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 booleans, lists or trees form a core part of modern functional programming. Most functions written work directly on such datatypes, but sometimes the function we write aren't directly dependent on the datatype itself and serve only as a repetitive exercise such as for structural equality or pretty printing. In fact it seems that 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.

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, there has traditionally been issues regarding usability in ordinary functional languages such as Haskell. To represent the description, it is often required to use special language extensions and the programming style tend to be orthogonal to writing ordinary programs.

However, in dependently-typed languages such as Idris or Agda it seems that it 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. Furthermore, Chapman et al. (2010) how it is possible to build a self-supporting type system which is able to synthesise descriptions to ordinary types while still being powerful enough to describe the description datatype itself. Therefore in such system, generic programming is not different from ordinary programming in such system.

1.2 Problem Definition

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

Yet in many ways, 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 incompatible descriptions are often presented in the current field, sometimes even in the same paper, which is not 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 and as such both type checking and runtime performance is 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 when writing repetitive functions.

1.3 Aim and Scope

The main aim of this research is to provide a practical and efficient implementation of described types in Idris.

One key part is to find a good definition of the description that supports many common datatypes. I seek to mainly base my effort on reusing 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. Another key part is to present realistic examples using generic functions by implementing functions that can be used to derive type class instances and a Scrap Your Boilerplate library for generic querying and traversal. In the end, I seek to use partial evaluation to optimise the generic functions with regards to specific descriptions to try to achieve near hand-written performance metrics.

1.4 Significance

The main contributions of my research 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 what challenges 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 that show how generic programming using described types is a viable option to reduce boilerplate without significant cost in performance

1.5 Overview

I will present the report as follows. In Chapter 2, I will present an introduction to described types specifically focusing on recent developments using dependently-typed programming languages. In Chapter 3, I will present an overview of techniques for partial evaluation of functions and specialisation of datatypes. In Chapter 4, I will discuss specifically how described types are implemented in Idris and I will continue with the practical examples in Chapter 5. In Chapter 6 I will present what optimisations were made in order to improve the runtime performance of described types and generic functions. I will evaluate the results in Chapter 7, comparing the performance of generic implementations to hand-written ones. Finally, I will discuss what challenges still lie ahead and conclude in Chapter 8.

Chapter 2

Generic Programming

2.1 The Generic Structure of Inductive Data Types

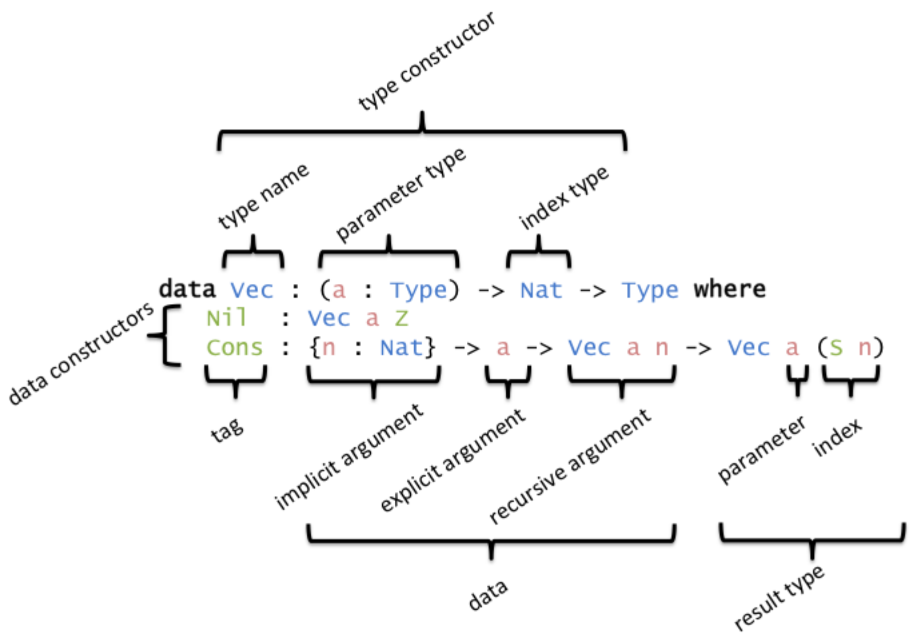
2.1.1 Anatomy of a Datatype

In order to understand how a description might look like, let us first start by looking closely at how datatypes are structured. In Figure 2.1, I will present 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 consists of 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 difference between a parameter type or an index type since Idris figures that out automatically.[1]

Similarly to the type constructor, data constructors need names too, also called *tags*. 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`. `Nil` doesn't hold any data, so it only needs to define the resulting type which is `Vect a Z` (a vector with length 0). `Cons` contains three different types of arguments: an ordinary implicit argument, an ordinary explicit argument, and an explicit argument to the type itself (recursive); the resulting type

[1]If the argument to a type constructor doesn't change in the data constructor declarations Idris considers that a parameter, and 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

for `Cons` is `Vect a (S n)` (i.e. a vector of length 1+n where n is the length of the recursive argument).

