtime size and time overhead for described types and accompanying generic functions
- metrics showing that generic programming using described types is a viable option to reduce boilerplate without significant cost in performance

1.5 Overview

The report is structured as follows. Chapter 2, presents an introduction to described types specifically focusing on recent developments using dependently-typed programming languages. Chapter 3, presents an overview of techniques for partial evaluation of functions and specialisation of datatypes. Chapter 4, discusses specifically how described types are implemented in Idris and continues with practical examples in Chapter 5. Chapter 6 presents what optimisations were made in order to improve the runtime performance of described types and generic functions. Evaluation of the results happens in Chapter 7, comparing the performance of generic implementations to hand-written ones. Finally, Chapter 8 discusses what challenges still lie ahead and concludes the effort.

Chapter 2

Generic programming

2.1 The generic structure of inductive data types

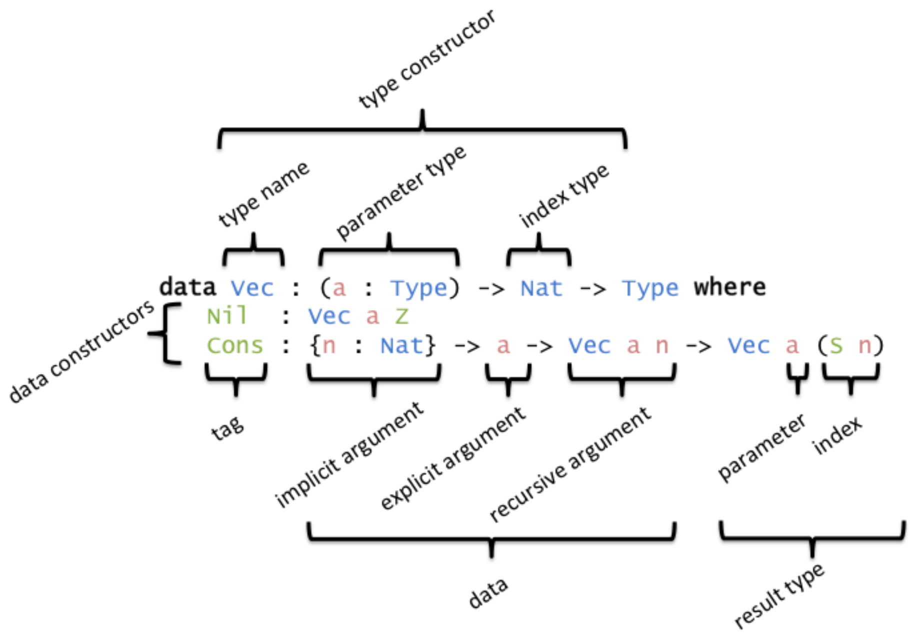
2.1.1 Anatomy of a datatype

To build an intuition that will be useful in understanding descriptions, let us first start by looking closely at how datatypes are structured. Figure 2.1, presents an annotated version of a typical dependently-typed datatype representing vectors.

A datatype consists of a type constructor which describes what type-level arguments are required, and zero or more data constructors which describe how to create values of the datatype. The type constructor has three components: a name for the datatype, types of any possible parameters, and the types of possible indices. In Figure 2.1 there is no syntactic difference between a parameter type or an index type since Idris figures that out automatically.[1]

Similarly to the type constructor, a data constructor needs a name, also called a *tag*. Following the tag, the data constructor declaration contains the types of the arguments stored in the constructor and the resulting type which must use the type constructor of the datatype. In our example two constructors are declared, `Nil` and `Cons`. Constructor `Nil` doesn't hold any data, so it only needs to define the resulting type which is `Vec a Z` (a vector with length 0). Constructor `Cons` contains three different types of arguments: an ordinary implicit argument, an ordinary explicit argument, and an explicit argument of the type itself

[1]If an argument to a type constructor doesn't change in the data constructor declarations, Idris considers it a parameter, otherwise an index.



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

Figure 2.1: Annotated components of a datatype Declaration

(recursive); the resulting type for `Cons` is `Vec a (S n)`, that is, a vector of length 1+n where n is the length of the recursive argument.

