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 some functions like structural equality or pretty printing (see Example 1) don't directly dependent on the datatype itself. Therefore, writing such functions for each different datatype becomes a repetitive exercise. 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 each 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 an aspect of Haskell (Peyton Jones et al. 2003) that is challenging to use by ordinary programmers. To represent the description, it is often required to use special language extensions (Magalhães et al. 2010; Jansson and Jeuring 1997) 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 some challenges 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 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 mainly seek to reuse some of the existing work, and not to further develop 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 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. Finally, Chapter 7 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. Recall from Page 2 that a description is a data value representing the structure of a particular datatype. 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 that 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

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

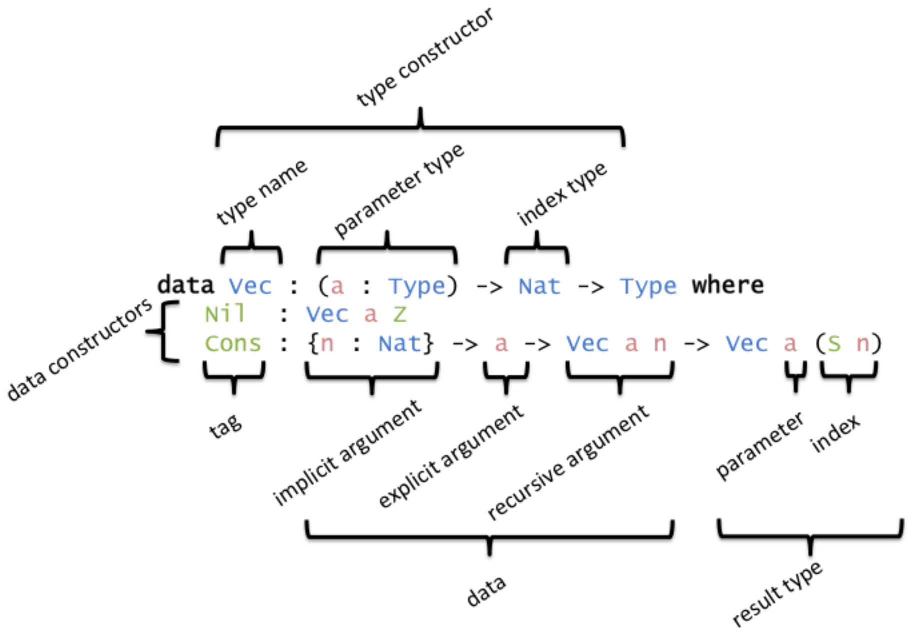

Figure 2.1: Annotated components of a datatype Declaration

three different types of arguments: an ordinary implicit argument, an ordinary explicit argument, and an explicit argument of the type itself (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 `Desc` 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 a recursive argument of the described datatype. The argument of constructor `Rec` is the specification of the rest of the description.

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
   MkUnit : Unit
```

(a) Declaration of `Unit`

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

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

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

(a) Declaration of `Pair`

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

(b) Description of `Pair`

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

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

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

(a) Declaration of `Either`

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

(b) Description of `Either`

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

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 2.6b which is mainly built up using the principles introduced before. The only addition is that the description for `Succ` now uses `Rec` to specify that it requires a recursive argument (to type `Nat` itself).

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

(a) Declaration of `Nat`

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

(b) Description of `Nat`

Figure 2.6: The Natural numbers (`Nat`)

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, `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.

The reader may have noticed that the datatypes presented so far are perfectly expressible in ordinary functional languages like Haskell or Standard ML (Milner, Tofte, and Macqueen 1997). To really exploit the power of dependently-typed programming languages, it should be possible to express datatypes that may be indexed by values. This will be discussed in Section 2.1.3.

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

(a) Declaration of `List`

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

(b) Description of `List`

Figure 2.7: A polymorphic list of elements

2.1.3 Indexing descriptions

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

To give 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 the 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

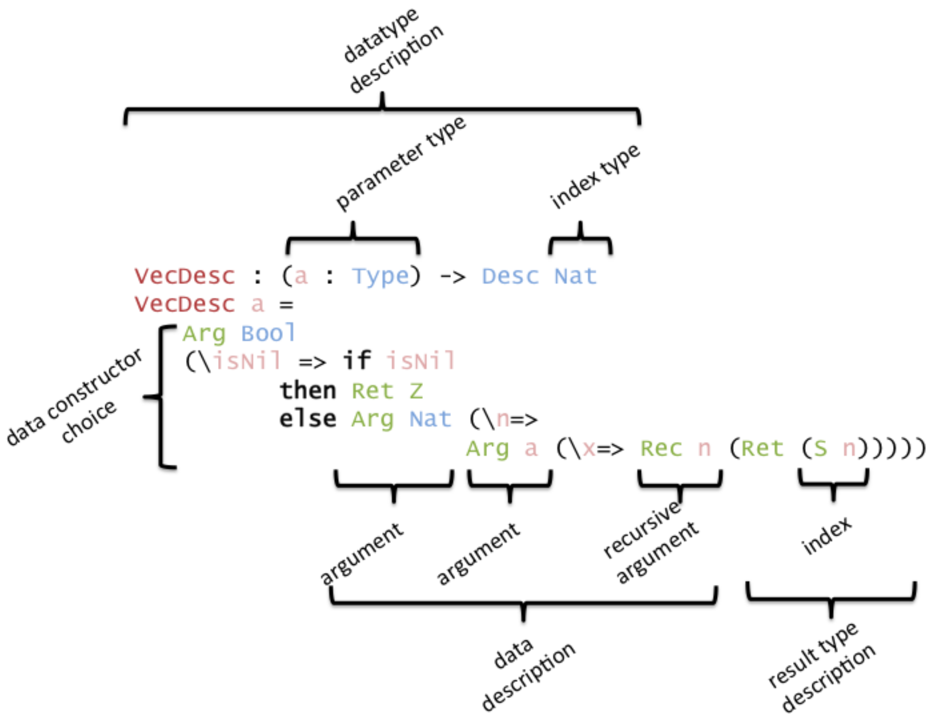

Figure 2.9: Described version of `Vec`

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 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 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? How is it possible to represent more complex datatypes such as mutual recursive ones? 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 does not capture important information such as the names of constructors, requires a series of possibly complicated tests and is not easily extendible if one wants to extend descriptions with new constructors. In Section 2.1.4, I will present a more sophisticated encoding that doesn't suffer from these limitations.

For more demanding datatypes that need 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.

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 preclude interesting generic programs from 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.

Finally, there is the question on how more complicated datatypes are to be represented. Datatypes that require recursive functions as arguments like `Desc`, can not currently be represented in the presented encoding and the description must be extended to be able to describe types as itself (see Section 2.3). Mutual recursive datatypes cannot be represented directly, but it is possible to use indices to represent an isomorphic representation. For example, for two mutually recursive datatypes one could use a boolean argument as index which determines the actual datatype is currently described. Unfortunately, a challenge that still persists is that the most complex inductive families—such as inductive-inductive and inductive-recursive definitions—can not be represented using the presented descriptions.

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 does not 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
CEnum = List CLabel
```

(a) Representation

```
VecCtors : CEnum
VecCtors = [ "Nil" , "Cons" ]
```

(b) Example: Constructors of Vec

Figure 2.10: Constructor labels

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

This encoding has multiple advantages: it ensures that all constructor labels stored in our data exist in the expected list of constructors, it ensures that all datatypes which are dependent on a tag must have at least one constructor, and as a consequence it is possible to specify the empty type by simply requiring a tag on an empty list of expected constructors (since such value would be impossible to create). This encoding 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
NilTag = TZ
```

(a) Tag for `Nil`

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

(b) Tag for `Cons`

Figure 2.12: Example: Tags for constructors of `Vec`

2.1.5 A constructive type of choice

Similarly to how **`if`** was used to map `Bool` values to the descriptions of the various constructors of a datatype in Section 2.1.2, it is desirable to have a way to map `Tag` values 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 types from descriptions

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.

```
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

We are able to actual types using `Synthesise` from the provided descriptions. However, a problem occurs when we want to use `Synthesise` 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 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 an infinite 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 synthesised version of `Vec`. Figure 2.18 shows `Vec` which is a function mimicking its corresponding type constructor, and it even shares the same type signature. The function `Vec` returns a described type using `Desc` passing along the description of `Vec` and its required parameters (*i.e.*, `VecDesc a`), and additionally the expected value of the result index (*i.e.*, `n`).

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

Figure 2.18: Synthesised Version of Vector Description

A simple example representing the vector `[1 , 2 , 3]` is presented in Figure 2.19. Although the value might seem a bit overwhelming at first, it follows a simple pattern: each time a value of `Vec` is needed `Con` is used, 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 occurrences 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.

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

```
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

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 synonyms for the constructors. Figure 2.20 shows `Nil` and `Cons` as synonyms 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 description

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 original 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 aliases from Figure 2.20

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 reader is that many of these constructions such as `Data`, `Tag` and `switch` cannot themselves be described, since they are necessary building blocks for having descriptions. However, what might be surprising is that the description type `Desc` itself, isn't in fact limited by such a constraint 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 primitive recurrences. 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.

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

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

Figure 2.23: Synthesising `HRec` to a real type

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)`.

```
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

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 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.

[2] Of course, such trick only works if the type `Desc` is already known to be 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.

```
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

2.4 Ensuring tagging of descriptions

The plain description type `Desc` accepts descriptions of any form, something which can limit how some algorithms are written. For example, it might be useful to pretty print the constructor tags differently from the constructor arguments, and therefore it would be nice if the type system ensured that it was possible to know where the tags were. Similarly, if one needs to extend the number of constructors in a described type, it is necessary to know how tags are used.

```
TaggedDesc : (e : CEnum) -> (ix : Type) -> Type
TaggedDesc e ix = (l : CLabel) -> Tag l e -> Desc ix
```

Figure 2.26: A datatype for representing descriptions with tags

Dagand (2013) suggests that it is possible to create a type representing tagged descriptions while keeping the same level of expression (see Figure 2.26 for an inspired implementation). The type `TaggedDesc` represents the type of functions from tags to descriptions, which is exactly the result type of function `switchDesc`.

```
Untag : {e : CEnum} -> TaggedDesc e ix -> Desc ix
Untag {e} d = Arg CLabel (\l=> Arg (Tag l e) (\t=> d l t))
```

Figure 2.27: Converting tagged descriptions to ordinary descriptions

A function `Untag` which converts tagged descriptions to ordinary descriptions is shown in Figure 2.27. The `Untag` function converts the function arguments of `TaggedDesc` to described data by using `Arg`, which ensures that the provided constructor tags are stored with their respective constructor arguments.

A key advantage of accepting a `TaggedDesc` and then calling `Untag`—instead of merely accepting an ordinary `Desc`—is that the type system gains knowledge that the first two arguments of data are of type `CLabel` and `Tag`. This permits the algorithm designer, to treat those arguments differently when pretty printing the datatype, during serialisation, and so on and so forth.

```
TData : {e : CEnum} -> TaggedDesc e ix -> (ix -> Type)
TData d = Data (Untag d)
```

Figure 2.28: The described version of tagged descriptions

Figure 2.28 shows a type `TData` which is the analogous of `Data` for tagged descriptions. The definition is simple as it simply converts the provided tagged description to an ordinary description, and then calls `Data` on that. Therefore, the main point of using `TData` is to make it easy to convert a tagged description to a described type.

Chapter 3

Partial evaluation

The description datatype `Desc` presented in Chapter 2 was very flexible and could express many common algebraic datatypes. However as discussed in Section 2.2.2, the corresponding terms used to construct values of described types were much larger and more complex than the corresponding values of ordinary datatypes, yet they do not convey much more relevant information. The reason is that much of the contained data is static information needed solely to provide the right form for generic algorithms, but is not needed when dealing with specific structures. Therefore this chapter discusses relevant partial evaluation techniques needed to minimise the size of the large terms in order to improve runtime performance, by specialising the algorithm with regards to relevant static data when possible.

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. Sometimes it is due to the way programs are structured in a modular fashion (e.g. functions or objects), where readability and reusability are highly valued even if some of the input to these structures 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. Other times, it may be because that the input is known ahead of time and therefore in some way hard-coded (*e.g.*, configuration files or constants). No matter what reason, it can be said that any program `p` accepts a series of static input

$i_{s_0} \ldots i_{s_n} \in I_s$ and a series of dynamic input $i_{d_0} \ldots i_{d_m} \in I_d$, resulting in some output 0.

In many cases it is desirable to only compute programs with known input once and for all, instead of suffering a performance loss every time the program is run. One technique for static computation 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 `mix` is a new program called the *residual program* which is *specialised* with regards to the specified static input. That is for any program p, partially evaluating it regarding its static input I_s—that is `mix` p I_s—results in a residual program p_r. The residual program p_r accepts the remaining dynamic input I_d and produces the same expected result 0; therefore, the following equation is satisfied: $p\ I_s\ I_d \equiv (\mathtt{mix}\ p\ I_s)\ I_d \equiv p_r\ I_d \equiv 0$.

The canonical example of partial evaluation (Jones, Gomard, and Sestoft 1993; Mogensen and Sestoft 1997; Taha 2004) is the `power` function which calculates x^n and is shown in Figure 3.1. In this version of `power`, the control flow is mostly determined by its first variable `n`. Therefore, if `n` is provided statically then it is possible to specialise `power` to avoid the branching dependent on `n` and recursion at run-time.

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

Figure 3.1: The function `power` which calculates the value x^n for input integers x and n

Figure 3.2 shows a partially evaluated version of power, where `n` is fixed statically to `5`. To achieve such optimisation, the partial evaluator must support various interesting optimisation techniques such as *constant folding*, *program point specialisation* and *unfolding/transition compression* (Jones, Gomard, and Sestoft 1993) which will be discussed further in Section 3.2.

