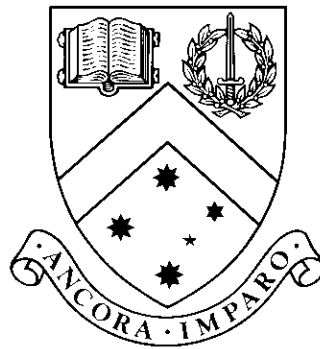


Generalising Data Description for Machine Learning

by

James Bardsley, BCompSc



Thesis

Submitted by James Bardsley

in partial fulfillment of the Requirements for the Degree of
Bachelor of Computer Science with Honours (1608)

Supervisor: Dr. Lloyd Allison

Clayton School of Information Technology
Monash University

November, 2006

© Copyright

by

James Bardsley

2006

Contents

List of Tables	v
List of Figures	vi
Abstract	vii
1 Introduction	1
1.1 Thesis Outline	2
2 Literature Review	3
2.1 Machine Learning	3
2.1.1 Existing Statistical Packages	4
2.1.2 Inductive Programming	4
2.2 Haskell	5
2.2.1 Functional Programming	5
2.2.2 Lazy Evaluation	6
2.2.3 Polymorphism	7
2.3 Meta Programming	9
2.4 Template Haskell	9
2.4.1 History	10
2.4.2 Syntax	12
2.4.3 Summary	14
2.5 Data	14
2.5.1 Types of Data	14
2.5.2 Bad/Missing Data	15
2.5.3 Summary	16

3	Methods	17
3.1	Type Generation	17
3.1.1	Compile Time I/O	17
3.1.2	Determining Types	18
3.1.3	Generating Types	21
3.2	Utilities	23
3.2.1	zipN/unzipN	23
3.2.2	Type Classes	25
4	Testing	31
4.1	Trained Monkey Data	31
4.2	Childhood Respiratory Disease Data	32
4.3	Cholesterol Level Data	33
4.4	Missing Person Data	34
5	Conclusion	35
5.1	Further Work	35

List of Tables

2.1	Sample data – swimmers	15
4.1	Sample trained monkey data	32
4.2	Sample childhood respiratory disease data	33
4.3	Sample cholesterol level data	33

List of Figures

2.1	An example higher order function.	6
2.2	Infinite list of Fibonacci numbers.	7
2.3	Instantiating a new variable type in the Enum type class.	8
2.4	Example AST	10
2.5	Demonstration of zip and unzip	11
2.6	A TH function for generating code for factorials	13
3.1	Example type representation list.	18
3.2	The type representation generation function	19
3.3	Swimming data in CSV format with example format specifiers	21
3.4	The createTypes function	22
3.5	Example zip function	24
3.6	Example unzip function	25
3.7	Show instantiation for a three-tuple	26
3.8	Read instantiation for a three tuple	27
3.9	Enum instantiation for a three-tuple	29
3.10	Bounded instantiation for a three-tuple	30

Generalising Data Description for Machine Learning

James Bardsley, BCompSc
jwbar3@csse.monash.edu.au
Monash University, 2006

Supervisor: Dr. Lloyd Allison
lloyd@csse.monash.edu.au

Abstract

The main aim of Inductive Programming is to make it possible to create solutions to new inductive inference problems with as little modification as possible to solutions for existing problems. This has initially been done through the creation of type classes which represent general statistical models. However, these models alone only provide a degree of generalisation, as the data-set being operated on must still be described through the creation of variable types to represent columns and the instantiation of these types in appropriate type classes. The creation of these types and the creation of functions to operate on them can be extremely tedious and is error prone when done manually. This thesis presents a method for overcoming this problem by using the Haskell meta-programming extension Template Haskell to automatically generate appropriate types by examining the contents of the data file being operated on. It also provides various tools for generating utility functions to help with handling the newly generated types.

Generalising Data Description for Machine Learning

Declaration

I declare that this thesis is my own work and has not been submitted in any form for another degree or diploma at any university or other institute of tertiary education. Information derived from the published and unpublished work of others has been acknowledged in the text and a list of references is given.

James Bardsley
November 7, 2006

Acknowledgments

I would like to thank and acknowledge the following people:

- My supervisor, Lloyd Allison, for his help throughout the project, for staying patient while I struggled to get to grips with Template Haskell, and for pushing me at times when I desperately wanted to procrastinate (more).
- My family for their help and support over the course of the year.
- Naomi Naismith for her patience and understanding.
- Julian Day, for proofreading everything I threw at him.

James Bardsley

Monash University
November 2006

Chapter 1

Introduction

Inductive Programming (IP) is a machine learning paradigm which aims to generalise as much as possible of machine learning so that solutions to new problems can be easily tailored. This has been done through the creation of type classes which represent general statistical models (Allison 2005). This project supports IP by providing mechanisms for generalising the creation of new types, functions and type class instantiations to aid in the “description” of data. This section gives a summary of the motivations for the project and the outline of the rest of the thesis.

One of the tasks required when a new data-set is being used for a machine learning application is the description of the data. This is done through the creation of variable types to represent columns. For example, if a column representing user satisfaction contained the categorical data fields “Good”, “Bad” and “Medium” then the following data type would be created to represent the column:

```
data Satisfaction = Good | Bad | Medium
```

As well as the newly generated types to represent columns, a type synonym – that is, a new name for an existing type – needs to be created to represent records. If a data set contained columns for customer name, customer ID number, total spent and the aforementioned satisfaction type then the type synonym for records in the data-set would be:

```
(String, Int, Float, Satisfaction)
```

In addition to the creation of new types to represent columns and new type synonyms to represent records, functions need to be written for the handling

of records - tasks such as dropping columns or reading in a record from the data file.

