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 an element of any algebraic data type follows a very simple procedure:

1. Print the name of the constructor

2. Iterate over the constructor arguments and pretty print them, with each argument surrounded by parentheses if necessary

   a) If the argument 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, then

      i. Find the correct pretty printing function for that type
      ii. Pretty print the field using the found function

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) it is possible to create a correct description using ordinary datatype definitions (Benke, Dybjer, and Jansson 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. 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 the end, 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 goals.

The first goal 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 goal 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-style (SYB) library for generic querying and traversal.

The final goal is that I seek to describe how partial evaluation and related techniques can be used 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 (Jones, Gomard, and Sestoft 1993) and constructor specialisation (Mogensen 1993).

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 I implemented described types in Idris and continues with practical examples in Chapter 5. Chapter 6 presents the optimisations that can be 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 the challenges that 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 lists 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], unlike other dependently-typed languages like Agda which have a syntactic distinction.

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`. The 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`). The constructor `Cons` contains three different types of arguments: an ordinary implicit argument,

---

[1] If an argument to a type constructor doesn't change in the data constructor declarations, Idris considers it a parameter, otherwise an index.
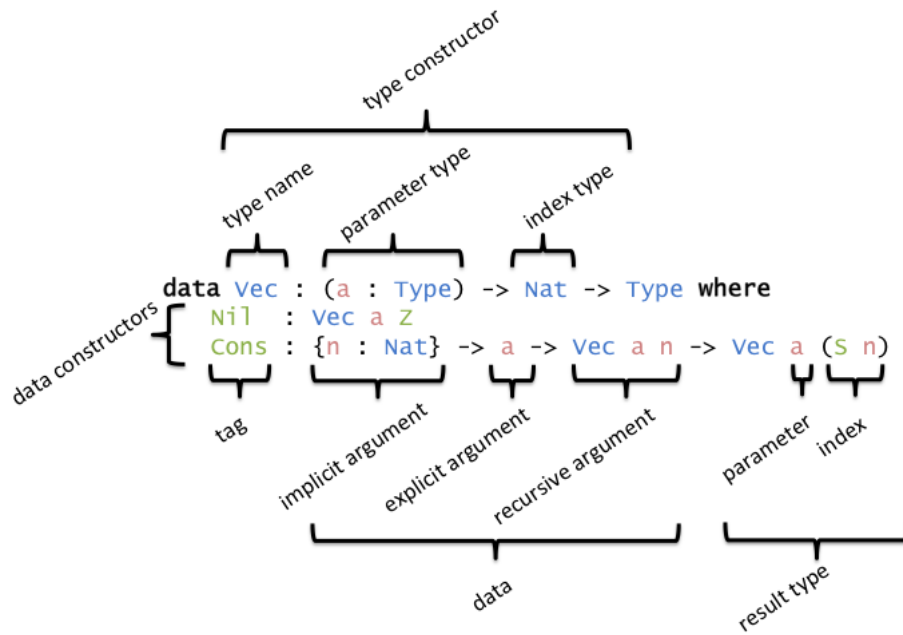
Figure 2.1: Annotated components of a datatype Declaration

an ordinary explicit argument, and an explicit argument of the type it-self (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.)

Bold Black is used for keywords (if, data, etc.)

2.1.2   A description for datatypes

It is now possible to try to represent a suitable datatype for descriptions. Figure 2.8 presents one possible solution, influenced mainly by the work of McBride (2010) and Diehl and Sheard (2014). Similarly based on that

work, Section 2.2 will later present how to construct actual described types from these descriptions.

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

Figure 2.2: A datatype that describes other datatypes

The description datatype has three main constructors:

- Constructor `Ret` represents the end of a description

- 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 providing a side-by-side comparison of 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 the sole constructor `MkUnit` doesn't contain any arguments, `Ret` is used to simply end the description.

```
data Unit : Type where              UnitDesc : Desc
    MkUnit : Unit                   UnitDesc = Ret
```

(a) Declaration of `Unit`   (b) Description of `Unit`

Figure 2.3: The `Unit` datatype and its description

Constructor arguments 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 would be of the form `Arg a (\arg => b)` Finally, to specify the end of the description, `Ret` is used.

```
data Pair : Type where          PairDesc : Desc
    MkPair : (fst : Int) ->      PairDesc = Arg Int (\fst =>
              (snd : Bool) -> Pair          Arg Bool (\snd => Ret))
```

(a) Declaration of `Pair`   (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.