The colouring scheme for code presented in this paper uses the following conventions:

> **`Blue`** is used for type constructors
>
> **`Green`** is used for data constructors
>
> **`Dark Red`** is used for top-level declarations
>
> **`Light Red`** is used for locally-bound variables
>
> **`Purple`** is used for literals (integer, string, etc.)

2.1.2 A description for datatypes

It is now possible to try to represent a suitable description datatype. Figure 2.8 presents one possible solution, influenced mainly by the work of McBride (2010) and Diehl & Sheard (2014).

The description datatype has three main constructors:

- Constructor `Ret` represents the end of a description

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

Figure 2.2: A datatype that describes other datatypes

- Constructor `Arg` represents the addition of an argument of any type to a given description; the first argument of `Arg` is the type of argument expected and the second argument is the rest of the description dependent on a value of that type.
- Constructor `Rec` represents an recursive argument of the described datatype. The argument of constructor `Rec` is the specification of the rest of the description.

In order to get an idea on how descriptions for various interesting datatypes look like, the following paragraphs will show a series of examples translating ordinary declarations to descriptions. The declaration of the trivial singleton type `Unit` is shown in Figure 2.3a, and its corresponding description is shown in Figure 2.3b. Since `Unit` doesn't hold any interesting arguments, `Ret` is used to simply end the description.

```
data Unit : Type where
   MkUnit : Unit
```

(a) Declaration of `Unit`

```
UnitDesc : Desc
UnitDesc = Ret
```

(b) Description of `Unit`

Figure 2.3: The `Unit` datatype and its description

A more interesting datatype is shown in Figure 2.4a, namely the datatype `Pair` representing a pair of `Int` and `Bool`. The translation to the corresponding description, as shown in Figure 2.4b, seems straightforward. For each argument of `MkPair`, we use `Arg` to specify its corresponding type and bind its value in a lambda expression so that it can be used by the rest of the description. In fact, for most arguments with the form `(arg : a) -> b` the corresponding description would be of the form `Arg a (\arg => b)` Finally, to specify the end of the description, `Ret` is used.

```
data Pair : Type where
   MkPair : (fst : Int) ->
            (snd : Bool) -> Pair
```

(a) Declaration of `Pair`

```
PairDesc : Desc
PairDesc = Arg Int (\fst =>
           Arg Bool (\snd => Ret))
```

(b) Description of `Pair`

Figure 2.4: A pair of `Int` and `Bool`

A key aspect of algebraic datatypes is the ability to choose between multiple constructors. Figure 2.5a shows a simple datatype `Either` which provides two constructors `Right` and `Left`, than can hold a value of `Int` and `String` respectively.

Since there is no explicit way to encode a choice between multiple constructor, a boolean argument `isRight` is used to determine which constructor is specified. If the value of `isRight` is true then the resulting description is expected to be for the `Right` constructor, otherwise is is expected to be for the `Left` constructor. The description for each constructor is then specified in a similar fashion to datatypes with one constructor such as `Pair` described above.

```
data Either : Type where
   Right : (x: Int)    ->
           Either
   Left  : (x: String) ->
           Either
```

(a) Declaration of `Either`

```
EitherDesc : Desc
EitherDesc = Arg Bool (\isRight =>
   if isRight
   then Arg Int (\x => Ret)
   else Arg String (\x => Ret))
```

(b) Description of `Either`

Figure 2.5: the sum type of `Int` and `String`

In addition to allowing the choice between multiple possible constructors, what makes algebraic datatypes interesting is the ability to have recursive (or *inductive*) instances. The simplest recursive datatype is the natural numbers `Nat` (shown in Figure 2.6a) which has two constructors, `Zero` which represents `0` and the recursively defined `Succ` which represents `1`+`n` for any natural number `n`. The corresponding description is shown in Figure 2.6a which is mainly built up using the principles introduced before. The only addition is that the description for `Succ` now uses `Rec` to specify that it requires a recursive argument (to type `Nat` itself).