Although these type and function definitions can be written by hand it is not a pleasant task. It is extremely repetitive and as such it becomes prone to errors due to loss of concentration. This project provides a solution to this problem through the use of the experimental Haskell meta-programming extension Template Haskell to automatically generate the code for these definitions.

1.1 Thesis Outline

Chapter two discusses machine learning as the application domain for the project in more detail. It gives details on the project's implementation language, Haskell. It briefly introduces meta programming, then moves to discussion about Template Haskell, the meta programming extension to Haskell which was used in this project. Finally it discusses the kinds of data which may be encountered. The purpose of this is to provide context for the work which is presented in this thesis and to give background details to aid in the understanding of future discussion.

Chapter three presents the methods which were used for extracting data types from a file and generating appropriate variable types. It also discusses the methods used in the generation of functions and type class instantiations to work with large tuples.

Chapter four presents results of testing the type generation on various data-sets in order to verify that types are generated correctly.

Chapter five talks about the work which was done in the project and how it solved the problem. Possible future work which could be done on the project is presented.

Chapter 2

Literature Review

The purpose of the literature review component of this thesis is to give context to the work which is presented, as well as to provide background information so that later discussion of the methods used can be understood, and to define terms.

The literature review is split up into five sections. The first presents machine learning as the application domain and discusses how Inductive Programming is the motivation for the project. The second discusses the Haskell programming language, which was the implementation language of the project. The third provides a brief explanation of meta programming. The fourth is a discussion of Template Haskell, which was the Haskell meta programming extension used for this project. The fifth gives an overview of the different types of data there are, and discusses what can go wrong with data.

2.1 Machine Learning

Although the application domain of this project is machine learning, it is not a machine learning project in its own right. This section of the thesis will give an overview of various systems which could be used for creating statistical models for machine learning and the problems with these systems.

2.1.1 Existing Statistical Packages

Various free and commercial statistical packages already exist. Among the more well known of these are:

- Weka – A free Java package of machine learning algorithms for data mining. Witten and Frank (2005) contains information about the package as well as various other topics relevant to data analysis.
- R – A free general purpose statistical platform. Venables et al. (2006) gives an overview of R.
- S-Plus – As with R, S-Plus is a general purpose statistical platform. However, S-Plus is commercial. Crawley (2002) documents the use of various statistical methods within the S-Plus environment.

Such statistical packages tend to have their own scripting languages which are separate from the main implementation language – see the R Language Definition (The R Development Core Team 2006) for a detailed overview of R’s language, for example – which can be used to write scripts to run within the main program. This separation between implementation language and scripting language makes some tasks – such as defining data structures and variable types – difficult as there is often no way of “getting at” the underlying type system.

There are also problems due to the fact that the scripting languages of statistical packages are interpreted rather than compiled. This means that they lack compile time type-checking and thus any potential type errors may not be detected until they have occurred at run time. Additionally there is the fact that interpreted languages generally run far more slowly than compiled languages, which may become evident when a large data-set is being processed. Finally, the languages being interpreted limits the distribution of the program, as anyone wishing to run it must have the appropriate interpreter, which means they need to possess the statistical package.

2.1.2 Inductive Programming

Inductive Programming (IP) is a machine learning paradigm which aims to generalise as much as possible of machine learning. Allison (2006) illustrates how IP can be used to tailor new solutions through a case study where

Bayesian networks are applied to a data-set of missing people. The study shows four distinct layers of components interoperating successfully.

IP is implemented using the purely functional programming language Haskell (Peyton Jones 2003). Haskell is a good choice of language for this domain. In addition to being strongly typed and compiled it has a powerful type system and a wide range of in-built type classes, some of which will be seen in section 2.2.3. Despite Haskell's suitability for the application domain there were problems encountered with the monadic I/O system. These problems will be further discussed in section 3.1.1.

Allison (2003, 2005) defined several simple, general purpose statistical models. More recently it has been shown how they can be quickly adapted to suit new problems (Allison 2006) (as previously mentioned they were adapted for Bayesian networks). This demonstrates a degree of generalisation, but there could still be more. Each time a new data-set is being processed new variable types must be defined and special functions written for handling these variable types. It would be extremely useful if the amount of code needed for these definitions was minimised.

Although writing the code for these variables and associated functions is fairly simple, it is a mundane and error prone process as the code is fairly repetitive. This repetition, however, makes the creation of the code suitable for automation. This is the main aim of the project: to generalise as much as possible of the description of data through the automated creation of appropriate variable types and the instantiation of these types in the appropriate type classes, with functions generated to assist in handling variables of these types.

2.2 Haskell

As was previously mentioned, the implementation language for IP is Haskell. Given that the aim of this project is to augment IP the implementation language for the project is also Haskell. This section gives an overview of the important aspects of the language.

2.2.1 Functional Programming

Haskell's main distinguishing feature is that it is a purely functional language. Functional programming languages differ from imperative program-

ming languages such as C in that they are not based upon states. Functional programming lacks side effects, that is, no function can have any effect other than returning a value, and the process of creating this return value has no influence outside of the function. This lack of side effects leads to a property known as referential transparency, where any call to a function can be replaced by the return value of that function without affecting the result of the program. In an imperative language a called function may modify global values before returning anything (or it may not return anything at all), this is not possible in a functional language.