Choice of constructors Since there is no explicit way to encode a choice between multiple constructor in the provided description, instead, a boolean argument `isRight` is used as a tag to determine which constructor is described. 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         EitherDesc : Desc
    Right : (x: Int)      ->     EitherDesc = Arg Bool (\isRight =>
            Either                   if isRight
    Left  : (x: String) ->           then Arg Int (\x => Ret)
            Either                   else Arg String (\x => Ret))
```

(a) Declaration of `Either`   (b) Description of `Either`

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

Recursive arguments   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          NatDesc : Desc
    Zero : Nat                 NatDesc = Arg Bool (\isZero =>
    Succ : Nat -> Nat             if isZero
                                  then Ret
                                  else Rec Ret)
```

(a) Declaration of `Nat`   (b) Description of `Nat`

Figure 2.6: The Natural numbers (`Nat`)

Parameters   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 function describing the particular datatype, which allows them to be qualified over the whole description. The description 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`. Like the description for `Zero`, the description for `Nil` is simply `Ret` since it doesn't accept any arguments. 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 then ends the description.

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

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

(a) Declaration of `List`   (b) Description of `List`

Figure 2.7: A polymorphic list of elements

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 of the result type and recursive argument must be respectively. Descriptions that don't require indices, they can be converted to indexed descriptions by 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 new look at `Vec` from Figure 2.1. Figure 2.9 shows the corresponding description of `Vec` with comparable annotations.

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`, which 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` representing the length of the tail, i.e. the index of the recursive
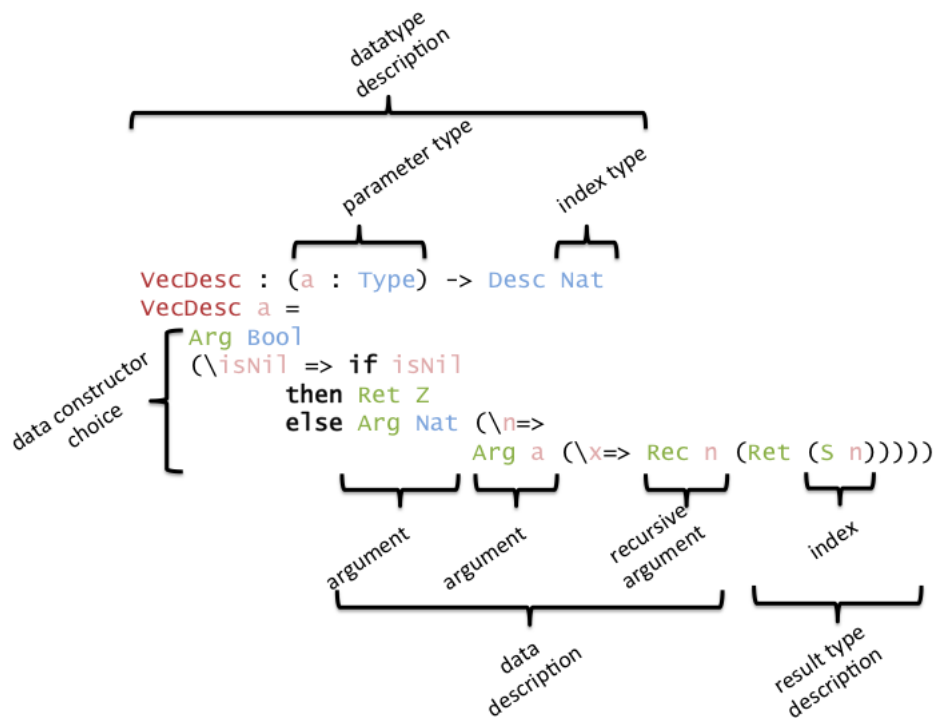
Figure 2.9: Described version of `Vec`

argument, and an argument of the parameter type `a` representing the head. Following these arguments we take a recursive argument representing the tail—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`, just as in Figure 2.1.