```
power_n5 : Int -> Int
power_n5 x =
  x * let half =
          let half' = x
          in half' * half'
      in half * half
```

Figure 3.2: `power` specialised with regards to `n` set to `5`

3.2 An optimising partial evaluator

Since the only requirement for a partial evaluator is that the residual program depends only on some dynamic input, it is simple to make a trivial partial evaluator. The trivial partial evaluator simply "hard-codes" the provided static input, and otherwise leaves the input object program unchanged. However, such partial evaluator is hardly interesting from a performance perspective. In order for a partial evaluator to be interesting, it must be able to utilise a set of optimisation techniques while evaluating a program. In this section, three commonly used optimisation techniques for partial evaluators are presented: constant folding, program point specialisation and unfolding.

3.2.1 Program Point Specialisation

According to Jones, Gomard, and Sestoft (1993), a *program point* is a referable point of execution that forms a part of a larger program. For many modern languages, a program point would be a function or procedure; however, it could also be a label in an assembly language or a clause definition in a logic language. Program point specialisation is the act of creating new versions of existing program points specialised with regards to some statically provided input. That is, a specialised version of a program point `l` is a pair $\langle$`l` , $I_S$$\rangle$ such that I_S is some provided static input somewhere in the program. Figure 3.5 shows a version of `power` where `n` is specialised to `5`, and all recursive calls to power are partially evaluated using program point specialisation.

Polyvariant Specialisation

A program point specialisation is said to be *polyvariant* if there are multiple versions of originally the same program point specialised with varying static input (Hughes 1999; Jones, Gomard, and Sestoft 1993). For example, the `power` function is specialised with regards to different values for the exponent `n` in Figure 3.5 and is therefore polyvariant.

A special case of polyvariant specialisation, is when a program point specialisation is said to exhibit a *polyvariant division*. A division is polyvariant if the set of static arguments varies for specialised versions of that particular program point. Finding a division between static and dynamic arguments of a program point is not a trivial task, and especially not finding a polyvariant one. Section 3.3 discusses what techniques there are for finding such divisions.

3.2.2 Constant folding

Simply put, constant folding is the idea of reducing pure expressions with statically known arguments as much as possible. This includes simple primitive arithmetic and logic operations such as addition, multiplication, conjunction and equality testing; but also pruning statically-determined branching such as **`if`**-expressions (Wegman and Zadeck 1991; Jones, Gomard, and Sestoft 1993) or **`case`**-trees (Boquist 1999).

Reducing arithmetic and logical operations If all the operands of a given arithmetic or logical operator are statically determined then reducing such an operation is simply evaluating the result, *e.g.*, for an expression `2 + 2` it can simply be reduced to the value `4`. However, constant propagation algorithms are often allowed to do other types of simplifying reductions where some operands are dynamic, such as reducing addition with `0`, multiplication with `1` or conjunction with `True`.

Figure 3.3 shows `power` from Figure 3.1 again under specialisation with `n` set to `5`. In the figure, all arithmetic expressions (namely `mod`, `div` and "`-`") and logic expressions (namely "==") have been reduced to simple integer values.

Branch pruning If a conditional is somehow reduced to a constant value by another optimisation, it is possible to completely eliminate branching in **`if`**- or **`case`**-expressions. For **`if`**-expressions, if the conditional is reduced to `True` then the whole expression is reduced to the **`then`** branch, otherwise (if it is reduced to `False`) then the whole expression is reduced to the **`else`** branch. For **`case`**-expressions, they are simply reduced to the branch that matches the pattern of the value provided.

```
power_n5 x =
  if False
  then 1
  else if False
       then let half = power 2 x
            in half * half
       else x * power 4 x
```

Figure 3.3: Reduction of arithmetic and logical expression in `power` with `n` set to `5`

Figure 3.4 shows an updated version of `power_n5` from Figure 3.3. Since all **`if`**-expressions depended on constant values, it was possible to completely eliminate branching from the result.

```
power_n5 x = x * power 4 x
```

Figure 3.4: Pruning of statically determined branches for `power_n5`

Using more advanced optimisation techniques, it is still possible to reduce branches of **`case`**-expressions if only some parts of the conditional are statically determined. If the reader is interested, please refer to Peyton Jones and Lester (1992) and Boquist (1999).

3.2.3 Unfolding

To avoid too many unnecessary indirections in specialised programs, a perhaps important technique is unfolding. Unfolding is usually done at sites where there are function calls and corresponds to inlining the body of the function that is called at the place where it is called. In addition to simply inlining the body of the function, unfolding usually needs to do renaming of local variables to avoid clashes with the external environment. For a languages with labels, transition compression serves as a good analogue to unfolding, where a jump to a label is replaced with the following instructions. While there are many advantages to unfolding, and likewise transition compression, if not careful the partial evaluator can end up in an infinite loop or the resulting code can end up with exponential size in the case of poorly chosen static variables (Jones, Gomard, and Sestoft 1993). Therefore, unfolding cannot be seen as a

```
power_n5 x = x * power_n4 x
power_n4 x =
  let half = power_n2 x
  in half * half
power_n2 x =
  let half = power_n1 x
  in half * half
power_n1 x = x * power_n0 x
power_n0 x = 1
```

Figure 3.5: The function `power_n5` and necessary dependencies after program point optimisation and constant folding

generally safe technique and must be used with care in places where there are branching or similar.

Figure 3.5 shows further transformation of the code in Figure 3.4, after the completion of necessary program point specialisation and constant folding. As can be observed in the figure, there is alot of indirection in each specialisation of `power` requiring a new function call for all cases except when `n` is set to `0`. It is therefore desirable to do unfolding to improve performance, and doing unfolding for `power_n5` brings us back to Figure 3.2 which is the final form of the optimisation.

3.3 Dividing the static and dynamic parts of a program

In Section 3.1, an argument was made that programs usually contain both static and dynamic input. A division for a program point is specifically an assignment of a binding-time, either static or dynamic, to each argument and expression at that point. There are generally two non-exclusive ways to get the binding-time of variables, one is using a binding-time *analysis* and the other is requiring *annotation* of binding-time by the user using *e.g.*, *two-level* syntax (Nielson 1989; Jones, Gomard, and Sestoft 1993). In the broadest sense, binding-time analysis is vaguely similar to type inference while checking whether binding-time is well-annotated is vaguely similar to type checking.

3.3.1 Binding-time analysis

The core idea in a binding-time analysis (BTA) is to infer from given initial program input, what parts of a program may be executed statically. For functional programming languages like Idris, there are roughly four classes of expressions which have different set of binding-time behaviour: constants, variables, function application/operators, and conditionals/case analysis.

- Constants are always assumed to be static, independent of whether they are for primitive types such as integers or strings, or base constructors of datatypes.
- The binding time for a variable usually depends on the type of variable and the surrounding environment. If the variable is let-bound or globally declared it will get the binding-time of the expression that is assigned to it. For function arguments they are assumed to be static in the function body, unless they appear dynamically in a recursive call. The partial evaluator can only optimise function calls when their arguments are known to be static, but such requirement is put where the function is called and not in the body of the function.
- The result of function and operator application is assumed to be static if all input arguments are static, otherwise it is dynamic. One must take special care of functions which perform environmental side-effects, such as those using the `IO`-monad in Idris, which result must always be classified as dynamic (since the result can't be determined at binding time).
- Conditionals expressions like **`if`** and **`case`** are considered to be static if the condition they depend upon is static (since that determines the control flow), otherwise dynamic.

When the BTA is complete, the partial evaluator can use such information to create a new residual program. For each function call with static arguments, it can choose to reduce all dependent static control flow and expressions; thereby creating a new program that depends solely on dynamic input.

3.3.2 Two-level syntax

Another approach to state binding-time of an expression is to allow the user to annotate programs using two-level syntax. This approach is used in Jones, Gomard, and Sestoft (1993), but is also available for popular programming languages such as OCaml (Taha 2004) and Java (Westbrook et al. 2010). The core idea behind two-level syntax is to provide static version of available expressions such as control structures, application and lambda abstractions in addition to the dynamic versions. Two-level expressions that are static are often presented as underlined versions of their dynamic counterparts (*e.g.*, **`if`**), and usually an operator is used to distinguish between static and dynamic application such as **`$`** or **`@`**. In order to allow embedding of static expressions inside dynamic values a built-in annotation **`lift`** is usually provided.

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

Figure 3.6: Two-level syntax annotated version of `power` function, where underlined operations are static

Figure 3.6 shows an annotated version of `power` where `n` is assumed to be provided statically. The annotation shows what will be reduced after a value for `n` is provided, and as can be seen in the specialised version (see Figure 3.2) there are no traces left of expression marked as static in the residual program. If `n` is to be provided at runtime, it is usually possible to forget the binding-time annotations and execute all of `power` dynamically.

3.4 Constructor specialisation

The previous sections focused mostly on partial evaluation as a way of optimising code, however Mogensen (1993) and Dussart, Bevers, and

De Vlaminck (1995) suggest that partial evaluation is also a useful technique to optimise data. The core idea is to specialise constructors in the same vein as specialising functions, or rather specifically, create new constructors as alternatives of algebraic datatypes where the statically provided input is fixed (or perhaps completely erased).

3.4.1 Example: Serialisation to S-expressions

Assume there is a program that serialises an association list to an S-expression and is specialised with regards to a specific schema where it is either the name and age of a person or the name of a department. Figure 3.7 shows how a reasonable result could look like after using ordinary partial evaluation techniques.

```
serialize : List (String, String) -> String
serialize [("name", name), ("age", age)] =
    "(:name " ++ show name
 ++ " :age " ++ show age ++ ")"
serialize [("department", department)] =
    "(:department " ++ show department ++ ")"
```

Figure 3.7: A program for serialisation `serialize`, specialised using classic techniques with regards to a specific schema

There are however still some drawbacks with such program: it requires pattern matching on nested constructors, and does multiple string comparisons, both of which are potentially very time consuming. Yet, much of the data is statically specified, so it seems that there is still room for improvement. Luckily, constructor specialisation permits specialisation of data by creating suitable constructors in a suitable datatype. Figure 3.8 shows how new constructors are created in a datatype `Schema` such that all static data is eliminated, and two suitable constructors are created: `Person` and `Department` [1]. After constructor specialisation, now the only comparison necessary is comparing tags of datatypes, something which should be much more efficient than the old solution.

[1]For ease of reading, the naming is prettified compared to what automated constructor specialisation would generate

```
data Schema =
        Person String String
      | Department String

serialize : Schema -> String
serialize (Person name age) =
    "(:name " ++ show name
 ++ " :age " ++ show age ++ ")"
serialize (Department department) =
    "(:department " ++ show department ++ ")"
```

Figure 3.8: A constructor specialised version of `serialize`

3.4.2 Algorithm for specialising constructors

Dussart, Bevers, and De Vlaminck (1995) presents a step-by-step algorithm for performing constructor specialisation that has multiple advantages compared to the original one presented in Mogensen (1993). One advantage is that it specialises constructors in a polyvariant fashion, which means that specialisation of one datatype can create multiple new datatypes. Another advantage is that it only requires one pass for calculating the fix-point calculations necessary in order to create new suitable types. The algorithm consists of three phases: finding a minimal pattern that describes the occurrences of constructors with static values, generating the necessary code operating on such values and finally creating suitable datatypes to hold the specialised constructors.

A grammar of datatypes The first phase of the algorithm is to find a possibly recursive pattern or *grammar* that describes what particular sets of constructors occur in the same expression at given program points. For the example presented in Figure 3.7, the final grammar would look like something shown in Example 2 (in BNF style notation).

Example 2

$$
\begin{array}{lcl}
\langle \textit{string} \rangle & ::= & \ldots \\
\\
\langle \textit{schema} \rangle & ::= & [(\text{"name"}, \langle \textit{string} \rangle), (\text{"age"}, \langle \textit{string} \rangle)] \\
 & | & [(\text{"department"}, \langle \textit{string} \rangle)]
\end{array}
$$

The grammar is extracted by analysing the code structure for where data is constructed and suitably combining such data, e.g. having alternatives in the grammar where there are **case**-expressions. To avoid non-termination while partially evaluating, the analysis tries to generalise (that is specify as dynamic) places where there occur recursive references to the same datatype. The effect of that is that non-terminals in the grammar might be defined recursively, which closely mimics the structure of inductive datatypes. Following extraction of the grammar, fix-point computation techniques inspired by Jones and Mycroft (1986) are used to find a minimal function graph that depends on such grammar.

Code generation The second phase uses the minimal function graph found in the first one to construct specialised versions of functions. This happens by traversing the graph and specialising each function as normal, recursively rebuilding expressions from specialised versions. **case**-expressions are handled specially during code generation in order to accommodate the specialised datatypes, and therefore they are restructured, adding new branches, such that they fit the extracted grammar.

Defining suitable types The final phase is to group the newly generated specialised constructors into suitable type definitions. This is done in order for the residual program to be valid (type correct) and the compiler can perform other datatype optimisations. The process prestented is simple: it starts with all constructors being in their own datatype, and merges datatype as necessary when a program dependens on values from either datatype.

Chapter 4

Levitating Idris

4.1 Creating descriptions from ordinary datatype declarations

The description datatype `Desc` mentioned in Chapter 2 provides the necessary plumbing to perform generic programming. However, creating described types required a lot of heavy encoding; requiring labels, descriptions and aliases to be manually written out using the provided low-level constructs. It would be better if the compiler could generate the necessary descriptions and aliases from ordinary datatype declarations since that would provide the powerful generic constructs without sacrificing the immediate readability of how the datatype is structured. This section highlights my effort on extending the Idris language with constructs to make it simpler to work with described types.

4.1.1 High-level overview

To make it easier to use described types in Idris, I have added three new language constructs: one annotation on datatypes **`%described`** and two built-in operations **`labels_for`** and **`desc_for`**. The **`%described`** annotation has to appear prior to a datatype declaration (see Figure 4.1), and specifies that the compiler should generate the relevant constructs for working with described types. The constructs that are generated are similar in style to the ones presented in Sections 2.1.2–2.2.2, and represent the constructor labels of type `CEnum` for the datatype, the description of type `TaggedDesc` for the datatype, an alias for the described version of the datatype and relevant aliases for its constructors.