In addition to the property of referential transparency, in functional programming languages functions are first-class values. This means that functions can be passed as arguments to and returned by other functions. Functions which either have functions as their arguments or return functions are referred to as higher-order functions. An example of a higher order function would be a “createFilter” function which took a comparison function returning a Boolean as an argument and returned a function which could be applied to a list to remove all values for which the comparison function returned false when they were passed to it as a value. Figure 2.1 gives Haskell code for such a function.

```
createFilter p =
  let
    f [] = []
    f (x:xs) = if (p x) then (x:(f xs)) else (f xs)
  in
    f
```

‘[]’ stands for an empty list, ‘:’ is an infix cons operator.

Figure 2.1: An example higher order function.

2.2.2 Lazy Evaluation

In Haskell a lazy evaluation strategy is used. Lazy evaluation means that nothing is evaluated until it is requested. For example, it is possible to write a function which returns a list of all Fibonacci numbers (as in Figure 2.2).

This is an infinite list. If this function was used in a language such as C (which uses a strict evaluation strategy) it would result in an infinite loop. However, in Haskell it is possible to run this function with a request for the tenth Fibonacci number, and only up to the tenth list element is evaluated.

```
fib 0 = 0:(fib 1)
fib 1 = 1:(fib 2)
fib x = (head (fib (x - 1))) + (head (fib (x - 2))):(fib (x + 1))
```

‘head’ returns the first element of a list.

Figure 2.2: Infinite list of Fibonacci numbers.

Haskell also utilises a related strategy called “delayed evaluation” where expressions are not evaluated as soon as they are bound. If, as in the above example, the request for the tenth Fibonacci number is bound to variable x , the value is not evaluated as soon as the binding is done. Instead it is evaluated the first time x is referenced.

2.2.3 Polymorphism

Haskell provides general support for parametric polymorphism (Milner 1978). This means that it is possible to write functions which will operate upon more than one type. An example of a function of this type is the “head” function, which takes a list of elements of type a and returns a value of type a , but it is unspecified what type a is. The head function can be used on lists of integers, lists of characters, lists of lists, etc.

Haskell also provides support for “bounded” parametric polymorphism where the attributes of values dictate whether or not they can be passed to a function. This is done using type classes (Wadler and Blott 1989). A type class gives a list of operations which can be defined upon variable types. Instantiating a variable type within a type class involves defining these operations for that variable type. Figure 2.3 gives Haskell code for the creation of a new variable type “AsciiNum” and the instantiation of it in the Enum type class. Type classes provide bounded parametric polymorphism as some functions will only take values whose types are members of a given type class. An example of this is a function to add two numbers together.

It will only take values whose types are members of the Num type class, so neither of the members may be a character as it is not a member of the Num type class. However, it will take different types as long as they are members of Num, for example it will take either two integers or two floats.

```
data AsciiNum = One | Two | Three
```

```
instance Enum AsciiNum where
  toEnum p = case p of
    1 -> One
    2 -> Two
    3 -> Three
  fromEnum p = case p of
    One -> 1
    Two -> 2
    Three -> 3
```

The Enum type class defines operations for converting to and from numerical representations. It is a standard Haskell type class.

Figure 2.3: Instantiating a new variable type in the Enum type class.

As previously mentioned one of the reasons why Haskell is suitable for the application domain of this project is its large number of pre-existing type classes. The following existing type classes are of interest for this project:

- Read – Operations for creating members of the type class from representative strings.
- Show – Operations for converting member types to strings.
- Enum – Operations for converting member types to and from representative integers.
- Bounded – Specifies the minimum and maximum values of member types.

2.3 Meta Programming

In general terms meta programming refers to writing programs which create code. This code could be part (or all) of a program separate from the program doing the creation, or the program could modify part of itself. There are different kinds of meta programming; generative meta-programming where code is generated and then compiled as part of a program, or meta-programming in an interpreted language where code can be modified at run time (i.e. using Lisp's eval function to run a string as code). As Haskell is compiled, the meta programming technique used for the project is generative.

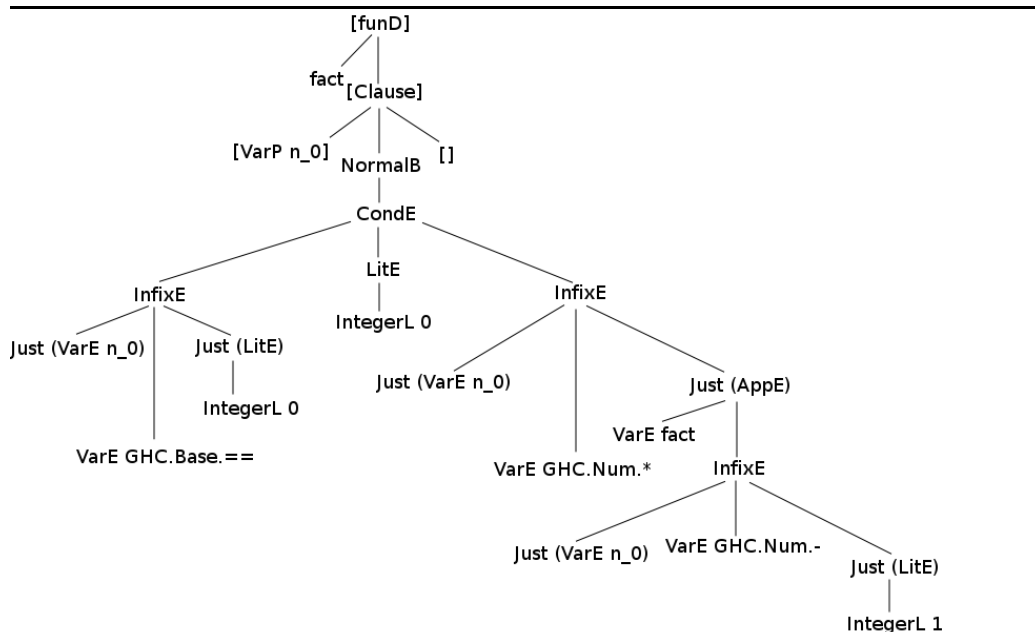
The language which a meta program generates code for is referred to as the “object-language” while the language doing the generation is referred to as the “meta-language” (Tarski 1956). If a language can be its own meta-language then it has a property known as “reflexivity”.