2.1.2 A Description for Datatypes

It is now possible to try to represent a suitable description datatype. Figure 2.2 presents one possible solution, influenced mainly by the work of McBride (2010) and Diehl (2014).

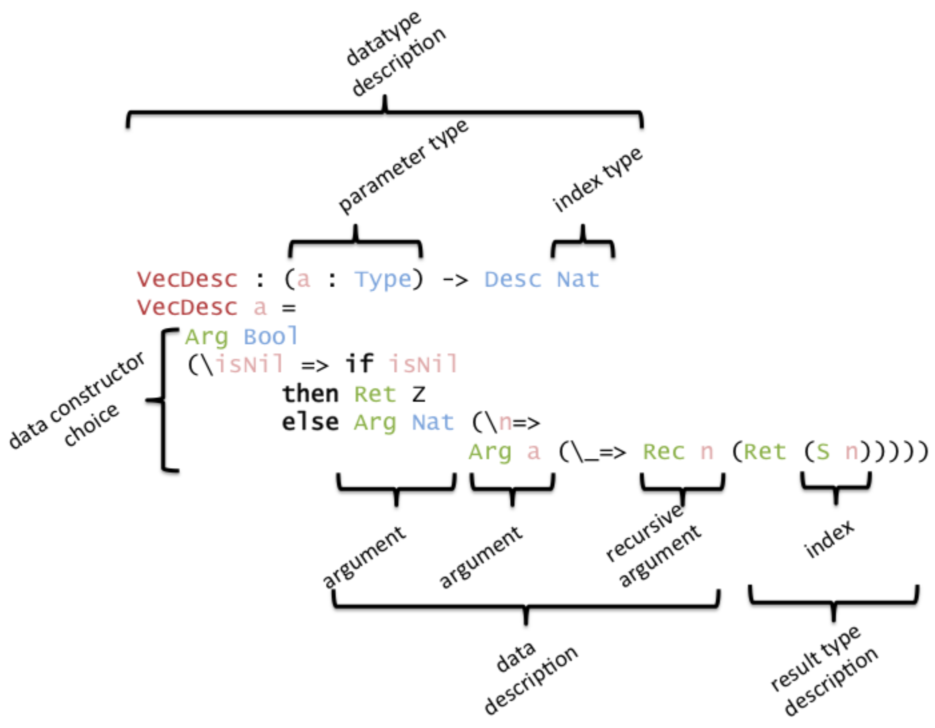
```
data Desc : (index : Type) -> Type where
    Ret  : index -> Desc index
    Arg  : (a : Type) -> (a -> Desc index) -> Desc index
    Rec  : index -> Desc index -> Desc index
```

Figure 2.2: Datatype for describing other Datatypes

In order to allow datatypes to be indexed by values of various types, the description structure takes a parameter that describes what the type of indices must be. Additionally, the description has three constructors:

- `Ret` represents the end of a description and takes as argument what value the index of the resulting type must be

- `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.
- `Rec` represents a recursive argument of the described datatype and take two arguments: the index of the recursive instance of the datatype and the rest of the description.



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

Figure 2.3: Described version of Vec

The astute reader might at this point have some questions such as: how is it possible to choose between different constructors? where do parameters go in the description? and why is there only one type for indicies? I will now try to answer these questions using Figure 2.3.

One step towards building a value that describes a particular datatype is defining its type signature including what types the parameters and indices of the finished datatype are. In our example of Figure 2.3, the parameter is specified to be given as an argument to the description value and is not represented internally as such in the final description, i.e., it is not possible to distinguish a parameter from other values when the final description is constructed. Contrary to parameters, the type of indices must be specified as an internal part of the description in order to ensure that indices specified by recursive arguments match in the final description. Since `Desc` only takes one argument for specifying the types of indices, more demanding datatypes will need to collect multiple indices into one dependent pair, e.g., a datatype with signature `(n : Nat) -> Fin n -> Type` must use the dependent pair `(n : Nat ** Fin n)` as the type for its index.

