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 tree form a core part of modern functional programming. While most functions work directly on such datatypes, sometimes it is the case that the function we write are purely repetitive such as those for structural equality or pretty printing. In fact 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* where the target data is the datatype describing the structure of other datatypes which is often called the *description*. While generic programming sounds promising, there has usually been issues regarding usability in traditional functional languages. To represent the description it is often required to use special language extensions, and the programming style is almost orthogonal to ordinary programs.

In dependently-typed languages such as Idris or Agda however, 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. present in 'The Gentle Art of Levitation' how it is possible to build a self-supporting type system which is able to synthesise descriptions to ordinary types and is still powerful enough to describe the description datatype itself. Therefore, generic programming is not different from ordinary programming in such system.

1.1.1 The Importance of Genericity in Dependently-typed Languages

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

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 generic programming based on ordinary programming techniques.

Yet in many ways, the state of the art has been mostly theoretically oriented which can lead to some issues when described types are to be used in a practical manner. First of all, multiple incompatible descriptions are presented in the current field and often even in the same paper, which is not particularly attractive in a practical setting. Secondly, there has been little work 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 version of described types could be implemented successfully it would save programmers both time and effort in writing repetitive functions without any large performance penalties.

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 that supports many common datatypes. I seek to mainly base my effort on reusing some of the existing work on descriptions, and not try to improve underlying type theory to support inductive-inductive and inductive-recursive datastypes; 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 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 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 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

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 present an annotated version of a typical dependently-typed datatype that represents 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 is no difference between a parameter type or an index type due to the fact that Idris can figure this out automatically. [1]

Similarly to the type constructor, data constructors also need names, also called *tags*. Following the tag, the data constructor declaration contains the types of the arguments stored in the constructor and resulting type which must use the type constructor of the data type. 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.

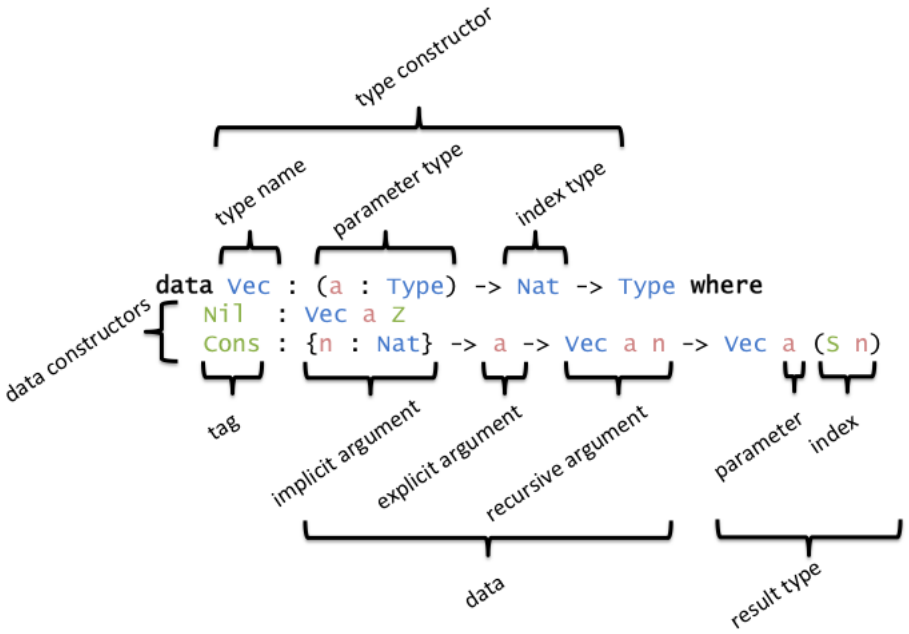

Figure 2.1: Annotated Components of a Datatype Declaration

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 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 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.

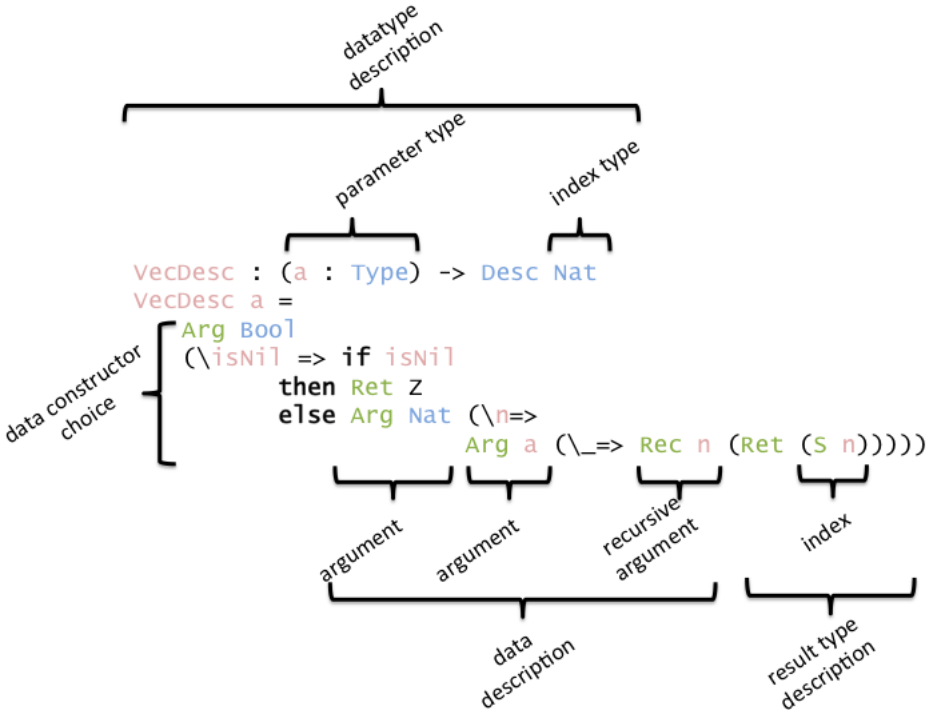

Figure 2.3: Described version of Vec

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 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 type of 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 for 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 had 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.

[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.

2.2 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