2.4 Template Haskell

Template Haskell (TH) (Sheard and Peyton Jones 2002) is a compile-time meta programming extension to Haskell. The initial paper which proposed TH stated that the idea was the “algorithmic construction of programs at compile-time”. It does this by generating abstract syntax trees (ASTs) at compile time (Figure 2.4 shows an example AST) and then “splicing” these ASTs into concrete syntax at the location where the code generation was started.

There are several advantages to using TH: standard Haskell variable types and functions are used to represent and handle ASTs (so TH gives Haskell reflexivity) which means that there is very little additional syntax required (the only additions required for this project were quasi-quotes and a splice operator, these will be discussed in section 2.4.2). In addition to this TH is type checked, which means type errors which may “slip through” simple macro expansion are avoided. Unfortunately as relatively new technology there are still bugs and incomplete features, but these problems were worked around as they were encountered.

This section of the literature review gives an overview of the usage of TH and past work which has been done on it and using it. This is to give the background required for later discussion of the work which was done using TH for this project. It also establishes that TH has not been previously used



An abstract syntax tree using variable types defined by Template Haskell. It is for a simple factorial function:
`fact n = if (n == 1) then 1 else n * (fact (n - 1))`

Figure 2.4: Example AST

for similar work to that which is presented in this thesis.

2.4.1 History

TH was initially proposed in 2002 (Sheard and Peyton Jones 2002) under the name “Template Meta-Haskell”, and was implemented in the Glasgow Haskell Compiler (GHC) (The GHC Team 2002). This initial paper gave example functions to demonstrate TH’s usage. Notable amongst these functions was `mkZip`, which defined a function for generating `zip` functions. The `zip` functions are a group of Haskell functions which convert lists of tuples into tuples of lists in a way which can be viewed as grouping columns (with each list representing the contents of a column) into rows (with each tuple representing a row). There is also a corresponding group of `unzip` functions which go the other way. Figure 2.5 gives an example of the operation of these

functions.

```
> zip [1,2] ["Cat", "Dog"]
[(1,"Cat"),(2,"Dog")]

> unzip [(1, "Cat"),(2, "Dog")]
([1,2],["Cat","Dog"])
```

Lines beginning with ‘>’ represent input to an interactive top level, and the following line is the result of this input.

Figure 2.5: Demonstration of zip and unzip

There are also corresponding functions to zip/unzip different numbers of lists (i.e. zip3). These functions are useful for the handling of data, as they can be used to drop columns which are not needed. However, often datasets contain a large number of columns, and the zip/unzip functions are only defined up to zip7/unzip7. For this reason the mkZip function defined by the initial TH paper would have been extremely useful. However, a year after the initial paper proposing TH there were changes made to TH (Sheard and Peyton Jones 2003) which made mkZip unusable. It has been rewritten for this project as well as a corresponding function for generating unzip functions, these are discussed in section 3.2.1.

As well as work done *on* TH there has been some previous work done *using* TH. A list of currently known papers is available from the Template Haskell website (Lynagh 2003b). The rest of this section summarises previous work done by others using TH.

The initial paper which showed applications for TH (Lynagh 2003a) gave a number of different projects it had been used for. These were: a curses system for a terminal based Tetris-like game written in Haskell (Hetris), a method for generalising type class instantiations for newly defined type classes (this may be of interest if future work is done on this project), an expansion on a formatted printf function introduced in the original TH paper, an unroller for fixed depth recursion (Fraskell) and a program which takes a functional description of an image or animation and outputs optimised Haskell code for it.

A paper was written at around the same time which went into further

detail and gave more examples of the use of Fraskell (Lynagh 2003c).

TH has been used for automated selection of appropriate skeletons at compile time (Hammond et al. 2003). Skeletons are generic program structures which capture common algorithmic forms, and are used to try to help add simplicity to programming parallel systems (Rabhi and Gorlatch 2003).

Seefried et al. (2004) discusses problems with the methods used in the implementation of embedded domain specific languages (EDSLs) and puts forward the idea of a language with compile-time meta programming being the ideal platform for implementing EDSLs. Haskell with TH is presented as an example of this.

Of these listed TH projects the only one with some similarity to this project was the generalised type class instantiations. However, this does not overlap with anything presented in this thesis. The thesis presents methods for generating type class instantiations for large tuples rather than simplifying the instantiation of types in newly defined type classes.

2.4.2 Syntax

As previously mentioned there were only two TH extensions to Haskell syntax which were required for this project. These were quasi-quotes and the splice operator. This section of the literature review gives an overview of these, as well as a brief introduction to syntax construction functions.

Quasi-quotes are syntactic sugar for converting from concrete syntax to ASTs (they can be used instead of the explicit syntax construction functions which are discussed later in this section). They take the form of square brackets with vertical bars: `[| |]`. The plain quasi-quotes return a TH representation of an expression, alternately there are declaration quasi-quotes `[d| |]` and pattern quasi-quotes `[p| |]`. An example of quasi-quote usage is:

```
[d| fact n = if (n == 1) then 1 else n * (fact (n - 1)) |]
```

This produces the AST seen in Figure 2.4.