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

(a) Declaration of `Nat`

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

(b) Description of `Nat`

Figure 2.6: The Natural numbers (`Nat`)

Figure 2.7a shows one of the classical datatypes in functional programming languages, namely `List`. Unlike `Pair` and `Either` which were monomorphic in the presented examples, this time `List` is polymorphic in its elements. The way to represent parameters is by having them as arguments to the description value, which allows them to be qualified over the whole description. The description value itself is built using the previously described methods and is shown in Figure 2.7b. There is a Boolean argument `isNil`, which encodes the choice between the two constructors of the list, `Nil` and `Cons`. The description for `Nil` is simply `Ret` since it doesn't accept any arguments, and the description for `Cons` takes an argument of the parameter type (the head of the list) a recursive argument (the tail of the list) and ends the description.

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

(a) Declaration of `List`

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

(b) Description of `List`

Figure 2.7: A polymorphic list of elements

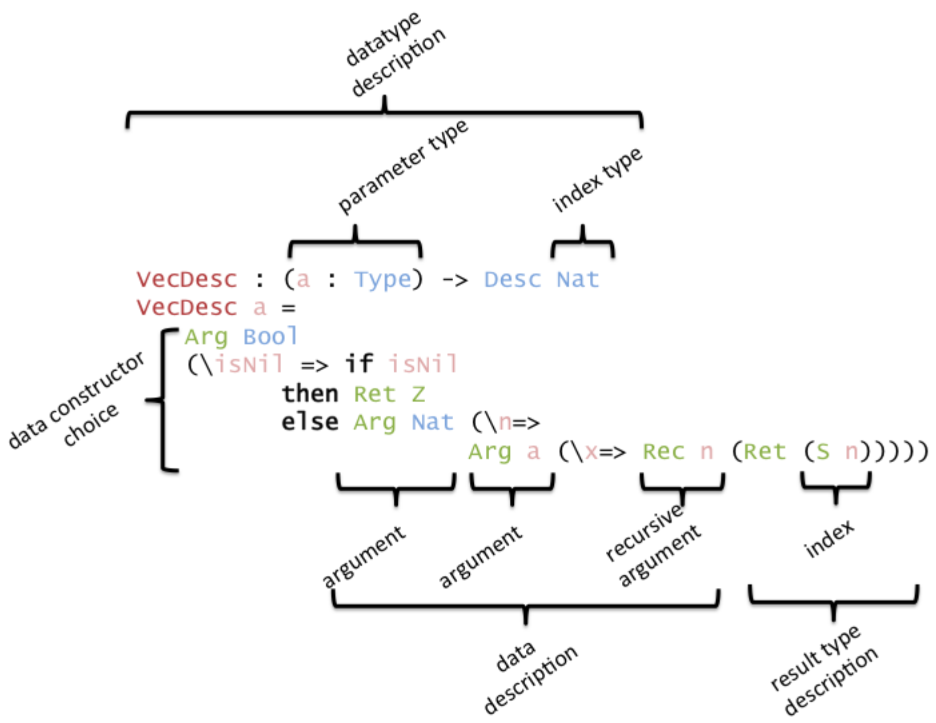
2.1.3 Indexing descriptions

In order to allow datatypes to be indexed by values, the description structure takes a parameter that describes what the type of indices must be. Figure 2.8 shows an updated version of Figure 2.2, that contains the necessary parameter `ix` for indexing datatypes. The constructors `Ret` and `Rec`, must also be updated to take a value of `ix` in order to represent what the index for the result type and recursive argument must be respectively. For descriptions that don't require indices, they can be converted to indexed descriptions using the unit type `Unit` (or using its syntactic form `()`) as index.

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

Figure 2.8: Description for datatypes with possible indices

In order to given an example on how an indexed datatype looks like, let us take a look at `Vec` from Figure 2.1. Figure 2.9 shows the description of `Vec` with comparable annotations.



```
VecDesc : (a : Type) -> Desc Nat
VecDesc a =
  Arg Bool
  (\isNil => if isNil
         then Ret Z
         else Arg Nat (\n=>
                 Arg a (\x=> Rec n (Ret (S n)))))
```

Figure 2.9: Described version of `Vec`

The type signature for the description of `Vec` mimics the one for the actual datatype closely, but there are nonetheless some differences. There is now an explicit distinction between parameters and indices; the type for a parameter can still be specified as an argument for the description value, whereas the type of an index must be provided to the `Desc` type constructor. This is to ensure that all provided indices conform to same expected type, while still allowing the values to change inside the description.