```
%described data Nat : Type where
  ...
```

Figure 4.1: An annotation `%described` for generating descriptions from declarations

The `labels_for` operation provides the generated constructor labels for a particular datatype. For example, to access the generated version of `VecCtors` from Figure 2.10b, one must use `labels_for Vec`. Similarly `desc_for` is used to access the generated description for a datatype, *e.g.*, `desc_for Vec` returns the equivalent value of `VecDesc` from Figure 2.15. The type and constructor aliases are chosen to match the signature and name of the datatype to be described, similarly to how it was done in Section 2.2.2.

4.1.2 Algorithms for generation

When a user asks the compiler to generate relevant functions for working with described types (using `%described`), then the compiler generates four types of functions: one representing the constructor labels of the datatype, one representing the description of the datatype, one representing a type alias based on the description, and, for each constructor of the datatype, a function that constructs an isomorphic value of the described type. The following paragraphs describe informally how each of these types of functions are generated from the perspective of the compiler. The main part of the Haskell function that generates these functions can be seen in Appendix A.[1]

Generation of labels

The first part of what is generated is a value containing a list of all the constructor names of the target datatype. Since all the constructor names are represented internally as strings by the compiler, this step simply involves creating a new clause of type `CEnum` and then assigning to it the Idris representation of a list where all the constructor names are converted to string literals.

[1]Due to time constraints, the current version of the function only supports datatypes without parameters and indices.

Generation of descriptions

The second part of what is generated is a value containing the description of the target described datatype. Algorithm 1 works by traversing the Idris AST, finally producing a new declaration which contains the description.

Algorithm 1
Generating descriptions for datatypes

1. Given a division of parameters `x`$_j$ and indices `y`$_k$ create a fitting type signature for the description
   a) For parameters quantify those over the whole description value
   b) For indices, they are first "uncurried" in a dependent pair, and then the whole dependent pair is passed to the tagged description type `TaggedDesc` as index

      That is, for a datatype `D:(x`$_1$`:a`$_1$`)...(x`$_n$`:a`$_n$`) -> (y`$_1$`:i`$_1$`)...(y`$_n$`:i`$_n$`) -> Type` the signature of the corresponding description becomes `(x`$_1$`:a`$_1$`) -> ... -> (x`$_n$`:a`$_n$`)`

      `-> TaggedDesc (`**`labels_for`** `D) (y`$_1$`:i`$_1$`**...**y`$_n$`:i`$_n$`)` where `a`$_j$ is the type of a parameter and `i`$_k$ is the type of an index
2. Create a fitting clause for the description where all parameters `x`$_1$`...x`$_n$ are available as arguments
3. To assign a value to the clause, calculate the value conforming to the π type for the description and apply `switchDesc` to the result
   a) To calculate the value conforming to π type for the datatype, iterate through the constructors such that for each constructor `c`$_j$:
      i. Create a new pair where the first component is the description for `c`$_j$ and the second component is the descriptions for the rest of the constructors ending with `()`
      ii. To calculate the description for `c`$_j$, iterate through all its arguments:

A. If the argument is recursive, use the relevant constructor for recursion, either `Rec` or `HRec`, and apply it to the index values "uncurried" in a dependent pair

   For example, for an argument `D` x_1 `...` x_n y_1 `...` y_n `-> ...` use `Rec (`y_1`**...**`y_n`) ...`

B. Otherwise if the argument is not recursive, use `Arg`

   For example, for an argument `(x:A) -> ...` use `Arg A (\x=>...)`

C. Finally, when there are no arguments left use `Ret` applying it to the expected indices similar how to it was done for recursive arguments

After the description is generated it is elaborated by the Idris compiler to ensure that the generated code is correct. If the description is elaborated successfully, the user can then access it using the **`desc_for`** operation as described in Section 4.1.1.

Generation of aliases

The generation of a type alias for a datatype `D` is simple. First, create a new declaration with the same name `D` and type signature of the datatype. Then, the result of `D` is `TData` applied to the description for that datatype **`desc_for`** `D`, with the parameters and indices adjusted to fit the expected form. That is, the values of the parameters must be provided to the description to get a value of type `TaggedDesc (`**`labels_for`**` D) (`y_1`:`i_1`**...**`y_n`:`i_n`)`. Then `TData` is applied to that value in addition to the values of the indices which are packed together in a dependent pair yielding a value of type `Type` as required.

Generating constructor aliases requires a bit more effort than with type aliases, but still follows a step-by-step process. The first thing to do is to enumerate all constructors, incrementally assigning each one a number (starting from `0`) which is used to calculate the corresponding `Tag` value. Similarly to how type aliases were generated, a new declaration is then created for each constructor `C` with the corresponding name `C` and type signature. The only difference is that recursive arguments in the type signature are changed to use the described version of the datatype using the generated type alias. The result of `D` is the application of `Con` to an expression formed of right-nested dependent pairs that represent the synthesised version of the description for the datatype of `C`. That is, the nested dependent pairs contain the constructor label, followed by the constructor tag which is calculated from the assigned number, then followed by the arguments of the constructor and finally ending with the value `Refl`.

4.2 Parametric extension to descriptions

While it was possible to describe parametrised datatypes using the description presented in Chapter 2, it was not possible to distinguish between values of parameters and other kinds of values after instantiating the parameter. Therefore, some of the algorithms which depend upon the knowledge of where a particular parameter is, such as the functorial map, cannot be implemented in a generic fashion. Since it would be desirable to implement such algorithms, a suitable description which has a built-in encoding of parameters must be created. Section 4.2.1 investigates some of the existing notions of what parameters are, and Section 4.2.2 shows a new description which supports the parametric notion of parameters.

4.2.1 The parametricity of parameters

What is a parameter? In an environment with dependent types, this simple question can lead to more answers than bargained for. While almost all notions agree that the type `Vec a n` is parametrised by `a` and indexed by `n`, there has been seemingly less agreement on what a parameter is in general. The issue is probably due to the different things that different kinds of parameters are used for, which overloads the word parameter. I will in the following sections present some definitions of what a parameter actually is, explain the rationale behind each one and show where they don't agree.

Parameters as eliminator quantifiers

One notion of a parameter is the one presented by Dybjer (1997), and implemented in Idris (Brady 2013). The notion is defined in terms of how a parameter appears in the elimination rule for a particular datatype, where it is expected that parameters appear first in the type signature such that they are quantified over the complete elimination rule.

```
VecElim :
  (a : Type) ->
  (prop : (n : Nat) -> Vec a n -> Type) ->
  (propNil  : prop Z Nil) ->
  (propCons : (n' : Nat) -> (x : a) -> (xs : Vec a n') ->
              (ih : prop n' xs) prop (S n') (Cons x xs)) ->
  (m : Nat) ->
  (scrutinee : Vec a m) ->
  prop m scrutinee
```

Figure 4.2: Elimination rule for `Vec`

Since parameters are quantified over the elimination rule, it is ensured that they are constant relative to the property to be eliminated, *i.e.*, they do not change depending on the value that may be provided as scrutinee. Figure 4.2 shows the elimination rule for `Vec`, where it can be observed that `a` is quantified over the whole expression and thus considered to be a parameter, while `n` changes in the property to be eliminated `prop` and is therefore considered an index.

A restriction that is necessary to ensure that parameters are quantified correctly in the elimination rule, is that the parameter should appear uniformly in the result type and all of the recursive arguments of a constructor. This rules out data structures such as the one presented in Figure 4.3 to be considered having any parameters in this system, since the type argument `(a,a)` to the recursive argument of `Cons` is different than the `a` in the rest of the structure.

Parameters as terms with parametricity

Another notion of a parameter which is presented by Bernardy, Jansson, and Paterson (2010), is that it is a type argument which exhibits parametricity (Reynolds 1983; Wadler 1989). A type argument exhibits parametricity if the same relations in the datatype are satisfied independently of which value is provided; that is, one should not be able to inspect or constrain the value of a parameter (*e.g.*, using propositional equality). Figure 4.3 shows a nested datatype (Bird and Meertens 1998) `NList`, which is parametric even though `a` does not appear uniformly in the datatype declaration. The reason that `a` is still considered parametric, is that the recursive argument uses *functorial composition*. With functorial composition, recursive arguments can use parameters in a non-uniform fashion as long as it does not change parametricity and preserves the expected relations. In the `NList` example, the `NList (a,a)` argument can be seen as a composition of `NList` with the homogenous pair type `\p=>(p,p)` which itself satisfies the necessary parametricity requirements.

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

Figure 4.3: The nested datatype `NList`

Parametricity of type arguments is necessary to correctly implement certain useful declarative functions, such as those that are methods of the `Functor`, `Traversable` and `Foldable` type classes from Haskell. If constraints were made based on the type argument of the datatype, it wouldn't be possible to satisfy the associated laws of these type classes and possibly not even provide an implementation that can type check.

Parameters as uniform indices

The last notion of a parameter is the one used in Agda (Norell 2009) which only requires that parameters are uniform in the resulting type of constructors. This allows the creation of more expressive datatypes, and can encompass both parametric parameters and parameters which are quantified uniformly over eliminators; however, it does not ensure either of these properties. While this provides more freedom to decide how to structure datatypes, it makes it harder to see what correspondence there is between a parameter and the semantic properties it imposes.

```
data Vec : (a : Type) -> (n : Nat) -> Type where
   Nil  : {p : n = 0} -> Vec a n
   Cons : {m : Nat} -> {p : n = 1 + m} ->
            a -> Vec a m -> Vec a n
```

Figure 4.4: Alternative definition of `Vec` using equality constraints

For example, Figure 4.4 shows a datatype where both `a` and `n` are uniform in the result and which could accepted as a Norell-style parameter. However, the arguments in this definition are neither parametric nor uniform and therefore do not fit into the other notions of parameters.

4.2.2 Parametrically extending the description

One suggestion on how to ensure the parametricity of parameters was made by Bernardy ("A theory of parametric polymorphism and an application"). It was suggested that whenever a parameter was bound and parametricity was needed, one could provide an additional argument; a *witness* which proved that the parameter was parametric. However such suggestion could be hard to work with in practical generic programming and instead this section will focus on providing an encoding that is conservative in a way that disallows non-parametric use of parameters but is still able to express many interesting datatypes.

Explicit parameters

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

Figure 4.5: A description for parametrised types

Figure 4.5 presents a description [2] akin to the one presented in Figure 2.2 except a new constructor `Par`—inspired by an encoding in Benke, Dybjer, and Jansson (2003)—is added. The constructor `Par` represents an argument of the provided parameter in the datatype to be described.

```
SynthesisePar : ParDesc -> (Type -> Type) -> (Type -> Type)
SynthesisePar Ret       x a = ()
SynthesisePar (Rec d)   x a =
    (rec : x a ** SynthesisePar d x a)
SynthesisePar (Arg b d) x a =
    (arg : b ** SynthesisePar (d arg) x a)
SynthesisePar (Par d) x a =
    (arg : a ** SynthesisePar d x a)
```

Figure 4.6: Synthesising an ordinary type from `ParDesc`

To convert the newly presented description to an ordinary datatype a suitable `SynthesisePar` function is declared. The function looks mostly

[2] `HRec` is omitted for presentation purposes, but can be added in the same style as presented in Chapter 2.

the same as the one presented in Chapter 2, however the described type is now of type `Type -> Type` and there is an additional clause for `Par`. The clause for `Par` produces a dependent pair similar to most other clauses, where the first component has to be an argument of the provided parameter as expected and the second component is the type synthesised from the rest of the description. Since it is not possible to depend on the type nor values of parameters in the presented encoding, parametricity is ensured by construction. In return, some expression power is lost since it is not possible to have complex arguments that use these parameters.

Supporting functorial composition

In Section 4.2.1 it was stated that parameters in nested datatypes were parametric because they were composed in a functorial fashion. Therefore, it would be desirable to add such ability to the presented description datatype and thereby increase the expressiveness.

Figure 4.7 shows the addition of the constructor `CompRec` which represents a nested recursive argument of a datatype. The first argument of the constructor `f` is the type to be functorially composed onto the parameter, and must therefore be a function that accepts the parameter and returns a type. To ensure that `f` acts functorially, an additional argument `ffunctor` must be given. This variable `ffunctor` provides the `Functor` type class instance for `f`, and uses the built-in type resolution mechanism of Idris to automatically find that instance. Finally the last argument is the rest of the description for the datatype to be described.