The other addition to syntax used in this project was the splice operator. Splicing goes from ASTs to concrete syntax. The splice operator takes the form of the dollar sign: `$`. For example, splicing the above quasi-quoted factorial function declaration into code would have the same effect as having done the function declaration normally. Quasi-quotes can also contain splices

```

genFact :: Int -> ExpQ
genFact n = doGenFact (n - 1) [| n |]

doGenFact :: Int -> ExpQ -> ExpQ
doGenFact 0 current = current
doGenFact n current = doGenFact (n - 1) [| $(current * n) |]

```

‘`::`’ stands for ‘has type’ and ‘`->`’ for function type. This is a recursive function which passes down a growing syntax tree. For example, if `genFact` was called with the argument 3 then the first call to `doGenFact` would pass the integer 2 and the syntax tree for 3 (with type `ExpQ`). The next call would pass the integer 1 and the syntax tree for `3 * 2`. Finally the next call would pass the integer 0 and the syntax tree for `3 * 2 * 1`, and as this matches the pattern of the top definition of `doGenFact` the syntax tree `3 * 2 * 1` would be returned.

Figure 2.6: A TH function for generating code for factorials

(just as splices can contain quasi-quotes). An example of this can be seen in Figure 2.6.

In addition to these two new pieces of syntax there are functions which are referred to as “syntax construction functions”. These allow for the manual construction of ASTs. In some cases it is simply not possible to create the desired syntax tree using quasi-quotes and splices alone, so manual construction must be done. These syntax construction functions are used for the creation of the variable types which TH uses to represent ASTs. An example of the use of such a function is:

```
infixE (Just [| 1 |]) [| (+) |] (Just [| 2 |])
```

This is the equivalent of writing `[| 1 + 2 |]`. It returns the TH representation of an infix expression (in this case `1 + 3`). It would also be possible to create the TH representation of the literals and operator using syntax construction functions, but for the sake of simplicity these have been done with quasi-quotes. `Just` is a data constructor for the standard Haskell variable type `Maybe`. Variables of this type can be either `Just something` or `Nothing`. This means it’s possible to represent only part of the expression, so the syntax tree for `1 +` or possibly only `+` could be returned. The `Maybe` variable type found application in this project. This is expanded upon later.

2.4.3 Summary

This section of the report has established what TH is as well as showing that it has not been previously used for applications overly similar to the applications presented in this thesis. It also laid the groundwork with examples and explanation of syntax so that later discussion of how TH was applied to solve problems can be understood.

2.5 Data

As has been previously mentioned, the main aim of this project is to generalise the description of data through the automated creation of new variable types. This section gives some background into the way data can be looked at as well as some discussion on the way in which data was handled in this project.

2.5.1 Types of Data

There are only a small number of different forms data can take. Data can be missing (i.e. a field which contains nothing, or contains N/A). Data can be scalar (a field containing only one value) and scalar values can be discrete or continuous. Data can also be structured in some way, for example a list of scalar values or multivariate data (like a tuple).

Table 2.1 is a sample data-set which contains multiple kinds of data. The ‘name’ column is a scalar field which contains strings, the ‘age’ column is a scalar field which contains integers, the ‘gender’ column is a scalar field of categorical data (Male/Female), the ‘styles’ column contains lists of categorical data (Butterfly/Breaststroke/Backstroke/Freestyle) and the ‘best times’ column contains lists of floating point continuous data. Note that the age of Linette Cason is missing data.

Because there are relatively few kinds of data it is possible to make a good guess at the type of a column through inspection. It is also possible to make a guess at whether or not a column contains categorical data (as the ‘gender’ column in Table 2.1 does) if there is only a small number of values repeating over many rows. Being able to get types through inspection means that it is possible to look at a data-set and generate representative types as required.

Name	Age	Gender	Styles	Best Times
Cameron Buntin	21	Male	[Butterfly, Breaststroke]	[23.53, 27.92]
Amanda Haynes	23	Female	[Backstroke, Breaststroke]	[28.32, 31.04]
Linette Cason		Female	[Freestyle, Backstroke]	[24.97, 27.21]
Dave Sybilla	25	Male	[Freestyle, Backstroke]	[21.52, 23.88]
Michael Sullivan	29	Male	[Butterfly, Freestyle]	[21.63, 23.42]

Table 2.1: Sample data – swimmers

2.5.2 Bad/Missing Data

It is possible that – for various reasons – data may be badly formatted or missing. This may be as a result of typographical errors (“Nale” being entered instead of “Male” for example) or the data may not have been available for entry.

There are some errors with badly entered data which are recoverable. For example, if the only difference between two entries in a column is that the first letter of one is capitalised while the first letter of the other is not then in many cases it’s a reasonable assumption that they mean the same thing and have just been entered differently. This can be recovered by making the first letter uppercase when each entry is read in.

However, in a situation like “Nale” being entered instead of “Male” there is no way to tell that these actually mean the same thing (and even if there was, it would be difficult to tell which one was correct). In situations like this the data would need to be fixed by hand before types were generated.

Situations where data is missing are simpler to handle. The standard Haskell `Maybe` variable type was introduced in section 2.4.2 and it was mentioned that variables of this type can either be `Just something` or `Nothing`. This can be used to handle cases where data has not been entered, for example, in the case of the age column in Table 2.1 the data would be represented as:

```
Just 21
Just 23
Nothing
Just 25
Just 29
```

A potential pitfall in the handling of missing data is the fact that different representations can be used. Some fields may not be simply left empty, but

may contain entries such as “N/A”, a dash, a question mark, or other things specific to the given data-set. To get around this problem it must be easy to specify in code what missing data may look like.