A Boolean argument `isNil` is used to describe the choice between constructors `Nil` and `Cons`. The constructor `Nil` doesn't contain any data so we simply use `Ret Z`, this indicates that the description is finished and the resulting type is expected to have index `Z`, analogously to Figure 2.1. The constructor `Cons` takes first two ordinary arguments: a `Nat` argument to be used to describe the necessary indices, and an argument of the parameter `a`. Following these arguments we take a recursive argument—specified using `Rec`—that must satisfy to have the value of the input `Nat` argument `n` as index, i.e. the argument must be of type `Vec a n`. We finish the description with `Ret` and specify that the resulting index must be `S n` as expected.

Challenges and limitations

Even though it was possible to describe a variety of datatypes there are still a few questions that can be raised, such as: How is it possible to choose between more than two constructors? Why is there only one type for indices? Why aren't the type of parameters required to be encoded inside the description datatype itself? I seek to clarify these questions in the following paragraphs.

To encode the choice of more than two constructors, a simple solution could be to nest multiple Boolean values acting as a form of binary enumeration of tags. However this encoding is fairly crude: it doesn't capture important information such as the names of constructors, requires a series of possibly complicated tests and is not easily extendible if one wants to extend descriptions with new constructors. In Section 2.1.4, I will present a more sophisticated encoding that doesn't suffer from these limitations.

For more demanding datatypes that needs more than one type of index, the indices must be uncurried using dependent pairs. For example, a datatype with signature `(n : Nat) -> Fin n -> Type` must use the dependent pair `(n : Nat ** Fin n)` as the type for its index.

Finally, since parameters are usually qualified over the whole datatype (*i.e.*, they do not change) it is possible to just accept them as external arguments when building a description. However, this can encoding can exclude interesting generic programs for being written such as the functorial map. In Chapter 4, I will discuss a modification to the description datatype that permits encoding the types of parameters directly.

2.1.4 An informative encoding of constructors

In Section 2.1.2 a Boolean variable was used to determine the choice between two constructors, and concluded that his approach had multiple disadvantages. First of all, the Boolean encoding doesn't capture the names of the respective constructors which might be important when it is desired to pretty print or serialise a data structure. Secondly, when there are more than two constructors, it can quickly become complicated to provide a suitable description. Multiple Boolean arguments are required and mapping these Boolean values to description is not exactly straightforward, *e.g.*, should two Boolean values encode the choice between 3 or 4 constructors? Finally, and perhaps more importantly, it is not easy to modify the number of constructors easily with the Boolean encoding. That is, it might be desirable to compute a new description from a provided one and in that process to add a new constructor, *e.g.*, adding a default "error" constructor to each datatype. This section presents a more informative encoding of constructors, and shows how it is possible to use that encoding when describing non-trivial datatypes.

To represent which constructors are available we first are going to declare two types as shown in Figure 2.10a: `CLabel` which represents a name for a constructor, and `CEnum` which represents a list of constructor names. Figure 2.10b show an example of how to represent the available constructor names of `Vec`.

```
CLabel : Type
CLabel = String

CEnum : Type
CEnum = List CLabel
```

(a) Representation

```
VecCtors : CEnum
VecCtors = [ “Nil” , “Cons” ]
```

(b) Example: Constructors of Vec

Figure 2.10: Constructor Labels

Now that it is possible to represent the available constructors, we can encode a way of choosing a particular constructor tag. Figure 2.11 shows a datatype `Tag` with two constructors: `TZ` which represents the constructor that is on top of the current list and TS which represents a constructor further along the list. As such, Tag specifies a valid index into a (non-empty) list of constructor tags.

This encoding has multiple advantages: it ensures that all constructor labels stored in our data exist in the expected list of constructors, it ensures that all datatypes which are dependent on a tag must have at least one constructor, and as a consequence it is possible to specify the empty type by simply requiring a tag on an empty list of expected constructors (since such value would be impossible to create). This encoding eases the implementation of automatically retrieving a tag given a constructor label using tactics in Idris, which might save some time when values are constructed manually.

