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.

Contents

Chapter 1

Introduction

1.1 Context

Tradtionally, a core part of functional programming is the idea of algebraic datatypes such as booleans, natural numbers, lists, trees, and other types of inductive structures. These datatypes are often simple to work with since the possible cases are known, and the inherent design towards purity ensures that most functions stay declarative which makes programs easier to reason about.

While it is often the case that functions must be tailored specifically to a certain datatype, in certain cases it often seems as we are repeating ourselves such as in the cases of boolean structural equality or textual conversions. In fact it almost seems like we could instruct a computer to write the function for us, or rather it seems that it is possible to write a function over the structure of the datatype definition which the computer could use to derive actual function for particular datatypes.

Enter the world of *generic programming*, or colloquially *functional metaprogramming*, where the target data is the datatype describing the structure of other datatypes which is often called the *description*. While generic programming indeed sounds promising, there has usually been issues regarding usability and performance in traditional functional languages such as Haskell. First of all, generic programming often requires special language extensions to encode the datatype description which further complicates things; otherwise, the description might be constructed incorrectly and require the programmer to handle weird edge cases. Secondly, there is often some kind of discrepancy between ordinary programming and generic programming with generic programming almost requiring an orthogonal type-level style of programming; thus making it hard for the ordinary programmer to exploit. Finally, due to the way generic programs are applied abstractly to datatypes instead of fitted specifically, there is often a large amount of overhead which sometimes cannot be optimised away since it is hard to safely perform arbitrary function transform; hence, generic programming usually is not considered an option for performance critical applications.

In dependently-typed languages such as Idris or Agda however, it seems that many of the issues can be mitigated. Particularly, due to the nature of dependently-typed languages and the Curry-Howard isomorphism, it is possible to encode proofs in datatypes, and as such design the datatype description such that all values of that type are correct by construction. More interestingly, such as the one shown by Dagand et al. in 'The Gentle Art of Levitation' it is possible to build a complete type-system which is able to convert values of the descriptions to ordinary types and is even powerful enough to describe the datatype description itself. As such, in the Levitation-system generic programming can be achieved fully using ordinary programming techniques.

1.1.1 Anatomy of a Datatype

A question the reader might have now is then how does a datatype description look, since that might not be completely obvious. To better understand however, let us start by investigating what a typical datatype is composed by.

In Figure 1.1, I present an annotated version of a typical dependently-typed datatype that represents length-indexed lists, or vectors. While many things might initially seem trivial, it is nonetheless important to get a precise definition on how a datatype is defined; especially, when that is the target structure for our future algorithms.

To describe the type constructor of a datatype, three components are needed: the name of the datatype, types of any possible parameters, and finally, since this is a dependently-typed language, the types of indicies of the datatype. In the figure it is notable that it seems there is no difference between a parameter type or an index type, but that is because Idris figures this out automatically (unlike other dependently-typed languages).[1]

[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.

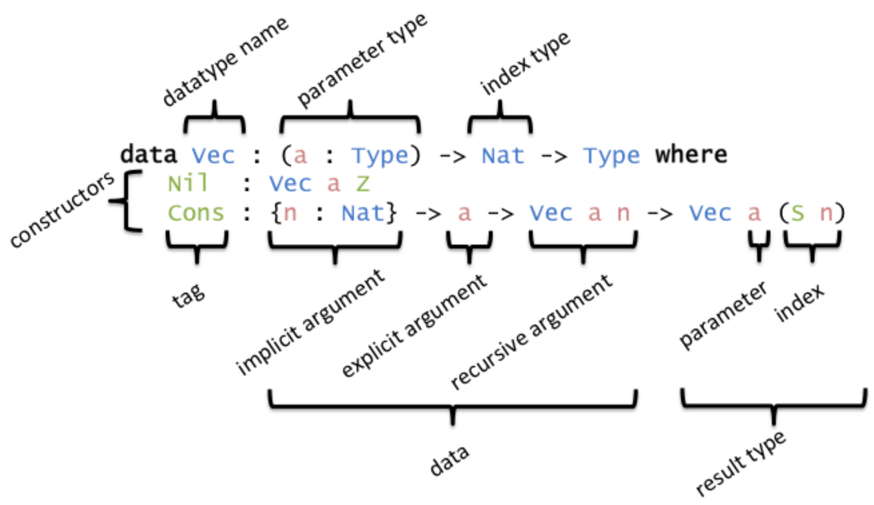

Figure 1.1: Annotated Components of a Datatype Declaration

In addition to the type constructor, most datatypes contain one or more data constructors which provides a way to store the actual data. Similarly to a type constructor, a data constructor also needs a name, also called the *tag* of the constructor. Following the tag, the data constructor declaration contains the types of the arguments stored in the constructor and resulting type which must be the same as the datatype we are currently declaring. In our example two constructors are declared, `Nil` and `Cons`. Since `Nil` doesn't hold any data, the declaration only shows the resulting type which is `Vect a Z` (i.e. it constructs a list of length zero). `Cons` is more interesting since it contains three different types of arguments: an implicit arguments of a different type, an explicit arguments of a different type, and an explicit argument to the type itself (recursive argument), with the resulting type being `Vect a (S n)` (i.e. it constructs a list of length one greater than the input recursive argument). As the reader might have noticed the index of the resulting type of `Nil` is different from the index of the resulting type of `Cons`, while the parameters are seamingly the same.