2.5.3 Summary

This section has given an overview of the types of data which exist, and some details on how types can be extracted through inspection. It has also given examples of the things which can go wrong with data, and suggested some solutions to these problems. This has been done to give background for the later discussion on the generation of new types.

Chapter 3

Methods

The goal of this project has been discussed and some background has been given into the mechanisms used to reach this goal. This chapter of the thesis discusses of the methods used to implement all the parts of the project.

3.1 Type Generation

A major part of the project was the generation of appropriate types for representing data. This section goes into the methods used to do this and the problems encountered with different methods tried.

3.1.1 Compile Time I/O

One of the major features implemented was the ability to specify the name of a data-file at compile time. Types are generated from the contents of this file. This means that it is not necessary to “hard code” the file name. This was particularly useful in testing as often many slightly different files were checked. The ability to specify which file to use during compilation expedited the process significantly.

Ideally it would have been possible to simply have a function which took the name of a file, read in each row and used the type generation functions to create the appropriate types, and then returned these types as a list of TH representations of declarations. However, due to constraints with Haskell’s monadic I/O system (monads are the system used to allow Haskell to perform state based computations when required) this was not possible, as strings can

not simply be extracted from values which use the IO constructor. This is because a function with the type signature

```
IO String -> String
```

cannot be used without destroying referential transparency.

Fortunately TH provides a function `runIO` which allows for an I/O function to be run in the `Q` monad which TH is based upon. This means it is possible to create and return the type representations with an IO constructor, use the `runIO` function to specify that the I/O should be run in the `Q` monad, and then the value is lifted to an AST, which is then spliced.

Initially the entire contents of the data file were spliced into the code using this method. However, this was problematic due to the time taken for the TH `lift` function to run on long strings. In addition to this, for large data files splicing in all the text would result in extremely large executables. To solve this problem the function was changed to splice a representation of the data-types found in the file. This representation took the form of a list of tuples in which the first field was the type of the corresponding column and the second field was a list of constructors if the column was a new type. For example, the type representation for the data-set shown in Table 2.1 can be seen in figure 3.1. Note that `T0` and `T1` are the names given to new types,

```
[("String", []), ("Int", []), ("T0", ["Female", "Male"]),  
 (" [T1]", ["Backstroke", "Breaststroke", "Butterfly", "Freestyle"] ,  
 (" [Float]", [])]
```

Figure 3.1: Example type representation list.

and all new types follow this trend. After this list is spliced into the code it is used for the actual generation of new types. The code for the function which generates this list of type representations can be seen in Figure 3.2.

3.1.2 Determining Types

Before new types can be generated it is essential to determine what the types are. The “base type” (string, integer, float) must be extracted from a column, and it must be determined whether or not the column contains

```

generatedType = $(
  let
    types = putStr "Enter file name: " >> getLine
           >>= readFile >>= \fc ->
           return (getTypes fc)
  in
    lift =<< runIO types)

```

‘>>’ and ‘>>=’ are sequencing operations, ‘>>’ for when the result of the first function is uninteresting and ‘>>=’ for when the result of the first function should be passed along. ‘=<<’ is the equivalent of ‘>>=’ but with the arguments ordered the other way around. ‘\fc ->’ is a lambda abstraction, an anonymous function with the argument ‘fc’ (the result of `readFile`). ‘`return`’ defines the result of the sequence of operations. ‘`getTypes`’ is the function which generates the type representations.

Figure 3.2: The type representation generation function

lists, whether it contains new categorical data for which new types must be defined, or whether or not it simply contains base type scalars. This process also involves correcting any badly formatted data if required. This section gives an overview of the methods used in determining column types and producing type representation lists like what is shown in Figure 3.1.

Base Types

The first step in determining the type of a column is the extraction of the base type. This is done using a simple “brute force” approach where each entry in the column is used as a parameter to a function which returns the string “Int” if all of the characters in the entry are digits, “Float” if all of the characters are digits with the exception of a decimal place or “String” if there are characters aside from digits and decimal places. The current state of “knowledge” for a column is passed to the base type finding function each time it is run, so if the value “3.1” occurs somewhere in a column and the value “3” occurs later it will not revert to thinking the column only contains integers. If the column contains lists (later in this section it is discussed how lists are identified and handled) then the base type of each element of each

list is found.

Categorical Data

After finding all of the base types it is checked whether or not the data is categorical. This is done on all of the columns with a base type of “string”. The categorical data finder checks the number of unique entries in a column (i.e. in a row containing {Dog, Cat, Cat, Mouse, Dog} the unique entries are {Dog, Cat, Mouse}) against the number of rows in the data-set. If it is determined that the number of unique entries is small relative to the number of rows (trial-and-error determined that the number of rows divided by the number of unique entries being greater than or equal to five is a good measurement) then the column is taken as containing categorical data. A new name for the variable type is created using the standard T0, T1, etc. and the list of unique fields is used as a list of constructors.

If a field in a column is found to contain data which has been determined from an earlier column to be categorical data then the column is assumed to be of the type which contains that category. Any new entries found in that column are added to the list of type constructors.

There are also various name corrections which need to be done for categories. Every element in the list of unique fields must be a legal data constructor. This means that when categories are read in their name must be capitalised, spaces removed and invalid characters replaced with textual representations (for example, ‘\$’ is replaced with “DollarSign”).

Format Specifiers

Because different data-sets have different formats it is necessary to provide a mechanism for easily switching between formats. For this project this was done using “format specifiers”, variables containing values representing special symbols in the data-set. The available specifiers are:

- Record Separator – The symbol which separates records (rows) of data. This will generally be “\n”, as records will be split between lines.
- Field Separator – The symbol which separates fields of data. It is “,” for comma separated values, for example.
- List Open & List Close – The symbols which indicate the opening and closing of a list.