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

Figure 2.11: Tags: A Structure for Picking a Constructor from a Label Collection

For the constructors of Vec, Figure 2.12 shows an example on what the tag values are. For Nil the value is TZ since it is the first in the list of VecCtors, and for Cons the value is TS TZ since it is the second. Since there are only two elements in VecCtors, it shouldn't be possible to create any other valid constructor, and as such it is a good representative for enumerating the constructors of Vec.

```
NilTag : Tag "Nil" VecCtors
NilTag = TZ
```

(a) Tag for Nil

```
ConsTag : Tag "Cons" VecCtors
ConsTag = TS TZ
```

(b) Tag for Cons

Figure 2.12: Example: Tags for constructors of Vec

2.1.5 A constructive type of choice

Similarly to how **if** was used to map Bool values to the descriptions of the various constructors of a datatype in Section 2.1.2, it is desirable to have a way to map Tag values. While an initial proposal would be to use **`case of`** to do such mapping, it doesn't compose well with extension of the count of constructors. This is because it requires that all possible cases are matched on, or at least the programmer must provide a suitable function to handle the rest of the cases (which is often an issue when doing generic programming). The following section will describe the `switch` function that allows mapping `Tag` values while remaining extensible.

Since the number of constructors for a datatype can vary in length, it is necessary to calculate a type that allows mapping the tag of each constructor to a suitable value (shown in Figure 2.13). It essentially is a function that maps the list of constructors to a series of right-nested pairs such that there is one value for each constructor tag, and ends with `()`. The type of the resulting value `prop` can be dependent on the input constructor tag, and therefore the function is called π or the *small pi operator*. The operator π is small in the sense that unlike the dependent function type Π which allows the result to dependent on any type of input, π only allows a dependency on constructor tags.

```
SPi : (e : CEnum)
      -> (prop : (l : CLabel) -> (t : Tag l e) -> Type)
      -> Type
SPi []       prop = ()
SPi (l :: e) prop =
   (prop l TZ, SPi e (\l' => \t => prop l' (TS t)))
```

Figure 2.13: The Small Pi Operator: Type for Case Analysis based on Constructor Tags

Given a way to map a list of constructors to a list of values using π, it is now possible to define a function `switch` which can look up the correct value for a particular `Tag`. The function `switch` is shown in Figure 2.14 and has two branches: if the constructor to map is the first one in a list of constructors, it simply returns the first value in the corresponding mapping, otherwise it continues the search using the rest of the provided elements (skipping the first constructor and its corresponding mapping).

As an example, Figure 2.15 shows the description of `Vec` (from Figure 2.9) again, but this time using the new constructor tag encoding instead of a Boolean variable. While the description might initially seem more complicated than before, it has a couple of clear advantages: the

```
switch : (e : CEnum)
   -> (prop : (l : CLabel) -> (t : Tag l e) -> Type)
   -> SPi e prop
   -> ((l' : CLabel) -> (t' : Tag l' e) -> prop l' t')
switch (l' :: e) prop ((propz, props)) l' TZ      = propz
switch (l  :: e) prop ((propz, props)) l' (TS t') =
   switch e (\ l => \ t => prop l (TS t)) props l' t'
```

Figure 2.14: Calculation of a Property based on a Specific Constructor Tag

encoding now contain the constructor tag and it is possible to choose between more than two constructors at the same time.

```
VecDesc : (a : Type) -> Desc Nat
VecDesc a =
   Arg CLabel (\l=>
      Arg (Tag l VecCtors)
          ((switch VecCtors (\l'=> \t'=> Desc Nat)
             ( Ret Z
             , (Arg Nat (\n=>
                         Arg a (\x=> Rec n (Ret (S n))))
             , () ) )) l))
```

Figure 2.15: Description of Vec given a Constructor Tag

2.2 Synthesising descriptions to types

In Section 2.1 I had shown how it was possible to create descriptions that support many common datatypes in Idris. In this section I will present a way to convert or *synthesise* these descriptions to actual types, that allows the programmer to construct values of these described types with actual data.