```
CompRec : (f : Type -> Type) ->
          {default %instance ffunctor : Functor f} ->
          ParDesc -> ParDesc
```

Figure 4.7: Adding support for describing functorial composition

As an example of a description for a nested datatype, please take a look at Figure 4.8. The figure shows the description for `NList` from Figure 4.3, and the most interesting part is the application of `CompRec`. In this context it is applied to a functor `PairP`, which represents the same type as `\p=>(p,p)` from Section 4.2.1. To type check this application of `CompRec` there must be a `Functor` instance for `PairP`, however it can be left out since Idris can resolve it implicitly.

```
NestedT : CEnum
NestedT = ["Nil", "Cons"]

NestedPD : ParDesc
NestedPD = Arg CLabel (\l =>
             Arg (Tag l NestedT)
               (switch NestedT (\l',t' => ParDesc) (
                  Ret
                , (Par (CompRec PairP Ret)
                , ()
                )) l))
```

Figure 4.8: Described version of `NList`

Converting `CompRec` to an ordinary type is mostly similar to converting ordinary recursive arguments `Rec` (see Figure 4.9). The only difference is that `f` must be applied to the parameter first before applying the described version of the type. Here, it is perhaps clearest why such operation is called functorial composition, since `x (f a)` is equivalent to `(x . f) a` which uses the ordinary function composition operator "`.`".

```
SynthesisePar (CompRec f d) x a =
   (rec: x (f a) ** SynthesisePar d x a)
```

Figure 4.9: Transforming `CompRec` to a type

Conversion to ordinary descriptions

Descriptions for parametrised datatypes can be converted to ordinary description using a simple step-by-step process (see Figure 4.10). The result type must be indexed by `Type`, because recursion expressed by `CompRec` is not uniform and therefore can't be quantified uniformly over the whole description. Since the converted description would hold the same data as the provided input description, it allows generic descriptions written for ordinary descriptions to be reused. In fact, converting the description and then performing synthesis should yield isomorphic data, which means that `Synthesise` only needs to be implemented for ordinary descriptions.

While `ParDesc` and `Desc` were presented as two separate datatypes for readability purposes, they can be in fact combined. The only thing that adds complexity is that the presence or absence of parameters must

```
convertDesc : ParDesc -> (Type -> Desc Type)
convertDesc Ret a              =
   Ret a
convertDesc (Par d) a          =
   Arg a (\arg => convertDesc d a)
convertDesc (Rec d) a          =
   Rec a (convertDesc d a)
convertDesc (CompRec f d) a =
   Rec (f a) (convertDesc d a)
convertDesc (Arg b d) a        =
   Arg b (\arg => convertDesc (d arg) a)
```

Figure 4.10: Converting `ParDesc` to ordinary indexed `Desc`

be accounted for at all stages, and further generalisation towards arity-generic programming requires some advanced fiddling with the type system.

4.3 Proposed Improvements

4.3.1 Any practical type theory permits forgetting unused dependencies

4.3.2 An annotation system for functional generic programming

Chapter 5

Practical examples

5.1 Generic algorithms for deriving type class instances

5.1.1 Specifying the necessary constraints

For many type classes, implementing them for a complex datatype sometimes requires an implementation of the type class for its parts. For example, to pretty print a list of items it is required that there is a way to print each individual element.

```
Constraints1 : (class : Type -> Type) -> Desc ix -> Type
Constraints1 class (Ret j)    = ()
Constraints1 class (Arg a b) =
  (class a, (arg :  a) -> Constraints1 class (b arg))
Constraints1 class (Rec j d) = ((), Constraints1 class d)
Constraints1 class (HRec j a d) = _|_
```

Figure 5.1: A function that calculates type class constraints for a single parameter type class

Figure 5.1 shows a function `Constraints1` that can be used to calculate type class constraints needed to have an instance of a type class `class` (which takes a simple parameter of type `Type`) for a datatype `D`, given the description for `D`. The function works by iterating over the description type and specifies that each external data member with description `Arg` must have an instance of the provided type class. The only other interesting case is `HRec`, where the false type `_|_` is required. The false type makes it impossible to fulfil the constraints for a datatype using `HRec`, and is used intentionally to avoid implementing algorithms using descriptions of that form. This is because it is generally hard to implement arbitrary algorithms such as decidable equality on functions, of which `HRec` is a representative.

5.1.2 Pretty printing

One of the most commonly used and automatically derived operations on datatypes is pretty printing. Therefore, this function is suitable as an introductory example of a generic algorithm using our implementation of descriptions.

```
gshow : (d : TaggedDesc e ix) ->
        (constraints : Constraints1 Show (Untag d)) ->
        {i : ix} -> (x : TData d i) -> String
gshow d (l, constraints) (Con (label ** (tag ** rest))) =
      let (t, constraints') = (constraints label)
      in label ++
          (assert_total $ gshowd d (l, constraints)
                    (d label tag) (constraints' tag) rest)
```

Figure 5.2: Generically pretty printing a described type

An algorithm for generic pretty printing `gshow` is given in Figure 5.2, and clearly follows the informal algorithm presented in Example 1. The type signature of `gshow` might look a bit daunting at first, but the only major difference from ordinary pretty printing is that it requires not just the data `x` but also the associated description `d` and some constraints on its subcomponents `constraints` to be able to pretty print them. The function first prints the name of the input constructor, and then calls a function `gshowd` which pretty prints the individual constructor arguments according to the provided description.

```
gshowd : (dr : TaggedDesc e ix) ->
         (constraintsr : Constraints1 Show (Untag dr)) ->
         (d : Desc ix) ->
         (constraints : Constraints1 Show d) -> {i : ix} ->
         (synth : Synthesise d (TData dr) i) -> String
gshowd dr constraintsr (Ret i) () Refl = ""
gshowd dr constraintsr (Rec j d) ((), constraints) (rec ** rest)  =
    " " ++ parenthesise (gshow dr constraintsr rec)
        ++ gshowd dr constraintsr d constraints rest
gshowd dr constraintsr (Arg a b) (showa, showb) (arg ** rest)  =
    " " ++ parenthesise (show @{showa} arg)
        ++ gshowd dr constraintsr (b arg) (showb arg) rest
gshowd dr constraintsr (HRec j a d) constraints (rec ** rest) = absurd constraints
```

Figure 5.3: Iterating through the description and pretty printing individual components

Function gshowd in Figure 5.3 iterates through the arguments of the input constructor. If the end of the constructor (having description Ret) is reached, the empty string is printed. If it is a recursive argument, the generic show gshow is called again with the description for the whole datatype, parenthesised if necessary; then the rest of the constructor arguments are printed. Otherwise if it is an ordinary argument, the associated show method is called using the instance provided from the constraints, again parenthesised if necessary, and then the rest of the constructor arguments are printed. Finally, if a higher-order recursive argument is met it is dismissed using absurd since it should not be possible to reach this case because the constraints require a value of type _|_.

Example: Implementing Show for the described type Pair

As an example of how to use the generic pretty print function gshow to implement the show method of Show type class for the described type Pair, see Figure 5.4. The call to gshow is passed the description PairDesc and the necesssary constraints pairShowConstrs defined in Figure 5.5. To create the constraints structure, one simply has to create a tuple to match the type required by Constraints1 and then call %instance the right places. The %instance expression is evaluated at compile time, and allows Idris to perform type class resolution returning the associated instance for the target type class.

```
instance Show Pair where
  show = gshow PairDesc pairShowConstrs
```

Figure 5.4: Creating a Show instance for Pair using the generic gshow

The Show constraints for Pair are easy to create, and are almost on the verge of being boilerplate. In fact calculating the constraints for other single-parameter type classes have almost the exact same structure. However, Idris cannot do type class resolution for any arbitrary type class c, and therefore it is impossible to define a function that calculates these constraints (since it would not be able to type check).

There could be two solutions to this problem: one would be to create a macro system for Idris which only type checks the expression after it is instantiated with the right type class, and another would be to create a suitable tactic that calculates such constraints at compile time when

```
pairShowConstrs : Constraints1 Show (Untag PairDesc)
pairShowConstrs =
   (%instance, \l =>
     (%instance, \t =>
       switch ["MkPair"]
         (\l',t' => Constraints1 Show (PairDesc l' t'))
         ( (%instance,\_ => (%instance, \_ => ()))
         , ()) l t))
```

Figure 5.5: Constraints necessary to implement `Show` for `Pair`

needed. Since these solutions would require considerable effort beyond the scope of this project, implementing any of them is considered to be future work.

5.1.3 Decidable equality

Something which is a bit more interesting to do generically is decidable equality, since that usually requires handling a quadratic number of cases relative to the number of constructors, in addition to writing a significant number of associated lemmas. However, before implementing decidable equality for described types, I will start by implementing it for `Tag` (recall from Figure 2.11) to make it easier for the user to satisfy the necessary constraints.

```
lemma_tz_not_ts : {t : Tag l e} ->
                  TZ {l} {e} = TS {l'} {e} {l} t -> _|_
lemma_tz_not_ts Refl impossible
```

Figure 5.6: Lemma specifying that `TZ` is not equal to `TS`

To implement decidable equality for `Tag`, a couple of lemmas are necessary. The first necessary lemma `lemma_tz_not_ts` is shown in Figure 5.6 which shows that the different constructors of `Tag`, namely `TZ` and `TS` are different.

```
lemma_ts_injective : {t : Tag l e} -> {t' : Tag lr e} ->
     TS {l'} {e} {l} t = TS {l'=lr'} {e} {l=lr} t' -> t = t'
lemma_tz_injective Refl = Refl
```

Figure 5.7: Lemma proving the injectivity of `TS`

The second necessary lemma to prove is the injectivity of the constructor TS (see Figure 5.7). That is, if given two values of constructor TS which are equal, then it is possible to show that their inner arguments of type Tag are equal.

```
instance DecEq (Tag l e) where
  decEq TZ      TZ      = Yes Refl
  decEq TZ      (TS t)  = No lemma_tz_not_ts
  decEq (TS t) TZ       = No (negEqSym lemma_tz_not_ts)
  decEq (TS t) (TS t') with (decEq t t')
    decEq (TS t) (TS t)  | Yes Refl = Yes Refl
    decEq (TS t) (TS t') | No nope  =
                       No (\x => nope (lemma_ts_injective x))
```

Figure 5.8: DecEq instance for Tag

Given the necessary lemmas, it should now be possible to implement the DecEq instance for Tag. The cases are straightforward: if both constructors are TZ then they are always equal, if they differ then they are not equal and the lemma_tz_not_ts (or its symmetric version) is used as evidence, and finally if both constructors are TS then they are equal if their inner tags are equal, otherwise they are not and the injectivity lemma is used to compose the proof.

Now that there is a suitable implementation of decidable equality for the tags, it is possible to implement the generic version of decidable equality. Similarly to the implementation for tags, a few injectivity lemmas are also needed for the generic version of decidable equality: one for the injectivity of Con, and two for the injectivity of dependent pairs (which is the type that Synthesise converts most descriptions to).

```
lemma_con_injective : {ix : Type} -> {d : Desc ix} ->
                      {i : ix} ->
                      {x,y : Synthesise d (Data d) i} ->
                      Con x = Con y -> x = y
lemma_con_injective Refl = Refl
```

Figure 5.9: Lemma proving that Con is injective

The injectivity lemma for Con is shown in Figure 5.9. It shows that given that two described types of type Data are equal, then the contained data must be equal too.

The injectivity lemmas for dependent pairs are shown in Figure 5.10 and Figure 5.11. The first lemma states that given that two dependent

```
lemma_fst_injective : {a : Type} -> {b : a -> Type} ->
                      {x,y : a} -> {xs : b x, ys : b y} ->
                      (x ** xs) = (y ** ys) -> x = y
lemma_fst_injective Refl = Refl
```

Figure 5.10: Injectivity lemma for the first component of dependent pairs

```
lemma_snd_injective : {a : Type} -> {b : a -> Type} ->
                      {x,y : a} -> {xs : b x, ys : b y} ->
                      (x ** xs) = (y ** ys) -> xs = ys
lemma_snd_injective Refl = Refl
```

Figure 5.11: Injectivity lemma for the second component of dependent pairs

pairs are equal, then their first components are equal. The second lemma states the same, but for the second components instead.

The implementation of generic decidable equality is shown in Figure 5.12. The type signature is a similar style to the one for generic pretty printing, and requires both the description and necessary instances of decidable equality for the subcomponents of the datatype. The implementation is defined such that two described types constructed with `Con` are equal, if their individual data components (including constructor tag) are equal. Otherwise if the individual data components are not equal, then the injectivity lemma for `Con` is composed with the counter-proof to be used as a new counter-proof for inequality.

```
gdecEq : (d : TaggedDesc e ix) ->
         (constraints : Constraints1 DecEq (Untag d)) ->
         {i : ix} -> (x : TData d i) -> (y : TData d i) ->
         Dec (x = y)
gdecEq d cstsrs (Con x) (Con y)
                with (assert_total $
                        gdecEqd d cstsrs (Untag d) cstsrs x y)
  gdecEq d cstsrs (Con x) (Con x) | Yes Refl = Yes Refl
  gdecEq d cstsrs (Con x) (Con y) | No  nope =
                      No (\x => nope $ lemma_con_injective x)
```

Figure 5.12: Generic implementation of decidable equality `DecEq`

To check whether the individual components are equal or not, the function `gdecEqd` is used, which takes as argument the original description and constraints (to use for recursive calls) in addition to the description and constraints which need to be iterated. The type signature for `gdecEqd` is shown in Figure 5.13.