- Missing Values – A list of symbols which represent missing values in the list. This could be “N/A” or “#Error”, for example. Fields which contain nothing are automatically assumed to be missing data.

Figure 3.3 gives example format specifiers for the swimming data.

```

Cameron Buntten,21,Male,[Butterfly,Breaststroke],[23.53,27.92]
Amanda Haynes,23,Female,[Backstroke,Breaststroke],[28.32,31.04]
Linette Cason,N/A,Female,[Freestyle,Backstroke],[24.97,27.21]
Dave Sybilla,25,Male,[Freestyle,Backstroke],[21.52,23.88]
Michael Sullivan,29,Male,[Butterfly,Freestyle],[21.63,23.42]

```

The swimming data from Table 2.1 in CSV format. The record separator is ‘\n’, the field separator is ‘,’, the list open character is ‘[’ and the list close character is ‘]’ and the missing value list contains ‘N/A’.

Figure 3.3: Swimming data in CSV format with example format specifiers

Lists

Although lists have been mentioned, the mechanism for dealing with lists has yet to be specified. List handling is achieved by expanding lists so that each element is treated as a separate part of the column. This means that if a data-set contains 30 rows, but one of the columns contains lists, and there are 90 list entries in total in that column then that column will be treated as though there are 90 rows for base type finding and categorical determination.

When the type representation list is generated each column is checked to see whether it starts with the list open format specifier. If it does then square brackets are appended to the start and end of the type representation for that column to indicate that that column contains lists.

3.1.3 Generating Types

After the type representation list has been created the actual generation of the types can be done. This process is started by splicing the value of the createTypes function into the main module. Figure 3.4 gives the code for this

function. This section explains how each part of the `createTypes` function works.

```
createTypes =
  let
    dtList = filter ((/=) [] . snd) generatedType
    newDataType x = makeType (fst x) (snd x)
    newDataTypes = map newDataType dtList
    newTypeSyn = makeTypeSyn "MainType" (map fst generatedType)
  in
    sequenceQ (newDataTypes++[newTypeSyn])
```

'`sequenceQ`' is a TH function with the type signature `sequenceQ :: [Q a] -> Q [a]`. In this case it is used to convert a list of quoted declarations to a quoted list of declarations which can be used in a splice.

Figure 3.4: The `createTypes` function

The first action taken is the extraction of new types and their type constructors from the type representation list. All of these new types and type constructors are then passed to a function “`makeType`” which generates the AST for a declaration for the new type. These newly generated types are also instantiated in the `Read`, `Show`, `Eq`, `Bounded` and `Enum` type classes. Passing the `makeType` function the arguments:

```
"[T1]" ["Backstroke", "Breaststroke", "Butterfly", "Freestyle"]
```

will return the AST for the type declaration:

```
data T1 = Backstroke | Breaststroke | Butterfly | Freestyle
  deriving (Read, Show, Eq, Bounded, Enum)
```

Note that the type representation being for a list has no impact on the data type which is defined.

After the creation of new data type declarations the declaration for the type synonym “`MainType`” – a tuple representation of all the newly created types and base types – is generated. Each tuple field is also a part of the data-type “`Maybe`” in order to add support for missing data. This generation is done by passing the first field of each member of the type representation list to the `newTypeSyn` function. Passing the `newTypeSyn` the list:

```
["String","Int","T0","[T1]","[Float]"]
```

and the string “MainType” will return the AST for the type synonym declaration:

```
type MainType = (Maybe String, Maybe Int, Maybe T0, Maybe [T1],  
                Maybe [Float])
```

The result of the variable type declarations and type synonym declaration are then returned in a way which can be directly spliced into code.

3.2 Utilities

After the creation of the main type synonym new functions and type class declarations must be created to handle tuples of the given size. All of these functions and type classes already exist, but they are only defined for tuples up to a certain size. This section presents newly created utilities which can generate functions and instantiations for any sized tuples¹.

3.2.1 zipN/unzipN

As previously mentioned the `zip` and `unzip` functions are standard Haskell functions which can be used for converting lists of tuples to tuples of lists and vice-versa. Figure 2.5 shows how the functions work. These functions are useful in data processing as they can be used to drop unnecessary fields. However, they are only defined for up to a given number of fields (seven) and often data-sets will have more columns than this. For this reason functions to automatically generate arbitrarily sized `zip` and `unzip` functions are required.

A `mkZip` function for generating `zip` functions was defined in the initial TH paper (Sheard and Peyton Jones 2002), but due to changes to the syntax construction functions it does not work with TH version 2. In this project the function was updated for TH version 2 (with the name `zipN`) and a corresponding `unzipN` function was written. These are discussed in this section.

¹GHC actually defines a maximum tuple size of 62.

Zip

The generation of `zip` functions is performed by the `zipN` function. An example of code generated by `zipN` is shown in Figure 3.5. Different sized `zip` functions are created by changing the size of certain areas of the code being generated. For example, if a `zip5` function were being generated then there would be five arguments to the lambda abstraction and the tuples used in the case statement's pattern matching would be five elements large.

```
nZip3 =
  let
    newZip_0 =
      \y1 y2 y3 ->
        case (y1, y2, y3) of
          ((x1:xs1), (x2:xs2), (x3:xs3)) ->
            (x1, x2, x3):(newZip xs1 xs2 xs3)
          (_, _, _) -> []
  in
    newZip_0
```

The result of `nZip3 = $(zipN 3)`. The function name `newZip_0` changes dependant on how many other functions have been previously generated. This is a result of the way TH names values.