2.2.1 A heterogeneous form of equality

Before we proceed with the actual synthesis I would like to present one of the important types that is needed. The type is called heterogeneous equality and is built into the core language theory of Idris. Heterogeneous equality takes two values of different types and constructors a type. In order to actually construct a value of such type (i.e.,

```
data (=): {a : Type} -> {b : Type}
          -> a -> b -> Type where
   Refl : {value : a} -> value = value
```

Figure 2.16: Heterogeneous Equality of Values

provide a proof for the equality), we have to use `Refl` which will only succeed if the compiler believes that such two values (and their types) are really the same. This provides a simple yet powerful way to constrain values in order to ensure that any given input must conform to some expected form.

2.2.2 Datatype synthesis

It is finally time to convert the description to an actual type. Figure 2.17 shows the `Synthesise` method which takes a description, the final form of that datatype and the resulting index, then it returns a type which can contain the described data.

- If we reach the end of the description, i.e. `Ret`, the only thing that we need to ensure is that the provided resulting index matches the expected index provided in the description. In order to apply such constraint we use the heterogeneous equality type (see Section 2.2.1).
- For recursive arguments, i.e. `Rec`, we construct a dependent pair where the first argument contains a value of the fully-synthesised type with the given index and the second argument contains the synthesised version of the rest of the provided description. The reason that we need the final form of the datatype in order to construct a recursive argument is due to the fact that if we call `Synthesise` recursively on that argument we would get stuck in an infinite loop!
- For ordinary arguments, i.e. `Arg`, we also create a dependent pair. The first argument of the dependent pair is a value `arg` of the provided type `a`, and the second argument is the synthesis of the rest of the provided description `d` given `arg`. This is isomorphic to how an ordinary constructor would store the data and as such the dependent pair serves a good target structure for our synthesis.

Since the dependent pair is used as the target type for the synthesis, it itself must be a core part of the type theory similarly to the heterogeneous equality, if we want to treat all datatype declarations as describable.

```
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)
```

Figure 2.17: Synthesising Descriptions into Actual Types

We are able to synthesise our descriptions to actual types using `Synthesise`; However, an issue occurs when we want to use it since it requires the final form of the described datatype as input but the only way to synthesise the datatype is using `Synthesise` itself. In order to "tie the knot" and complete input for `Synthesise`, we define a final datatype `Data` that takes a description and provides the final form of the described datatype (see Figure 2.18). `Data` has only one constructor namely `Con` which takes as input the synthesised version of the description `d` with `Data d` serving as argument for the final form of the datatype in `Synthesise`. This works since each time we face a recursive argument it must be constructed using `Con` which avoids a infinite loop in `Synthesise` as long as the elements that are constructed are smaller in size; Luckily, the typesystem of Idris ensures *totality* of functions and as such doesn't permit the creation of values that are infinitely sized except when the arguments are explicitly declared as such using the `Inf` type.

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

Figure 2.18: Knot-tying the Synthesised Description with Itself

2.2.3 Example: Constructing vectors

To get a more concrete intuition on how it is possible to construct data values of described types, this section will look at the synthesised version of Vec. Figure 2.19 shows Vec which is a function mimicking its corresponding type constructor, and it even shares the same type signature. The function Vec returns a described type using Desc passing along the description of Vec and its required parameters (*i.e.*, VecDesc a), and additionally the expected value of the result index (*i.e.*, n).

```
Vec : (a : Type) -> Nat -> Type
Vec a n = Data (VecDesc a) n
```

Figure 2.19: Synthesised Version of Vector Description

A simple example representing the vector [1, 2, 3] is presented in Figure 2.20. Although the value might seem a bit overwhelming at first, it follows a simple pattern: Each time a value of Vec is needed Con is used, then it is followed by its required arguments in the form of nested dependent pairs, and finally it ends with Refl which ensures that the provided index of the value match up with the expected one. As such, it can be observed that there are 4 occurences of Con and Refl in the example, three for the various applications of Cons and one for the application of Nil. For all cases the first two arguments represent the constructor label and associated tag. For Cons the first following argument is the length of the rest of the vector (the value of index n) followed by the value of the list head and the list tail, ending with Refl. For Nil the value is ended with Refl since it doesn't contain any data.

```
exampleVec : Vec 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)
               ))))
```

Figure 2.20: Example Vector representing “[1,2,3]“ as a Value of a Synthesised Description

As might have become apparent there are a couple of shortcomings in the example, or rather the way values of described types are constructed. One shortcoming is that there is a lot of boilerplate required when values are constructed, which makes the result somewhat unreadable. A solution to overcome that shortcoming is presented in the next paragraph. Another shortcoming is that the resulting terms seem to become very large (and in turn slows down program execution) compared to the original version of the datatype. For example, `Nil` becomes inflated to `Con ("Nil" ** (TZ ** Refl))` which is significantly more complex. A description on how to improve the size of resulting terms and speed up the performance of dependent programs is presented in Chapter 6.