Another step required for describing a datatype is defining how to choose a particular constructor. A solution could be the one presented in Figure 2.3 where a boolean argument `isNil` is used to decide whether the resulting description should be for the `Nil` or `Cons` constructors. However, since this solutions is fairly primitive and doesn't capture information such as constructor tags, a better but more complex encoding will be presented in Section 2.1.3.

Finally, the constructors themselves must be described. `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. `Cons` takes first two ordinary arguments: a `Nat` argument[2] 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 `Vect a n`. We finish the description with `Ret` and specify that the resulting index must be `S n` as expected.

2.1.3 An Informative Encoding of Constructors

In Section 2.1.2 I had presented the choice of constructors using a boolean variable and concluded that it was both primitive and lacked necessary information such as the tag of a particular constructor. I will in this section present a more informative encoding of constructors, and show how it is possible to use that encoding when describing datatypes.

In order to represent which constructors are available we first are going to declare two types as shown in Figure 2.4a: `CLabel` which represents a name for a constructor, and `CEnum` which represents a list

[2]Implicit arguments is a feature of Idris that only works on top-level declarations and as such are all specified explicitly in the description.

of constructor names. Figure 2.4b 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 2.4: Constructor Labels

Now that it is possible to represent the available constructors, we can encode a way of choosing a particular constructor tag. Figure 2.5 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 is valid index into a (non-empty) list of constructor tags.

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

Figure 2.5: Tags: A Structure for Picking a Constructor from a Label Collection

2.1.4 A Constructive Type of Choice

Given a specific constructor tag, one question is then how it is possible to map these constructor tags to specific values? Figure 2.6 presents π (small pi operator) which represents the type of a map from a list of constructor tags to a corresponding list of values. The calculated type might look atypical at first, however on a closer look it is actually analogous to how lists are represented in Lisp. The calculated type consists of nested pair types where the first element is the type of the value mapped for the current constructor and the second element is the mapping for the rest of the constructors; finally, the list ends with the unit type `()` similarly to `nil` in Lisp.

The `switch` function shown in Figure 2.7 uses a specific constructor tag to lookup the corresponding value given by the π mapping. When

```
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 2.6: The Small Pi Operator: Type for Case Analysis based on Constructor Tags

the given π mapping is given inline switch works like a version of the `case of` operator on constructor tags, which can be useful when specifying descriptions for a particular constructor.

```
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 2.7: Calculation of a Property based on a Specific Constructor Tag

As an example, Figure 2.8 shows the description of `Vec` 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 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 (\_,_=> Desc Nat)
              ( Ret Z
              , (Arg Nat (\n=>
                         Arg a (\_=> Rec n (Ret (S n))))
              , () ) )) l))
```

Figure 2.8: 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 in Idris, 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. What the

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

Figure 2.9: Heterogeneous Equality of Values

heterogeneous equality type allows us to do is that it allows us to construct a type that equate values of different types. However in order to actually construct a value of such type (i.e., provide a proof for the equality), we have to use `Refl` which will only succeed if the compiler believes that such two values 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 2.10 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 index -> (index -> Type) -> (index -> 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 2.10: 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` in the core theory which takes a description and provides the final form of the described datatype (see Figure 2.11). `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 : {index : Type} -> Desc index
             -> index -> Type where
   Con : {d : Desc index} -> {i : index}
          -> Synthesise d (Data d) i -> Data d i
```

Figure 2.11: Knot-tying the Synthesised Description with Itself

2.2.3 Example: Constructing Vectors

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

Figure 2.12: Synthesised Version of Vector Description

```
exampleVec : Vec 3 Nat
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 2.13: Example Vector representing "[1,2,3]" as a Value of a Synthesised Description

2.3 The (Mostly) Gentle Art of Levitation

The elegance of a complete theorem for both ordinary and generic programming. Highlighting of possible issues with performance.

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

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

Figure 2.14: Functions for Constructing Values of Synthesised Vector Description

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

Figure 2.15: More Readable Version of "[1,2,3]" using previously-defined Aliases

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 Optimization

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

Optimizing 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 optimized. 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

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. (2014), 'Modeling elimination of described types'.
URL: *http://spire-lang.org/blog/2014/01/15/modeling-elimination-of-described-types*

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