```
gdecEqd : (dr : TaggedDesc e ix) ->
          (constraintsr : Constraints1 DecEq (Untag dr)) ->
          (d : Desc ix) -> (constraints : Constraints1 DecEq d) ->
          {i : ix} ->
          (x : Synthesise d (TData dr) i) -> (y : Synthesise d (TData dr) i) ->
          Dec (x = y)
```

Figure 5.13: The type signature of `gdecEqd`

If the end of the description is reached (*i.e.*, having the description `Ret`), then two values are always considered to be equal. The clause for `gdecEqd` in the case of `Ret` is shown in Figure 5.14.

```
gdecEqd dr constraintsr (Ret j)  () Refl Refl = Yes Refl
```

Figure 5.14: Checking that two described types with description `Ret` are equal

Probably the most interesting case is shown in Figure 5.15 and is when it is an argument that is described (*i.e.*, `Arg`). There are two things that needs to be checked, the equality of argument values themselves and the equality of the rest of the described type. To check whether the values of the provided arguments are equal, `decEq` is called with the provided type class instance from the constraints `deceqa`. If the values are equal then it is possible to proceed checking the rest of the described type, otherwise the `lemma_fst_injective` lemma is used to compose a counter-proof. If the rest of the described type is equal then an equality proof can be provided, otherwise the `lemma_snd_injective` lemma is used to compose a counter-proof.

```
gdecEqd dr constraintsr (Arg a b) (deceqa, deceqb) (arg ** rest) (arg' ** rest')
         with (decEq @{deceqa} arg arg')
  gdecEqd dr constraintsr (Arg a b) (deceqa, deceqb) (arg ** rest) (arg ** rest')
         | Yes Refl with (gdecEqd dr constraintsr (b arg) (deceqb arg) rest rest')
    gdecEqd dr constraintsr (Arg a b) (deceqa, deceqb) (arg ** rest) (arg ** rest)
         | Yes Refl | Yes Refl = Yes Refl
    gdecEqd dr constraintsr (Arg a b) (deceqa, deceqb) (arg ** rest) (arg ** rest')
         | Yes Refl | No nope = No (\v => nope $ lemma_snd_injective v)
  gdecEqd dr constraintsr (Arg a b) (deceqa, deceqb) (arg ** rest) (arg' ** rest')
         | No nope = No (\v => nope $ lemma_fst_injective v)
```

Figure 5.15: Decidable equality for described types with description `Arg`

The case of recursive arguments (*i.e.*, `Rec`) is similar to the case of ordinary arguments (see Figure 5.16). Since the equality of the recursive argument is independent of the equality of rest of the described type, both equalities can be checked at the same time. To check the equality of the recursive arguments gdecEq is called again with the description for the whole type and associated constraints, in addition to the values of the recursive arguments. If both the recursive arguments and the rest of the described type are equal than the result is that the data is equal, otherwise a contraproof is composed using the necessary lemma.

```
gdecEqd dr constraintsr (Rec j d) ((), constraints)  (rec ** rest) (rec' ** rest')
   with (gdecEq dr constraintsr rec rec',
                                 gdecEqd dr constraintsr d constraints rest rest')

  gdecEqd dr constraintsr (Rec j d) ((), constraints) (rec ** rest) (rec ** rest)
   | (Yes Refl, Yes Refl) = Yes Refl

  gdecEqd dr constraintsr (Rec j d) ((), constraints) (rec ** rest) (rec' ** rest')
   | (No nope, _)         = No (\v => nope $ lemma_fst_injective v)

  gdecEqd dr constraintsr (Rec j d) ((), constraints) (rec ** rest) (rec ** rest')
   | (_, No nope)         = No (\v => nope $ lemma_snd_injective v)
```

Figure 5.16: Decidable equality for described types with description Rec

Finally, for described types with description HRec, absurd is used to dismiss the required implementation since the constraints are not satisfiable similar to how it was dismissed for generic pretty printing (see Figure 5.17).

```
gdecEqd dr constraintsr (HRec a j d) constraints (rec ** rest) (rec' ** rest') =
    absurd constraints
```

Figure 5.17: Decidable equality for described types with description HRec

Similarly to how a Show instance was implemented using gshow, Figure 5.18 shows how to implement a DecEq instance using gdecEq. The specific DecEq constraints pairDecEqConstrs have exactly the same shape of definition as for pairShowConstrs from Figure 5.5, the only difference being that the type signature accepts DecEq instead of Show.

```
instance DecEq Pair where
  decEq = gdecEq PairDesc pairDecEqConstrs
```

Figure 5.18: Implementing the DecEq type class for Pair

5.1.4 Functorial mapping

One interesting generic algorithm that wouldn't work on described types just using `Desc` is the generic map function, since it requires an encoding of parameters. Therefore, the algorithm is implemented using the version extended with parameters `ParDesc`. The actual implementation of generic map `gmapp` is shown in Figure 5.19, and simply says that mapping a described type is simply the same as mapping its individual data components using `gmappd`.

```
gmapp : (d : ParDesc) -> (f : a -> b) ->
        DataPar d a -> DataPar d b
gmapp d f (Con x) = assert_total $ Con (gmappd d d f x)
```

Figure 5.19: Generic map

The function used to map a function on the individual components `gmappd` is presented in Figure 5.20. The implementation is straightforward and mostly iterates through the structure applying the generic mapping functions where possible, and the only two interesting cases are the ones for the value of the parameter type and for the functorially composed recursion. When a parameter value (described by `Par`) is met, then the function to mapped `f` is simply applied to that parameter and the mapping continues on the rest of the described type. For the functorially composed recursive arguments (described by `CompRec`), there are a couple of steps to be taken. First, a mapping function `map @{mapc} f` for the elements of the composed type is created by calling the associated `Functor` instance `mapc` for the composed functor `g`, which then has the type `g a -> g b`. Thereafter, this mapping function has the right type to be provided to the recursive call of `gmapp` and the recursive constructor argument can be mapped. Finally, it is possible to map the rest of the described type.

```
gmappd : (dr : ParDesc) -> (d : ParDesc) -> (f : a -> b) ->
         SynthesisePar d (DataPar dr) a -> SynthesisePar d (DataPar dr) b
gmappd dr Ret f () = ()
gmappd dr (Par d) f (x ** rest) = (f x ** gmappd dr d f rest)
gmappd dr (Rec d) f (x ** rest) = (gmapp dr f x ** gmappd dr d f rest)
gmappd dr (CompRec g {ffunctor = mapc} d) f (x ** rest) =
    (gmapp dr (map @{mapc} f) x ** gmappd dr d f rest)
gmappd dr (Arg c d) f (x ** rest) = (x ** gmappd dr (d x) f rest)
```

Figure 5.20: Generically mapping the data components of a described type

The `Functor` instance for a type `Nested`, that is described by `NestedPD` from Figure 4.8, is presented in Figure 5.21. The implementation simply passes the description `NestedPD` to the generic map function `gmapp`, which is enough since that implementation doesn't require any specific constraints.

```
instance Functor Nested where
  map = gmapp NestedPD
```

Figure 5.21: Implementing the `Functor` type class for `Nested`

5.2 Algorithms with purely generic properties

5.2.1 Generic tag testing

One common pattern in the Idris standard library is to check whether a variable is of a particular constructor type, *e.g.*, to get the `head` of a list `xs` then `isCons xs` must be true and to convert a `Fin` to an integer using `fromInteger` then `isJust` is used to ensure that the conversion is safe. Such patterns normally require setting up a new function for each particular constructor, however with the power of generic programming the pattern can be easily generalised into a function.

```
is : {d : TaggedDesc e ix} -> {i : ix} ->
     TData d i -> (l : CLabel) ->
     {default tactics { search 100; } t : Tag l e}
     -> Bool
is (Con (l' ** (t' ** rest))) l {t = t} with (l' `decEq` l)
  is (Con (l' ** (t' ** rest))) l {t = t} | (Yes prf) with
                                                        ((rewrite prf in t) `decEq` t')
    is (Con (l' ** (t' ** rest))) l {t = t} | (Yes prf)  | (Yes prf') = True
    is (Con (l' ** (t' ** rest))) l {t = t} | (Yes prf)  | (No contra) = False
  is (Con (l' ** (t' ** rest))) l {t = t} | (No contra) = False
```

Figure 5.22: Generic tag testing

Figure 5.22 shows a function `is` which can be used for generic constructor testing. The function uses the decidable equality on strings and values of type `Tag` to test whether a constructor the same tag to the one provided. An interesting feature of `is` is that it only requires the name of a constructor to be provided explicitly and will find the associated tag using proof search. This is where the way `Tag` was structured really shines: Since the name of the constructor to be found and the list of possible constructors are part of the type signature there is only one correct solution. Therefore, the proof search will never produce the wrong result, since that would not be well-typed.

5.2.2 Generic if expression

Another interesting function that could be made generically is a generic version of the **`if`**-expression. Normally the **`if`**-expression only works with conditions of type `Bool`, however the function `gif` presented in Figure 5.23 works with any datatype with at least two constructors. If the condition provided matches the first constructor of the datatype that had declared it, then the result is the false branch; otherwise the result of the function is the true branch.

```
gif : {d : TaggedDesc (l1 :: l2 :: e) ix} ->
      TData d i -> Lazy p -> Lazy p -> p
gif (Con (l ** (TZ ** rest)))    _ fb = fb
gif (Con (l ** (TS t ** rest)))  tb _ = tb
```

Figure 5.23: Generic **`if`**-expression

5.3 Scrapping your dependently-typed boilerplate is hard

The original purpose of this section was to present how I planned to implement a Uniplate-style (Mitchell and Runciman 2007) framework, as an example of how diverse the presented generic encoding is. However, I found out while trying to implement the framework that there was no straightforward solution in a dependent types setting. Instead, I will dedicate this section to presenting an informal explanation of the challenges that arise when trying to implement such framework. Please note that the Uniplate-related figures presented in this section are translated from Haskell to Idris.

5.3.1 Uniplate in 5 minutes

The goal of Uniplate and other Scrap Your Boilerplate-style (Lämmel and Peyton Jones 2003) frameworks is to present practical operations that permit generic traversal of datatypes. Specifically, these types of frameworks provide operations that are type directed, instead of focusing on particular data elements. For example, a Uniplate operation could be to extract all expressions from some given input statement, compared to ordinary functions which would focus on working with particular elements of a statement such as the condition expression and two branches of an **`if`**-statement. This allows many operations which often need to traverse deeply nested structures such as ASTs, to be succinctly written since only the necessary parts can be referred to.

While Uniplate provides a plethora of functions, there are two particularly interesting ones whose type signature is shown in Figure 5.24. The function `universeBi` allows the extraction of all elements of a target type `to`, from input data of the source type `from`. Complementarily, the function `transformBi` allows lifting of a function that homogeneously maps elements of a target type `to`, to a function that homogeneously maps elements of the source type `from`. The type signature of these functions require that there exists an instance of `Biplate` from the source type `from` to the target type `to`.

```
universeBi   : Biplate from to => from -> List to
transformBi  : Biplate from to => (to -> to) -> from -> from
```

Figure 5.24: A couple of the most interesting functions for Uniplate

```
class Uniplate to => Biplate from to where
  biplate : from -> (List to, List to -> from)
```

Figure 5.25: The `Biplate` type class

The interface for the `Biplate` type class is presented in Figure 5.25. One constraint on the `Biplate` type class is that the target type `to` must implement another type class `Uniplate`. The type class `Uniplate` provides a function `uniplate` which is similar to the `biplate` function that is provided by `Biplate` except the target datatype `to` and the source datatype `from` are restricted to be the same. The signature of the `biplate` function states that given input data of the source type `from`, then it should be possible to extract a list of the elements of the target type `to`, and given a list of new values with the same length it should be possible to create new data which conforms to the source type `from`. The whole interface provides a simple way to retrieve and create new elements for simple datatypes.

Example: Modelling blog posts

One place where something like Uniplate could be useful is when working with structured data models. For example, Figure 5.26 shows list of datatype declarations which represent a simplified model of blog posts. In that model a blog post is either a single blog post with a title and a publication date, or a summarising blog post that aggregates a series of other posts. Therefore in this model, blog posts form a tree-like structure.

```
data Title = MkTitle String

data Timestamp = MkTimestamp Int

data Post = Single Title Timestamp
          | Aggregate (List Post)
```

Figure 5.26: A simplified model over blog-posts

Figure 5.27 shows a couple of interesting operations on the blog post model, which utilises the Uniplate functions from Figure 5.24. The function `timestamps` would extract all creation dates that exist in the input blog post using `universeBi`, no matter how aggregated the blog posts are. Similarly, the function `capitaliseTitles` capitalises all the titles of the input blog posts, by applying `transformBi` on a function `capitalise` of type `Title -> Title`.

```
timestamps : Post -> List Timestamp
timestamps = universeBi

capitaliseTitles : Post -> Post
capitaliseTitles = transformBi capitalise
```

Figure 5.27: Interesting operations using Uniplate on blog-post model

Usually, the `Biplate` type class can automatically be derived in Haskell if a datatype already supports the necessary generic encoding; that is, if it has instances for the `Typeable` and `Data` type classes, which can also be derived. However, it is also possible to manually implement the necessary instances, and an example of an instance for `Biplate Post Timestamp` is shown in Figure 5.28. Since the `Post` datatype has two constructors, there are two cases to handle. The case for the constructor `Single` simply extracts the timestamp and returns it as a singleton list, and when to reconstruct the datatype the constructor `Single` is used on the same title as given originally and the newly provided timestamp. Since `Aggregate` doesn't contain any timestamps nothing is extracted, and when it is time to reconstruct the same element as the input is returned.