In order to make creation of values of described types easier and the resulting terms more readable, it is possible to use functions as synonyms for the constructors. Figure 2.21 shows `Nil` and `Cons` as synonyms for the described version of `Nil` and `Cons` respectively.

```
Nil : {a : Type} -> Vec a Z
Nil = Con (“Nil” ** (TZ ** Refl))

Cons : {a : Type} -> {n : Nat}
       -> a -> Vec a n -> Vec a (S n)
Cons {n} x xs = Con (“Cons” ** (TS TZ ** (n **
                           (x ** (xs ** Refl)))))
```

Figure 2.21: Functions for Constructing Values of Synthesised Vector Description

Using the provided synonyms, the constructor of the value in example in Figure 2.20 becomes simpler and much more readable. Figure 2.22 shows the updated version, and it looks almost exactly like the original value it needed to describe `[1, 2, 3]`.

```
exampleVec : Vec 3 Nat
exampleVec = Cons 1 (Cons 2 (Cons 3 Nil))
```

Figure 2.22: More Readable Version of “[1,2,3]“ using previously-defined Aliases

2.3 The (mostly) gentle art of levitation

In the previous sections, I had presented a series of constructions that makes it possible to have described types. What may have become apparent for the user is that many of these constructions such as `Data`, `Tag` and `switch` cannot themselves be described since they are necessary dependencies for having descriptions. However, what might be surprising is that the description type itself `Desc`, isn't in fact limited by such constraint and can be described itself. This is the key point addressed by Chapman et al. (2010) in "The gentle art of levitation".

The description datatype `Desc` contains many of the constructors needed to describe itself. However, one might experience trouble when trying to describe `Arg` since it requires an argument of the following type: `(a ->Desc ix)`. This argument describes a function which result type is the datatype itself (a so-called higher-order inductive argument), which `Rec` isn't strong enough to express since it only permits recursive values. Figure 2.23 shows a new constructor `HRec`, which allows specification of such higher-order inductive arguments taking a type `a` which specifies the expected input type of such argument in addition to the rest of arguments that are expected by `Rec`.

```
HRec  : ix -> (a : Type) -> Desc ix -> Desc ix
```

Figure 2.23: A constructor for `Desc` to represent higher-order recursion

Similarly to the other constructors of `Desc`, `Synthesise` must be extended with a clause for `HRec`. Figure 2.24 shows the corresponding clause, which looks very similar to the one for `Rec`, except the first component now requires a function from the provided type `a` to the datatype itself instead of just a reference to the datatype.

```
Synthesise (HRec j a d)    x i =
   (rec : a -> x j ** Synthesise d x i)
```

Figure 2.24: Synthesising `HRec` to a real type

Finally, all the required constructors are present and it is now possible to piece together a description for `Desc`. Figure 2.25 shows the complete description, including for the newly added `HRec` constructor. The description of `Desc` is parametrised by the type of indices `ix` that possible derived descriptions can have, and is not indexed by anything particularly interesting (the unit type `()` is used in the figure). For each constructor, it matches the original definition presented Figure 2.8. The only interesting case is the one for `Arg`, which uses `HRec () a (Ret ())` to represent the higher-order inductive argument (`a -> Desc ix`).

```
DescDesc : (ix : Type) -> Desc ()
DescDesc ix =
  Arg CLabel (\l =>
    Arg (Tag l ["Ret", "Arg", "Rec", "HRec"])
      (switchDesc
          (Arg ix (\i => Ret ()),
          (Arg Type (\a => HRec () a (Ret ())),
          (Arg ix (\i => Ret ()),
          (Arg ix (\i => Arg Type (\a => Rec () (Ret ()))),
      ())))) l))
```