1.1.2 A Description for Datatypes

After understanding how datatypes are structured, it is now possible to try and construct a suitable description datatype. Figure 1.2 presents one possible solution, influenced mainly by the work of McBride and Diehl.

```
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 1.2: Datatype for describing other datatypes

The type constructor for the description takes only a single parameter which specifies the type of a possible index for the datatype to describe. The description datatype has three constructors `Ret`, `Arg` and `Rec`. `Ret` represents the end of a description and takes an index as its argument describing what the index of the resulting type of constructor would be; a particular interesting use of `Ret` is for describing empty constructors such as `Nil` in Figure 1.1. `Arg` represents an addition of an argument to a given description and can be used for describing arguments of any type; the first argument of `Arg` is the type of argument expected and the second argument is the rest of the description given a value of the expected type which allows us to use the value as a dependency in the resulting type. Finally, `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 1.3: Described version of Vec

The astute reader might have some questions at this point 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 1.3.

Since we are working in a dependently-typed language, it is possible to encode the choice between constructors by taking one or more multiple arguments and then return the desired description depending on what value is received. For example, in Figure 1.3 we take a boolean argument `isNil` deciding whether the resulting data should be a `Nil` or `Cons` and thereafter returning the fitting description. As this is a fairly primitive encoding, I will present a better but more complex encoding in Section 2.1 which permits choosing a constructor using its tag.

Regarding parameters, they might not necessarily be required to be encoded in the description datatype itself since they do not change value. Instead, they may merely be given as function arguments when writing the description of a particular datatype such as in the case of Figure 1.3.

Finally for the index, only one type is required since we can represent arbitrary number of index arguments using a dependent pair by uncurrying. For example, for a type constructor with signature `(n : Nat) -> Fin n -> Type`, i.e. with indices `Nat` and `Fin n`, it is possible to just uncurry the indicies such that they are `(n : Nat ** Fin n)` and so forth for more complicated indicies.

Now that the interesting questions has been answered, let us in the end try and get an overview of Figure 1.3. As mentioned, the description starts with a boolean argument `isNil` that we use to check for which constructor we want to create, i.e. either `Nil` or `Cons`. If we are trying to construct a `Nil` which doesn't contain any data then we simply use `Ret` stating that the description is finished with the index of the resulting type being `Z`, analogously to Figure 1.1. In the case of `Cons` we first take a `Nat` argument to be used for the necessary indices, similarly to Figure 1.1 but this time explicitly.[2]. Afterwards we take an argument of the input parameter type `a`, and a recursive argument—this time specified using `Rec` instead of `Arg`—that must satisfy to have `n` as index, i.e. the argument must be of type `Vect a n`. Finally we finish the description with `Ret` and specify that the resulting index must be `S n` as expected.

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. Yet in many areas the work that has

[2]Implicit arguments is a feature of Idris that only works on top-level declarations

been done is mostly theoretically oriented, which can lead to some issues when used in a practical manner. First of all, many authors often use multiple incompatible descriptions depending on what property they would like to show; however this is not particularly attractive in a practical setting, since this could potentially confuse the generic programmer and make it harder for the compiler to handle transformations from generic programs to ordinary programs, and vica versa. Secondly, there has been little work on how to integrate such descriptions in more practical languages such as Idris which contains many modern language features such as Type Classes where some operations might be hand-written on some substructures and may not exist at the time of defining a particular generic function. Finally, datatypes synthesised from descriptions create large canonical terms and as such both type checking and runtime performance is very slow.

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 therefore to find a good definition of the description datatype that supports many of the commonly used datatypes in Idris. I seek to mainly base my effort on reusing some of the existing work on descriptions, and not try to extend support for other inductive families such as inductive-inductive and inductive-recursive datastypes; neither will I focus on supporting all of the vast language features of Idris such as implicit arguments or codata definitions. Another key part is to present realistic examples using generic functions by implementing a mechanism for type class deriving 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 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 implementation of a mechanism for deriving type class instances in Idris, with examples for deriving decidable equality, pretty printing and functors.
- a discussion of 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 that show how generic programming using described types is a viable option to reduce boiler-plate 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

How Generic Programming generally works.

2.2 The Importance of Genericity in Dependently-typed Languages

The similarity of structure and various slightly-different indexing of types.

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.

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