```
instance Biplate Post Timestamp where
  biplate (Single ttl ts)  =
    ([ts],
     \[ts'] => Single ttl ts')

  biplate (Aggregate psts) =
    ([],
     \_   => Aggregate psts)
```

Figure 5.28: An instance of `Biplate` for working with timestamps in blog posts

5.3.2 Automatic deriving

As mentioned in Section 5.3.1, one of the more useful features is the ability to automatically derive instances for the `Uniplate` and `Biplate` type classes. However, as I will present in this section, such thing is not completely obvious how to do this correctly in a setting with dependent types. In fact, the Uniplate documentation (Mitchell 2014) already warns that creating instances (even manually, not just by deriving) for polymorphic types could already produce incorrect results:

> When defining polymorphic instances, be carefully to mention all potential children. Consider `Biplate Int (Int, a)` - this instance cannot be correct because it will fail to return both `Int` values on `(Int, Int)`. There are some legitimate polymorphic instances, such as `Biplate a [a]` and `Biplate a a`, but take care to avoid overlapping instances.

The type casing challenge

The first challenge that arises when trying to generically derive instances, is that pattern matching on types is needed in order to identify which elements belong to the target type. However, Idris does not permit pattern matching on types for two reasons: it would break parametricity and it would significantly hurt runtime performance.

Therefore, something like `Typeable` for Haskell is needed in order to recover type information at compile time. Traditionally, the Universe pattern (Altenkirch, McBride, and Morris 2007) is used in dependently typed languages as an analogue to provide an internal encoding of a limited set of types that can be matched on. This is done by having a set of constructors called *codes* in some type `U` to represent these types, and then an interpretation function `El` which takes elements of `U` and translates them to actual types. However, this can be problematic when types need to depend on values, since dependencies must be encoded as functions from interpreted codes to other codes. Since it is not possible to pattern match the interpretation of codes which are actual types, it is not usually possible to recover the correct form of the type information at runtime. In summary, the author has not been able to find a good encoding for an external way to do type casing for dependent types, and there are both proof theoretic and performance-related problems if such system is included in the core language.

Distinguishing between the kind of data

Assuming it was possible to have a way to do something similar to type casing, there are still a challenge left regarding automatic deriving of relevant Uniplate instances. In dependently-typed programming languages, terms are not just used to contain data used at runtime, but sometimes they also contain data that is needed at type level such as when indexing datatypes. The challenge occurs when the two kinds of data intermix in datatypes when trying to derive the relevant `Biplate` and `Uniplate` instances.

For example, in the type `Vec Nat n` there are natural numbers used both as elements and as indices of the datatype. However, it is not generally possible to distinguish between the `Nat` element that is used as index and the `Nat` member elements, and when trying to extract all natural numbers in order to implement `Biplate (Vec Nat n) Nat` it is hard to only get the relevant member data. Worse yet, it is not possible to reconstruct new data of type `Vec` if given an arbitrary `Nat` which should act as index, since that wouldn't necessarily fit the definition of the datatype.

One idea would be to try to completely avoid any data members which appear at type level, however such solution cannot be generally applied. As an example see Figure 5.29 which presents a list which enforces a less-than ordering on the elements. In such datatype, there is no difference between the data members and the members used as indices. If the data members were omitted, then there is nothing interesting left to operate on. In summary, it is hard to do automatic deriving for any arbitrary indexed datatype and therefore they must be created manually in applicable cases in order to have a correct implementation.

```
data OList : Nat -> Type where
  Nil  : {n : Nat} -> OList n
  (::) : {n : Nat} ->
         (head : Nat) -> {ok : n <= head} ->
         (tail : OList head) ->
         OList n
```

Figure 5.29: A datatype describing a list with ordered elements

5.3.3 Generic traversal

Even if assuming that there somehow is a relevant `Biplate` instance for an indexed datatype, there would still be challenges with regards to generic querying and transformation. If one examines the type signature of the generic traversal functions `universeBi` and `transformBi`, one can see that they force types to be monomorphic. While there are many interesting operations that can be achieved on a non-indexed type like `List Nat`, the available operations to be done on a monomorphic indexed type like `Vec Nat Z` are far less interesting.

Therefore it is desirable that the generic traversal functions can have polymorphic indices which can change values such that they fit the relevant contained data. Figure 5.30 shows an updated version of these operations which allow individual elements to have different indices, and transformations to change the indices. The changes require a minor update to the `Biplate` interface such that the target type `to` has type `ix -> Type` instead of just `Type`.

Given the update interface and a fitting implementation of `Biplate`, the `universeBi` function should work adequately. However, there is something not completely right about the `transformBi` function; surely, it should not be possible to change arbitrary indices of a datatype? Unfortunately no, if such a function was allowed the type system would be inconsistent since it would be possible to break datatype invariants.

```
universeBi  : {from, ix : Type} -> {to : ix -> Type} ->
              {default %instance bip : Biplate from to} ->
              from -> List (i : ix ** to i)
transformBi : {from, ix : Type} -> {to : ix -> Type} ->
              {default %instance bip : Biplate from to} ->
              ((i : ix ** to i) -> (j : ix ** to j)) ->
              from -> from
```

Figure 5.30: Updating Uniplate functions to allow changes in the index of the target type

```
data TwoVec : Nat -> Type where
  MkTwoVec : {n : Nat} ->
             Vec Nat n -> Vec Nat (S n) ->
             TwoVec n
```

Figure 5.31: An example datatype that contains two vectors, where one has exactly one more element

For example, Figure 5.31 presents a datatype which contains two vectors and enforces the constraint that the second vector, must have exactly one element more than the first one. Figure 5.32 presents a simple function that appends a given input vector to itself. A problem occurs if a user tries to call `transformBi double xs`, since such function would try to double the size of all vectors in the input data `xs`. The result would be that both vectors would have twice the length, thus making one of the vectors having two elements more than the other breaking the invariant of the datatype.

```
double : (n : Nat ** Vec Nat n) ->
         (m : Nat ** Vec Nat m)
double (n ** xs) = (n + n ** xs ++ xs)
```

Figure 5.32: A function that appends a vector to itself

One suggestion to mitigate the problem would be to somehow separate actual data from constraints in the transformation, and then check that the resulting structure still conforms to the necessary invariants. However, such a solution is hard to generalise into a proper type signature and also it might not be apparent how such constraints could be formed. For example the `OList` shown in Figure 5.29 seems in fact to be the best way to enforce an ordering on list elements.

Chapter 6

Optimising Idris for flight

The generic encoding of datatypes via descriptions provided a powerful and flexible way to implement algorithms once for many datatypes as shown in Chapter 5. This could save the programmer the time to implement many algorithms each time a new datatype was declared, and at the same time lessen the probability that a bug is made.

However, the presented encoding comes at a price: it adds an overhead to the datatype which is linear to the size of the constructed data. This can significantly hurt the runtime performance, and therefore it is desired to find a way to optimise the generated code such that the presented framework is a viable option for realistic programs. Section 6.1 presents an analysis on how the encoding adds overhead to the constructed data and which parts could be optimised away. Section 6.2 presents an extension to the presented description type that enables erasure of arguments. Finally, Section 6.3 presents an informal description of a partial evaluation-based algorithm that enables specialisation of datatypes.

6.1 Analysing the encoding overhead

To examine the overhead which incurs when constructing described types, the following paragraphs will present a few simple examples of constructed data from described types and analyse the individual components.

6.1.1 The price of genericity

The case of informationless data

In the presented description scheme, even the simplest kind of data get complex results when converted to described types. For example, Figure 6.1 represents the described version of the boolean value `True`. What can be observed is that there is some additional data required to make it work in a described context. The constructor `Con` is required to build a possibly recursive described type, the label `"True"` and the dependent tag `TS TZ` are required to choose the type of data constructed, `Refl` is used to enforce restrictions on indices and all these arguments (of `Con`) must be wrapped in several dependent pairs to store the data. This is a stark contrast to simply using the constructor that originally was represented, namely `True`, which satisfies the required type by definition.

```
Con ("True" ** (TS TZ ** Refl))
```

Figure 6.1: The described version of the constructor `True` from `Bool`

Another example representing the described version of the vector `[42]`, is presented in Figure 6.2. The shape of the data is similar to the shape of the described version of `True`, in that it uses `Con` whenever data needs to be constructed, is immediately followed by the relevant constructor label and tag and always ends with `Refl` after containing the relevant constructor arguments. Probably, the more interesting part is the constructor arguments to the described version of `Cons` which are `0` and `42`. Recall from Figure 2.15, that `0` represents the length of the recursive argument and `42` represents data of the parameter type. Usually, the length of the recursive argument can be erased by Idris (Tejiščák 2014), since it can be inferred from the rest of the arguments. If it was not erased the size of the list would be quadratic, since the natural numbers which are used as indices use a unary representation, and a natural number must appear at every application of `Cons`. However, the encoding presented does not make it possible to represent erasable arguments, and Section 6.2 discusses what changes are necessary to achieve such feature.

However, since ordinary constructors and their described equivalent are isomorphic, the described versions do not strictly contain more information. This is because the data in the described type is all statically

```
Con ("Cons" ** (TS TZ ** (0 ** (42 **
   (Con ("Nil" ** (TZ ** Refl))
 ** Refl)))))
```

Figure 6.2: The described version of `[42]` of type `Vec Nat 1`

known and therefore not by itself interesting. The only thing that differs is the form, where the described versions are more suitable to be used with generic algorithms. This difference however comes at a price, it requires storing additional data with increasingly more references; something which could result in a significant performance loss at runtime if not optimised. It would therefore be desirable to use partial evaluation techniques when possible to reduce the size of the datatypes at runtime, and possible improve the performance of algorithms that use these structures.

It is all in the details

The figures presented in the previous paragraphs, do not provide a complete overview of the data there is because implicit parameters were not shown. In fact, when examining the implicit parameters it can observed that things like parameter and index values are usually stored in the data too.

For example, Figure 6.3 takes a look again at the described version of `True` (from Figure 6.1) but this time with the implicit arguments displayed. The first thing to notice is that in addition to the other data, when creating a described type using `Con`, the type of indices, the index value and the description for the whole type must also be provided as arguments. Additionally, the constructor tags also contain information about the current label, and the rest of the possible labels as implicit parameters.

```
Con {ix = ()} {i = ()} {d = Arg ...}
    ("True" **
      (TS {l = "True"} {e = "True" :: [] ...} {l' = "False"}
          (TZ {l = "True"} {e = [] ...})
            ** Refl))
```

Figure 6.3: The described version of `True` with implicit parameters displayed

Figure 6.4 shows similarly the described version of `[42]` from Figure 6.2 with implicit parameters described. One thing to notice that the type of indices and the description are provided for each application of `Con`. Similarly, alot of data seems replicated like the list of labels for tags. Looking at these examples, it would seem that the encoding might prove to provide a bigger overhead than initially assumed.

```
Con {ix = Nat} {i = 1} {d = Arg ...}
  ("Cons" **
     (TS {l = "Cons"} {e = "Cons" :: [] ...} {l' = "Nil"}
       (TZ {l = "Cons"} {e = [] ...}) **
         (0 ** (42 ** (Con {ix = Nat} {i = 0} {d = Arg ...}
           ("Nil"
             ** (TZ {l = "Nil"} {e = "Cons" :: [] ...}
             ** Refl))
         ** Refl)))))
```

Figure 6.4: The described version of `[42]` with implicit parameters displayed

Luckily, since many of these implicit arguments are inferable from the type signature, Idris can use various optimisation techniques like collapsing (Brady, Mcbride, and Mckinna 2004) and erasure (Tejiščák 2014) such that they do not affect the runtime performance. However, the type system must still be able to type check this kind of data and elaborating described types would be significantly slower than their equivalent constructors because of all the arguments that needs to be inferred and checked. Therefore it might be desirable to somehow limit the amount of time rechecking the same kind of structure. One solution could be to provide aliasing functions [1] just as the ones presented in Chapter 4, which could be used whenever the work involves specific described types. Thus, for specific datatypes many arguments only needs to be elaborated once for each alias, instead of every time new data is constructed.

Annotating static data

There are several ways one could analyse data to find what parts of data is static. For example, it could be possible to use the minimal function graph-based analysis used in Section 3.4. However, many of the described type can be generated from ordinary datatype declarations as presented in Chapter 4, and it may already be known at generation time what some of the static data in the generic representation is. Therefore,

[1] Unfortunately, at present Idris does not support pattern aliases so this would only work when constructing datatypes

it might be worthwhile to examine the generated constructor aliases, and try to identify what static data might be optimised already at those points.

```
Nil : Vec a Z
Nil = Con ("Nil" ** (TZ ** Refl))

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

Figure 6.5: Annotating the static (underlined) and erasable (dotted underline) parts of the constructors for the described version of `Vec`

Constructor aliases for the described version of `Vec` are shown in Figure 6.5, along with annotations to show the static (solid underline) and erasable (dotted underline) parts of the data. For `Nil` all the provided data to `Con` is statically provided, which makes sense since `Nil` is usually a nullary constructor and does not contain any data. The only data that is not static for `Cons` is `n`, `x` and `xs`, which mimics the data stored in the ordinary version of `Cons`. Since `n` is inferable from the type signature of `Cons`, it would usually be possible for Idris to erase such value if was otherwise not stored. In summary, there is plenty of room to specialise constructors to remove static and erasable data already at the point where aliases are generated, and perhaps more so if a more elaborate analysis was made.