Figure 2.25: Describing the `Desc` datatype itself

Perhaps, the key thing to notice is that a function `switchDesc` was used instead of `switch` when describing `Desc`. Figure 2.26 shows how to `switchDesc` is defined by specialising `switch`. However, if `Desc` is a described type based on `DescDesc` then such definition would be circular. This is because the general `switch` requires the result type `Desc` to be given as argument, but `Desc` is dependent on `DescDesc`. Therefore, Chapman et al. (2010) define `switchDesc` specially in their type theory, eliding the definition of the body and hard-wiring its return type to be `Desc`[2]. Now, `DescDesc` can be type checked without any issues and *levitation* is achieved.

```
switchDesc : {e : CEnum} -> {ix : Type}
    -> SPi e (\l => \t => Desc ix)
   -> ((l' : CLabel) -> (t' : Tag l' e) -> Desc ix)
switchDesc {e = e} {ix = ix} cs =
                     switch e (\l => \t => Desc ix) cs
```

Figure 2.26: A specialised version of `switch` which returns descriptions

[2]Of course, such trick only works if there is know to be a type `Desc` in the meta-theory. However, the knowledge of its elements is not necessarily required and it is possible to inspect these in a similar fashion to other datatypes.

Chapter 3

Partial evaluation

3.1 Functions and constant inputs

General introduction about partial evaluation.

3.2 Binding-time analyses of programs

Finding the relevant constant parts of the program.

3.3 Specialisation as a form of optimisation

Performance benefits of program specialisation. Pitfalls.

Chapter 4

Levitating Idris

4.1 A pragmatic implementation of levitation

How the general concept of levitation was transferred to Idris.

4.2 Description synthesis from ordinary data declarations

How ordinary data-declarations are synthesized to levitational descriptions.

Chapter 5

Practical examples

5.1 Generic deriving

Examples of generic deriving of algorithms like decidable equality, pretty printing and possibly eliminators via generic structure.

5.2 Uniplate for Idris

A version of the Uniplate library for Idris based on `http://community.haskell.org/~ndm/uniplate/` *and* `http://www-ps.informatik.uni-kiel.de/~sebf/projects/traversal.html` *. This is useful for traversing structures in a generic fashion and especially when dealing with small changes in large data structures (such as compiler ADTs)*

Chapter 6

Optimising Idris for flight

6.1 Specialising constructors for specific types

How generalized constructors of described types, are specialised as ordinary data structures.

6.2 Online erasure of unused arguments

How some type infromation is to be erased at compile time to reduce elaboration overhead. Very hypothetical.

6.3 Static initialization of generic functions

How algorithms that are dependent on the generic structure of a datatype are optimised. Discuss benefits of having a JIT/Profiling information for future work.

Chapter 7

Evaluation

Chapter 8

Discussion

8.1 Future work

8.2 Conclusion

Bibliography

Benke, M., Dybjer, P. & Jansson, P. (2003), 'Universes for generic programs and proofs in dependent type theory', *Nordic Journal of Computing* **10**(4), 265–289.

Brady, E. (2013), 'Idris, a general-purpose dependently typed programming language: Design and implementation', *Journal of Functional Programming* **23**(05), 552–593.

Chapman, J., Dagand, P.-E., McBride, C. & Morris, P. (2010), The gentle art of levitation, *in* 'Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming', ICFP '10, ACM, New York, NY, USA, pp. 3–14.
URL: *http://doi.acm.org/10.1145/1863543.1863547*

Diehl, L. & Sheard, T. (2014), Generic constructors and eliminators from descriptions, WGP '14.

Magalhães, J. P., Dijkstra, A., Jeuring, J. & Löh, A. (2010), 'A generic deriving mechanism for haskell', *ACM Sigplan Notices* **45**(11), 37–48.

McBride, C. (2010), 'Ornamental algebras, algebraic ornaments', *Journal of Functional Programming* .

Norell, U. (2009), Dependently typed programming in agda, *in* 'Advanced Functional Programming', Springer, pp. 230–266.

Peyton Jones, S. et al. (2003), 'The Haskell 98 language and libraries: The revised report', *Journal of Functional Programming* **13**(1), 0–255. `http://www.haskell.org/definition/`.