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 answer 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 cap-

ture 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 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 of its index.

Finally, since parameters are usually quantified 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 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. For example, 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 (heavily inspired by Dagand (2013); Diehl and Sheard (2014)) as shown in Figure 2.10a: `CLabel` which represents a name for

a constructor, and `CEnum` which represents a list of constructor names. For the sake of simplicity, the provided constructors in a `CEnum` are assumed to be provided uniquely by the user, however one could stipulate such uniqueness condition explicitly if desired. Figure 2.10b show an example of how to represent the available constructor names of `Vec`.

```
CLabel : Type
CLabel = String

CEnum : Type                      VecCtors : CEnum
CEnum = List CLabel               VecCtors = [ "Nil" , "Cons" ]
```

    (a) Representation   (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 makes it possible to use tactics in Idris to automate the retrieval of a tag given a constructor label; something which saves time when constructing values 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 tag, and as therefore it is a good representative for enumerating the constructors of `Vec`.

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

       (a) Tag for `Nil`   (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 to suitable values of a desired type. The following section will describe the `switch` function that does exactly this.

Since the count of constructors for a datatype can vary in size, 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 is essentially a function that provides a one-to-one mapping from the list of constructors to a series of right-nested pairs ending 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 dependencies 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 `switch` which can look up the corresponding result value in the map 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). Since there can be no value of `Tag` on an empty enumeration of constructors, it is not required to handle that case.

```
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

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.

```
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

In summary, while the description might initially seem more complicated than before, it has a couple of clear advantages: the encoding now contain the constructor tag and it is possible to choose between more than two constructors at the same time.

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   Datatype synthesis

It is finally time to convert the description to an actual type. Figure 2.16 shows the `Synthesise` function 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 propositional equality type.

- 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 propositional equality, if we want to treat all datatype declarations as describable.

    We are able to synthesise our descriptions to actual types using `Synthesise`. However, an issue occurs when we want to use `Synthesise`

```
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 2.16: Synthesising Descriptions into Actual Types

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 2.17). `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 in-
finite loop in `Synthesise` as long as the elements that are constructed
are smaller in size.

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

Figure 2.17: Knot-tying the Synthesised Description with Itself

2.2.2 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 synthe-
sised version of `Vec`. Figure 2.18 shows `Vec` which is a function mim-
icking 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`).

   A simple example representing the vector `[1 , 2 , 3]` is presented
in Figure 2.19. Although the value might seem a bit overwhelming at

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

Figure 2.18: Synthesised Version of Vector Description

first, it follows a simple pattern: each time a value of Vec is needed Con is used, followed by its required arguments in the form of nested dependent pairs, and finally with Refl, which ensures that the provided index of the value matches up with the expected one. There are 4 occurences of Con and Refl in the example, three for Cons and one for 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 2.19: 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 become very large, and in turn slowing 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 syn-

onyms for the constructors. Figure 2.20 shows `Nil` and `Cons` as syn-
onyms for the described version of `Nil` and `Cons` respectively. Since
Idris can infer the values of `a` and `n` automatically in this context, they
are converted to implicit parameters in these synonyms (which further
increases readability).

```
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 2.20: Functions for constructing values of synthesised vector de-
scription

Using these synonyms, the constructor of the value in example in
Figure 2.19 becomes simpler and much more readable. Figure 2.21
shows the updated version, and it looks almost exactly like the origi-
nal value it needed to describe `[1, 2, 3]`.

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

Figure 2.21: More Readable Version of `[1 , 2 , 3]` using previously-
defined Aliases

2.3   The (mostly) gentle art of levitation

The previous sections 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 de-
pendencies for having descriptions. However, what might be surprising
is that the description type `Desc` itself, isn't in fact limited by such a con-
straint and can be described using 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 2.22 shows a new constructor `HRec`, which allows specification of such higher-order inductive arguments, it takes 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 2.22: A constructor for `Desc` to represent higher-order recursion

The function `Synthesise` must be extended with a clause for `HRec`. Figure 2.23 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 2.23: 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 2.24 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). The description for each constructor is translated using the same techniques presented in Section 2.1. The only interesting case is the one for `Arg`, which uses `HRec () a (Ret ())` to represent the higher-order inductive argument `(a -> Desc ix)`.