6.2 Preparing descriptions for erasure

In Section 6.1, it was mentioned that it would be desirable if there was a way to synthesise types with erasable data from descriptions. This Section presents a few extensions to the description type that enables the user to exploit erasability as an optimisation technique.

```
data Erasure = None | Erasable
```

Figure 6.6: An annotation to specify erasure of arguments

Figure 6.6 presents a simple type `Erasure` which can be used to specify the erasability of arguments. The constructor `None` specifies that no

```
Arg : Erasure -> (a : Type) -> (a -> Desc ix) -> Desc ix
```

Figure 6.7: Extending the description constructor `Arg` to support an erasure annotation as argument

erasure should happen, and the constructor `Erasable` specifies that an argument may be erased.

To enable described types to have erasable arguments, the description must be slightly modified. Figure 6.7 shows an updated version of `Arg`, which has an added first argument requiring an annotation of type `Erasure`.

```
data Exists : (p : a -> Type) -> Type where
    Evidence : .(x : a) -> (pf : p x) -> Exists p
```

Figure 6.8: A dependent pair type `Exists`, with the first component being erasable

The next step is to convert described types with description `Arg` to ordinary types that satisfy the required properties of the given erasure annotation. The target type to convert described types with erased arguments to is going to be `Exists` and is presented in Figure 6.8. The type `Exists` is similar to the ordinary dependent pair, except the first argument of its constructor `Evidence` is marked to be erasable (by using `.` in front of the argument)

```
Synthesise (Arg None a d) x i =
   (arg : a ** Synthesise (d arg) x i)
Synthesise (Arg Erasable a d) x i =
 Exists (\arg : a => Synthesise (d arg) x i)
```

Figure 6.9: Synthesising the different versions of `Arg`, depending on erasure properties

The actual translation of the different kinds of `Arg` values to types using `Synthesised` is shown in Figure 6.9. The translation of non-erasable arguments is the same as before, but values having the `Erasable` annotation are translated to `Exists` instead to ensure the correct erasure.

Finally, Figure 6.10 shows an updated version of `Cons` where the value `n` can be erased. Notice however, that the resulting value must use `Evidence` instead of the ordinary dependent pair constructor when constructing erasable arguments.

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

Figure 6.10: Updated version of alias `Cons`, now using `Evidence` for storing erasable argument `n`

6.3 Sketching out an algorithm for specialising described types

This section will sketch out an algorithm for removing some of the static data identified in Section 6.1, by utilising some of the techniques presented in Chapter 3. The core idea of the algorithm to exploit the additional information provided by the power dependent type system of Idris, to eliminate some of the static overhead present in some datatypes. While this algorithm seem to be usable in the case of specialising described types, it is not limited just to that application and could probably be used for other kinds of datatypes, *e.g.*, one could imagine specialising `Fin 2` to `Bool`.

6.3.1 Specialisation of static parameters

The first step towards specialising datatypes, is by specialising parameters when they are statically provided. This is done by creating a new datatype declaration with almost identical definition, except that the parameters are fixed to the provided values and any recursive argument refers to the newly specialised datatype.

There a couple of reasons why it is desirable to specialise parameters, but not indices. The first reason is that different branches could differ in the provided index—like `Nil` and `Cons` of `Vec`—but it would still be desirable to keep both branches as part of the same type family. The other reason is that since indices may change, the specialisation process may not terminate. For example, specialising the `Nat` index `n` for any instance of the type family `Vec` requires creating an unbounded amount of datatypes (`Vec a 0`, `Vec a 1`, `Vec a 2`, etc.).

Figure 6.11 shows an example where `Data` (from Figure 2.17) is specialised with the type of indices `ix` set to `Nat`, and the description `d` set to `VecD Int`. Notice that the `Synthesise` argument of `Con` previously accepted a recursive reference `Data (VecD Int)` as argument, which must be changed to refer to the specialised version `Data__Vec_Int`.

```
data Data__Vec_Int : Nat -> Type where
  Con : Synthesise (VecD Int) Data__Vec_Int i -> Data d i
```

Figure 6.11: Specialising `Data` with parameter `ix` having value `Nat`, and parameter `d` having value `VecD Int`

Since some arguments of `Synthesise` in the constructor `Con` are provided statically now, it is possible to normalise the expression. Figure 6.12 shows that normalised version of `Synthesise` with regards to the provided arguments. Notice that it is not possible to further normalise the `Synthesise` application on `switchDesc`, since `switchDesc` is dependent on dynamic arguments `l` and `t`.

```
data Data__Vec_Int : Nat -> Type where
  Con : (l : CLabel ** (t : Tag l ["Nil", "Cons"] **
          Synthesise (switchDesc (
                              Ret Z
                            , Arg Erasable Nat (\n =>
                                Arg None Int (\arg =>
                                  Rec n (Ret (S n))))
                            ,()
          ) l t) i)) -> Data d i
```

Figure 6.12: Normalising the `Synthesise` call from Figure 6.11

6.3.2 Unboxing nested references

Taking a value of a dependent pair type as argument may be less optimal than simply accepting the arguments directly, since it requires an additional allocation or *box* in memory. This doesn't only increase the storage requirements, but may also hurt runtime performance since it would require chasing multiple pointers.

One particular technique used to avoid such situation is called *unboxing* (Peyton Jones and Launchbury 1991; Leroy 1997), which is used to inline references to other datatypes. To unbox a datatype, one takes each reference to it and replace it by its individual components. For example, to unbox a dependent pair `(x:a ** b)`, one takes the arguments of its constructor—of types `a` and `b a` respectively— and uses these as arguments in the enclosing constructor.

Figure 6.13 shows the unboxing of the nested dependent pair required of argument to three ordinary arguments of the constructor `Con`.

```
data Data__Vec_Int : Nat -> Type where
   Con : (l : CLabel) -> (t : Tag l ["Nil", "Cons"]) ->
         (arg : Synthesise (switchDesc (
                              Ret Z
                            , Arg Erasable Nat (\n =>
                                Arg None Int (\arg =>
                                   Rec n (Ret (S n))))
                            ,()
          ) l t) i) -> Data d i
```

Figure 6.13: Unboxing the dependent pair type argument to ordinary arguments of `Con`

These resulting arguments are the label `l`, the tag `t` and the rest of the description `arg`.

6.3.3 Trickery applies here

It would seem that after the Section 6.3.2, that all apparent static data has been eliminated and that we are seemingly stuck in doing further partial evaluation. However, save some inlining of data not much has been achieved, but as observed in Section 6.1 there is still some data which appears to be static after `l` and `t` is provided.

Jones, Gomard, and Sestoft (1993) suggests a technique called *the trick* where dynamically provided data can be partially evaluated if it is finitely enumerable. This is done by creating new code for each possible value of the enumerable data, and then further partially evaluating each new branch. Then a mechanism is used to select the correct branch dependent on the value of the enumerable data at runtime.

```
data Data__Vec_Int : Nat -> Type where
   Con_Nil : (prf : i = Z) -> Data_Vec_Int i
             «l = "Nil", t = TZ»
   Con_Cons : .(n : Nat) -> (arg : Int)
              -> (rec : Data_Vec_Int n) -> (prf : i = S n)
              -> Data_Vec_Int i
              «l = "Cons", t = TS TZ»
```

Figure 6.14: Splitting constructors based on the different possible values of `Tag` (with additional unboxing)

For the presented example in Figure 6.13, the list of possible constructors `["Nil","Cons"]` is known and so the value `t` is finitely enumerable and by dependency the value `l` becomes static. The specialised version using the trick on `t`, is presented in Figure 6.14 which has two new constructors representing the branches for the two valid values of `t`, namely `TZ` and `TS TZ`.

The values displayed between angle quotes are the static data specialised with their correspondingly assigned values. These values are not stored at runtime, but may be used by the compiler when specialising algorithms to substitute if these algorithms are dependent on these values. For example, the generic pretty printing algorithm depends on knowing the labels and thus requires substituting either `"Nil"` or `"Cons"` where `l` is used.

6.3.4 Index substitution

The final optimisation step that can be performed on our example, is to convert simple obvious restrictions on parameters given by propositional equality to ordinary indices. Figure 6.15 shows such optimisation where the parameter `i` has been specialised to the restricted value in each branch—`Z` for `Con_Nil` and `S n` for `Con_Cons`—and the corresponding equality arguments has been removed.

```
data Data__Vec_Int : Nat -> Type where
    Con_Nil :   Data_Vec_Int Z
              «l = "Nil", t = TZ, i = Z»
    Con_Cons : .(n : Nat) -> (arg : Int)
               -> (rec : Data_Vec_Int n)
               -> Data_Vec_Int (S n)
               «l = "Cons", t = TS TZ, i = S n»
```

Figure 6.15: Eliminating equality restrictions on type arguments, by inlining the expected values as indices

6.3.5 Expansion in functions

Similar to the other data-oriented specialisation algorithms presented in Chapter 3, dependent functions must also be specialised to support the updated structure. This includes creating new case trees when pattern matching or constructing objects of the specialised type, and then adjusting the control flow to match the new cases.

As an example of how to specialise functions to support the specialised datatypes, see Figure 6.16. The figure presents a specialised version of gshow with regards to the description VecD Int and the relevant constraints. Please notice, that the specialisation could not further reduce the d and cstrs arguments which represent the description and constraints for constructor arguments, since these are dependent on the label and tag variables which are bound dynamically.

```
gshow__Vec_Int : {i : Nat} -> (x : Data (VecD Int) i) -> String
gshow__Vec_Int (Con (label ** (tag ** rest))) =
    let d = switchDesc (Ret Z
                       , (Arg Erasable Nat (\n=>
                            Arg None Int (\x=>
                              Rec n (Ret (S n))))
                       , ())) label tag
    in let cstrs = switch ["Nil", "Cons"]
                        (\l, t => Constraints1 Show d)
                        (()
                       , (%instance, (\n=>
                           (%instance, (\x=>
                              ((), ()))
                       , ())) label tag
       in label ++ (assert_total $ gshowd__Vec_Int d cstrs rest)
```

Figure 6.16: Specialising gshow with regards to the description VecD Int and its related constraints

In order to be able to use the specialised datatype in gshow, new cases has to be created which mirrors the cases made using “the trick”. Figure 6.17 shows the specialisation of gshow from Figure 6.16, where new cases has been made by enumerating possible values of tag (TZ and TS TZ), similarly to when specialising Data__Vec_Int. In the figure, two minor non-essential adjustments have been done. The first is that it is assumed that gshow has been updated such that erasable arguments are not pretty printed, and thus neither is the argument n. The other adjustment is that after specialising the expression to show arguments has the form show @{%instance} x, which has been reduced to simply show x since they are equivalent.

```
gshow__Vec_Int : {i : Nat} -> (x : Data (VecD Int) i) -> String
gshow__Vec_Int (Con ("Nil" ** (TZ ** Refl))) = "Nil" ++ ""
gshow__Vec_Int (Con ("Cons" ** (TS TZ **
                            Evidence n (x ** (xs ** Refl))))) =
  "Cons" ++ " " ++ parenthesise (show x) ++ " " ++
      parenthesise (gshow__Vec_Int xs) ++ ""
```

Figure 6.17: The specialised version of gshow after branching on tag and further applying partial evaluation techniques

Since the function argument now has right shape, it is now possible to replace the generic datatype argument with the specialised one. Figure 6.18 shows an updated version of Figure 6.17, where the different branches now uses the specialised constructors Con_Nil and Con_Cons and the concatenation expression has been slightly simplified. Finally, it can be observed that the finished result looks very similar to how a manually-written Show instance for Vec Int would have looked like, and therefore a similar performance characteristic is expected.

```
gshow__Vec_Int : {i : Nat} -> (x : Data__Vec_Int i) -> String
gshow__Vec_Int Con_Nil = "Nil"
gshow__Vec_Int (Con_Cons n x xs) =
  "Cons " ++  parenthesise (show x) ++ " " ++ parenthesise (gshow__Vec_Int xs)
```

Figure 6.18: The final specialised version of gshow, using the specialised version of the datatype Data__Vec_Int

Chapter 7

Discussion

7.1 Future work

7.2 Conclusion

Bibliography

Altenkirch, Thorsten, Conor McBride, and Peter Morris (2007). "Generic programming with dependent types". In: *Datatype-Generic Programming*. Springer, pp. 209–257.

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.

Bernardy, Jean-Philippe. "A theory of parametric polymorphism and an application". PhD thesis.

Bernardy, Jean-Philippe, Patrik Jansson, and Ross Paterson (2010). "Parametricity and dependent types". In: *ACM Sigplan Notices*. Vol. 45. 9. ACM, pp. 345–356.

Bird, Richard and Lambert Meertens (1998). "Nested datatypes". In: *Mathematics of program construction*. Springer, pp. 52–67.

Boquist, Urban (1999). "Code optimisation techniques for lazy functional languages". PhD thesis.

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

Brady, Edwin, Conor Mcbride, and James Mckinna (2004). "Inductive families need not store their indices". In: *Types for Proofs and Programs, Torino, 2003, volume 3085 of LNCS*. Springer-Verlag, pp. 115–129.

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.

Dussart, Dirk, Eddy Bevers, and Karel De Vlaminck (1995). "Polyvariant Constructor Specialisation". In: *Proceedings of the 1995 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation*. PEPM '95. La Jolla, California, USA: ACM, pp. 54–65. ISBN: 0-89791-720-0. DOI: 10.1145/215465.215554. URL: http://doi.acm.org/10.1145/215465.215554.