Perhaps, the key thing to notice is that a function `switchDesc` was used instead of `switch` when describing `Desc`. Figure 2.25 shows how `switchDesc` is defined by specialising `switch`. However, if `Desc` is

```
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 2.24: Describing the Desc datatype itself

a described type based on DescDesc then such a definition would be
circular. This is because the general switch requires the result type
Desc to be given as an argument, but Desc is dependent on DescDesc.
Therefore, Chapman et al. (2010) define switchDesc to be handled
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 2.25: 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   The static nature of programs

It hardly comes as a surprise that for many programs, not all of their
input might be dynamic. There are many reasons to structure programs
like readability, reusability and debuggability, even if all the input is
static. For example, writing `minutesInADay` = `24` * `60` is usually seen
as more preferable to writing `minutesInADay` = `1440`, since it better
captures the intent of the programmer.

However, in many cases it is desirable to only compute static for-
mulas once at compile time instead of suffering a performance loss at
runtime every time the program is run. One technique for static com-
putation of programs, is called *partial evaluation* (Jones, Gomard, and
Sestoft 1993). A program that does partial evaluation `mix` is called a
partial evaluator, and accepts as input another program (often called the
object program) and a series of static input for that particular program.
The result of the partial evaluator `mix` is a new program *specialised* with
regards to the specified input.

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.

```
square : Int -> Int
square x = x * x

power : Int -> Int -> Int
power x n =
   if (n `mod` 2) == 0
   then square (power x (n `div` 2))
   else x * power x (n - 1)
```

Figure 3.1: PowerExample

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 print-ing 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, Marcin, Peter Dybjer, and Patrik Jansson (2003). "Universes for Generic Programs and Proofs in Dependent Type Theory". In: *Nordic Journal of Computing* 10.4, pp. 265–289.

Brady, Edwin (2013). "Idris, a general-purpose dependently typed programming language: Design and implementation". In: *Journal of Functional Programming* 23.05, pp. 552–593.

Chapman, James et al. (2010). "The Gentle Art of Levitation". In: *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming*. ICFP '10. Baltimore, Maryland, USA: ACM, pp. 3–14. ISBN: 978-1-60558-794-3. DOI: 10.1145/1863543.1863547. URL: http://doi.acm.org/10.1145/1863543.1863547.

Dagand, Pierre-Évariste (2013). "A Cosmology of Datatypes Reusability and Dependent Types". PhD thesis.

Diehl, Larry and Tim Sheard (2014). "Generic Constructors and Eliminators from Descriptions". In: WGP '14.

Jones, Neil D., Carsten K. Gomard, and Peter Sestoft (1993). *Partial Evaluation and Automatic Program Generation*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc. ISBN: 0-13-020249-5.

Magalhães, José Pedro et al. (2010). "A generic deriving mechanism for Haskell". In: *ACM Sigplan Notices* 45.11, pp. 37–48.

McBride, Conor (2010). "Ornamental algebras, algebraic ornaments". In: *Journal of Functional Programming*.

Mogensen, Torben Æ (1993). "Constructor Specialization". In: *Proceedings of the 1993 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation*. PEPM '93. Copenhagen, Denmark: ACM, pp. 22–32. ISBN: 0-89791-594-1. DOI: 10.1145/154630.154633. URL: http://doi.acm.org/10.1145/154630.154633.

Norell, Ulf (2009). "Dependently typed programming in Agda". In: *Advanced Functional Programming*. Springer, pp. 230–266.

Peyton Jones, Simon et al. (2003). "The Haskell 98 Language and Libraries: The Revised Report". In: *Journal of Functional Programming* 13.1. `http://www.haskell.org/definition/`, pp. 0–255.