Dybjer, Peter (1997). "Inductive Families". In: *Formal Aspects of Computing* 6, pp. 440–465.

Hughes, John (1999). "A Type Specialisation Tutorial". English. In: *Partial Evaluation*. Ed. by John Hatcliff, TorbenÆ Mogensen, and Peter Thiemann. Vol. 1706. Lecture Notes in Computer Science. Springer Berlin Heidelberg, pp. 293–325. ISBN: 978-3-540-66710-0. DOI: 10.1007/3-540-47018-2_12. URL: http://dx.doi.org/10.1007/3-540-47018-2_12.

Jansson, Patrik and Johan Jeuring (1997). "PolyP—a Polytypic Programming Language Extension". In: *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '97. Paris, France: ACM, pp. 470–482. ISBN: 0-89791-853-3. DOI: 10.1145/263699.263763. URL: http://doi.acm.org/10.1145/263699.263763.

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.

Jones, Neil D. and Alan Mycroft (1986). "Data Flow Analysis of Applicative Programs Using Minimal Function Graphs". In: *Proceedings of the 13th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*. POPL '86. St. Petersburg Beach, Florida: ACM, pp. 296–306. DOI: 10.1145/512644.512672. URL: http://doi.acm.org/10.1145/512644.512672.

Lämmel, Ralf and Simon Peyton Jones (2003). "Scrap your boilerplate: a practical design pattern for generic programming". In: *ACM SIGPLAN Notices*. Vol. 38. 3. ACM, pp. 26–37.

Leroy, Xavier (1997). *The effectiveness of type-based unboxing*. Tech. rep. Boston College, Computer Science Department.

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*.

Milner, Robin, Mads Tofte, and David Macqueen (1997). *The Definition of Standard ML*. Cambridge, MA, USA: MIT Press. ISBN: 0262631814.

Mitchell, Neil (2014). *Haskell Documentation for package Data.Generics.Uniplate.Direct*. URL: `http://hackage.haskell.org/package/uniplate-1.6.10/docs/Data-Generics-Uniplate-Direct.html`.

Mitchell, Neil and Colin Runciman (2007). "Uniform boilerplate and list processing". In: *Proceedings of the ACM SIGPLAN workshop on Haskell workshop*. ACM, pp. 49–60.

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`.

Mogensen, Torben Ægidius and Peter Sestoft (1997). "Partial evaluation: introduction what is partial evaluation?" In: *Encyclopedia of computer science and technology*. Ed. by A. Kent and J. G. Williams. Suppl. 22. Vol. 37. Marcel Dekker Incorporated, pp. 247–279.

Nielson, F. (Dec. 1989). "Two-level Semantics and Abstract Interpretation". In: *Theor. Comput. Sci.* 69.2, pp. 117–242. ISSN: 0304-3975. DOI: `10.1016/0304-3975(89)90091-1`. URL: `http://dx.doi.org/10.1016/0304-3975(89)90091-1`.

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

Peyton Jones, Simon and John Launchbury (1991). "Unboxed values as first class citizens in a non-strict functional language". In: *Proceedings of the 5th ACM conference on Functional programming languages and computer architecture*. Springer-Verlag New York, Inc., pp. 636–666.

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.

Peyton Jones, Simon L. and David R. Lester (1992). *Implementing Functional Languages*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc. ISBN: 0-13-721952-0.

Reynolds, John C. (1983). "Types, Abstraction and Parametric Polymorphism". In: *IFIP Congress*, pp. 513–523.

Taha, Walid (2004). "A gentle introduction to multi-stage programming". In: *Domain-Specific Program Generation*. Springer, pp. 30–50.

Tejiščák, Matúš (2014). *Erasure by usage analysis*. URL: `https://github.com/idris-lang/Idris-dev/wiki/Erasure-by-usage-analysis`.

Wadler, Philip (1989). "Theorems for free!" In: *FUNCTIONAL PROGRAMMING LANGUAGES AND COMPUTER ARCHITECTURE*. ACM Press, pp. 347–359.

Wegman, Mark N. and F. Kenneth Zadeck (Apr. 1991). "Constant Propagation with Conditional Branches". In: *ACM Trans. Program. Lang. Syst.* 13.2, pp. 181–210. ISSN: 0164-0925. DOI: `10.1145/103135.103136`. URL: `http://doi.acm.org/10.1145/103135.103136`.

Westbrook, Edwin et al. (June 2010). "Mint: Java Multi-stage Programming Using Weak Separability". In: *SIGPLAN Not.* 45.6, pp. 400–411. ISSN: 0362-1340. DOI: `10.1145/1809028.1806642`. URL: `http://doi.acm.org/10.1145/1809028.1806642`.

Appendix A

Generation function

```
elabDescription :: [Int] -> Name -> PTerm ->
                   [(Docstring, [(Name, Docstring)], Name, PTerm, FC, [Name])] ->
                   ElabInfo -> Idris ()
elabDescription paramPos dn ty cons info = do
  elabDecl EAll toplevel labelsTyDecl
  elabDecl EAll toplevel labelsClauses
  elabDecl EAll toplevel descTyDecl
  elabDecl EAll toplevel descClauses
  elabDecl EAll toplevel aliasTyDecl
  elabDecl EAll toplevel aliasClauses
  mapM_ (elabDecl EAll toplevel) aliasCnssTyDecl
  mapM_ (elabDecl EAll toplevel) aliasCnssClauses
  where labelsTy :: PTerm
        labelsTy = PRef emptyFC (sNS (sUN "CEnum") ["Generic", "Prelude"])
        labelsName :: Name
        labelsName = SN . LabelsN $ dn
        labelsTyDecl :: PDecl
        labelsTyDecl = PTy emptyDocstring [] defaultSyntax emptyFC [TotalFn] labelsName labelsTy
        -- Extract names from constructors and map them to Idris lists
        labelsClauses :: PDecl
        labelsClauses =
          PClauses emptyFC [TotalFn] labelsName
```

```
          [PClause emptyFC labelsName (PRef emptyFC labelsName) []
            (mkList emptyFC (map (\(doc, adocs, cnm, cty, cfc, cargs)
              -> PConstant . Str . show $ cnm) cons)) []]
    descName :: Name
    descName = SN . DescN $ dn
    descTy :: PTerm -> PTerm
    descTy indexType =
      PApp emptyFC (PRef emptyFC (sNS (sUN "TaggedDesc") ["Generic", "Prelude"]))
        [pexp $ PRef emptyFC labelsName, pexp natZ, pexp indexType]
    descTyDecl :: PDecl
    descTyDecl = PTy emptyDocstring [] defaultSyntax emptyFC [TotalFn] descName (descTy (PRef emptyFC unitTy))
    descClauses = PClauses emptyFC [TotalFn] descName [PClause emptyFC descName (PRef emptyFC descName) []
              (switchDesc (foldr (flip (.) (\(_,_,_,term,_,_) -> descCns term) pairI) unitI cons)) []]
    natZ :: PTerm
    natZ = PRef emptyFC (sNS (sUN "Z") ["Nat", "Prelude"])
    natS :: PTerm -> PTerm
    natS t = PApp emptyFC (PRef emptyFC (sNS (sUN "S") ["Nat", "Prelude"])) [pexp t]
    unitI :: PTerm
    unitI = PRef emptyFC unitCon
    pairI :: PTerm -> PTerm -> PTerm
    pairI x y = PApp emptyFC (PRef emptyFC pairCon)
                  [pimp (sUN "A") Placeholder True, pimp (sUN "B") Placeholder True, pexp x, pexp y]
    eqRefl :: PTerm
    eqRefl = PApp emptyFC (PRef emptyFC eqCon) [pimp (sMN 0 "A") Placeholder True, pimp (sMN 0 "x") Placeholder True]
    dpairI :: PTerm -> PTerm -> PTerm
    dpairI x y = PApp emptyFC (PRef emptyFC existsCon)
                  [pimp (sUN "a") Placeholder True, pimp (sUN "P") Placeholder True, pexp x, pexp
```

```
                          y]
        tagZ :: PTerm
        tagZ = PRef emptyFC (sNS (sUN "TZ") ["Generic", "Prelude"])
        tagS :: PTerm -> PTerm
        tagS t = PApp emptyFC (PRef emptyFC (sNS (sUN "TS") ["Generic", "Prelude"])) [pexp t]
        tagFromNum :: Integer -> PTerm
        tagFromNum n | n == 0 = tagZ
                     | n >  0 = tagS (tagFromNum (n - 1))
        dataCon :: PTerm -> PTerm
        dataCon inn = PApp emptyFC (PRef emptyFC (sNS (sUN "Con") ["Generic", "Prelude"])) [pexp inn]
        switchDesc :: PTerm -> PTerm
        switchDesc consmappings = PApp emptyFC (PRef emptyFC (sNS (sUN "switchDesc") ["Generic", "Prelude"])) [pexp consmappings]
        descRet :: PTerm -> PTerm
        descRet ixval = PApp emptyFC (PRef emptyFC (sNS (sUN "Ret") ["Generic", "Prelude"])) [pexp ixval]
        descRec :: PTerm -> PTerm -> PTerm
        descRec ixval rest = PApp emptyFC (PRef emptyFC (sNS (sUN "Rec") ["Generic", "Prelude"])) [pexp ixval, pexp rest]
        descArg :: PTerm -> PTerm -> PTerm
        descArg typ rest = PApp emptyFC (PRef emptyFC (sNS (sUN "Arg") ["Generic", "Prelude"])) [pexp typ, pexp rest]
        dataTy :: PTerm -> PTerm -> PTerm
        dataTy datadesc ixval = PApp emptyFC (PRef emptyFC (sNS (sUN "Data") ["Generic", "Prelude"])) [pexp datadesc, pexp ixval]
        descCns :: PTerm -> PTerm
        descCns (PPi _ nm ty rest) = descCnsArg nm ty (descCns rest)
        descCns _                  = descRet unitI
        descCnsArg :: Name -> PTerm -> PTerm -> PTerm
        descCnsArg nm ty@(PApp _ (PRef _ nm') _) rest
          | simpleName dn == simpleName nm' = descRec unitI rest
          | otherwise = descArg ty (PLam nm ty rest)
        descCnsArg nm ty@(PRef _ nm') rest
```

```
          | simpleName dn == simpleName nm' = descRec unitI rest
          | otherwise = descArg ty (PLam nm ty rest)
        descCnsArg nm ty rest = descArg ty (PLam nm ty rest)
        aliasName :: Name
        aliasName = uniqueName dn [dn]
        aliasTyDecl :: PDecl
        aliasTyDecl = PTy emptyDocstring [] defaultSyntax emptyFC
            [TotalFn] aliasName PType
        aliasClauses :: PDecl
        aliasClauses = PClauses emptyFC [TotalFn] aliasName [
            PClause emptyFC aliasName (PRef emptyFC aliasName) []
                           (dataTy (PRef emptyFC descName) unitI)
                              []]
        aliasCnssTyDecl :: [PDecl]
        aliasCnssTyDecl = map (\(_,_,nm,ty,_,_) ->
                              PTy emptyDocstring [] defaultSyntax
                                 emptyFC [TotalFn] (aliasCnsNm nm) (
                                 aliasCnsTy ty)) cons
        aliasCnsTy :: PTerm -> PTerm
        aliasCnsTy ty@(PApp _ (PRef _ nm') args)
          | simpleName dn == simpleName nm' = PApp emptyFC (PRef
              emptyFC aliasName) args
        aliasCnsTy ty@(PRef _ nm')
          | simpleName dn == simpleName nm' = PRef emptyFC
              aliasName
        aliasCnsTy ty@(PPi pl nm ty' rest) = PPi pl nm (aliasCnsTy
             ty') (aliasCnsTy rest)
        aliasCnsTy ty = ty
        aliasCnsNm :: Name -> Name
        aliasCnsNm nm = uniqueName nm [nm]
        aliasCnssClauses :: [PDecl]
        aliasCnssClauses =
          map (\((_,_,nm,ty,_,_), i) ->
            let args = namePis . fst $ splitPi ty
            in PClauses emptyFC [TotalFn] (aliasCnsNm nm)
                 [PClause emptyFC (aliasCnsNm nm) (aliasCnsLhs nm
                    args) [] (aliasCnsRhs nm i args) []])
                    (zip cons [0..])
```

```

aliasCnsLhs :: Name -> [(Name, Plicity, PTerm)] -> PTerm
aliasCnsLhs nm args =
   (PApp emptyFC (PRef emptyFC (aliasCnsNm nm))
      (map (\(arg, _, _) -> pexp (PRef emptyFC arg)) args))
aliasCnsRhs :: Name -> Integer -> [(Name, Plicity, PTerm)] -> PTerm
aliasCnsRhs nm i args =
  dataCon
    (dpairI
      (PConstant . Str . show $ nm)
      (dpairI
      (tagFromNum i) (foldr (flip (.) (\(nm', pl, ty) -> PRef emptyFC nm') dpairI) eqRefl args)))

namePis :: [(Name, Plicity, PTerm)] -> [(Name, Plicity, PTerm)]
namePis = namePis' []
  where namePis' :: [(Name, Plicity, PTerm)] -> [(Name, Plicity, PTerm)] -> [(Name, Plicity, PTerm)]
    namePis' acc []                   = reverse acc
    namePis' acc ((nm, pl, ty):rest)  = namePis' ((uniqueName nm prevnm, pl, ty):acc) rest
       where prevnm :: [Name]
             prevnm = map (\(nm, _, _) -> nm) acc
```

Listing A.1: Generating relevant functions for working with described types