Figure 3.5: Example zip function

The code generated by `zipN` can be used in two ways. The first (and most efficient) way is to bind the code to a name. Writing

```
nZip3 = $(zipN 3)
```

would result in `nZip3` being a higher-order function which returned the function for zipping three elements. Using this method `nZip3` could be used in exactly the same way as the in-built `zip3` function. Alternately the splice could be done every time the function was needed, for example

```
$(zipN 3) [1,2] [3,4] [5,6]
```

would give the same results as having used `zip3`. This approach is inefficient however, as the AST generation and splice must be done each time the function is required.

Unzip

In addition to the `zipN` function there is also a `unzipN` function. The code generation for `unzip` functions works in much the same way as the code generation for `zip` functions in that there is a general function template and the different sized functions are created by modifying the number of elements in certain sections of the function. Figure 3.6 shows an example of the code generated by the `unzipN` function.

```
nUnzip3 =
  let
    newUnzip_0 =
      \lst ->
        case lst of
          ((x1, x2, x3):ts) ->
            let (xs1, xs2, xs3) = newUnzip_0 ts
            in (x1:xs1, x2:xs2, x3:xs3)
          [] -> ([], [], [])
  in
    newUnzip_0
```

The result of `nUnzip3 = $(unzipN 3)`. As with `zipN` the function name `newUnzip_0` changes dependant on how many other functions have been generated.

Figure 3.6: Example unzip function

As with the code generated by `zipN` the code from `unzipN` can be bound to a name or spliced every time it is required.

3.2.2 Type Classes

As well as the creation of the `zipN/unzipN` functions, a number of functions were written to automate the instantiation of large tuples in certain type classes. This is possible because – as long as the types of the tuple elements are members of the type class – the tuple instantiation in the type class is fairly general. This section discusses the type classes for which instantiation generation functions were written.

Show

The `Show` type class defines a method for converting member types to strings. For example:

```
> show 3.53
"3.53"
```

Only up to five-tuples are instantiated in the `Show` type class by Haskell's standard prelude. However, data-sets are often larger than five columns wide, and it is necessary for some of the records to be shown. For this reason large tuples need to be instantiated in `Show`, so it is one of the type classes for which an instantiation generation function was written.

As with the `zip` and `unzip` functions, `Show` instantiations take on a general form which can be easily modified depending on the size of the tuple. Figure 3.7 gives an example of an instantiation by showing how a three-tuple is instantiated in `Show`. This can be adapted to different sized tuples simply by changing the size of the tuples in the patterns and adding a new

```
shows bx . showChar ',' .
```

line for each tuple element between the start and end line of the definition of `showsPrec`.

```
instance (Show a1, Show a2, Show a3) => Show (a1, a2, a3) where
  showsPrec p (b1, b2, b3) =
    showChar '(' . shows b1 . showChar ',' .
    shows b2 . showChar ',' .
    shows b3 . showChar ')'
```

`.'` is a function composition operator, `shows` takes a string and returns a function to append a string to it. Using function composition with the `shows` function allows constant-time concatenation.

Figure 3.7: Show instantiation for a three-tuple

Read

The opposite of the **Show** type class is the **Read** type class, which defines operations for converting from a string to a member type. For example:

```
> read "43.64" :: Float
43.64
```

Note that unlike with **Show**, with **Read** the type of the result must be specified so that it knows which function definition to use.

As with **Show** there are only **Read** instantiations for up to five-tuples. As the `read` function is important for getting the contents of a data file into a useful form, a function for tuple instantiation in the **Read** type class was written.

Although more complex, **Read** instantiations for tuples also have a general form which can be modified depending on the size of the tuple. An example **Read** instantiation for a three-tuple can be seen in Figure 3.8. This can be adapted for different sized tuples by changing the size of the tuples in the patterns and adding new bindings for reading each element (reading the actual element, and reading the following comma).

```
instance (Read a1, Read a2, Read a3) => Read (a1, a2, a3) where
  readsPrec p = readParen False
    (\r1 -> [((b1, b2, b3), r8) | ("(", r2) <- lex r1
                                (b1, r3) <- reads r2
                                (",", r4) <- lex r3
                                (b2, r5) <- reads r4
                                (",", r6) <- lex r5
                                (b3, r7) <- reads r6
                                (")", r8) <- lex r7 ] )
```

'<-' binds the result of the right hand side to the left hand side. The reading is progressive, so first the open bracket is read, then the first element, then a comma, then the second element, and so forth.

Figure 3.8: Read instantiation for a three tuple

Enum

The `Enum` type class defines operations for converting member types to and from numerical representations. This is achieved with the `toEnum` function (which goes from numerical representation to type) and the `fromEnum` function (which goes from type to numerical representation). For example:

```
> toEnum 112 :: Char
'p'

> fromEnum 'p'
112
```

There are no tuples instantiated in `Enum`. This is presumably due to the fact that the return type of `fromEnum` is `Int`, which has an upper bound. Enumerating tuples of `Ints` (or other large members of `Enum`) would result in overflow. However, there are cases when it is desirable to enumerate tuples of relatively small members of `Enum`. For this reason a function has been written to generate tuple instantiations in `Enum`.

The method for converting to and from tuples to numerical representations was based on a method for converting between different base number systems, with each field being treated as a digit. For this method of conversion to work each element of the tuple must be a member of the `Bounded` type class as well as `Enum` so that the base (`maxBound + 1`) of each “digit” can be found.

An example of this system can be given with the following newly defined types:

```
data T0 = T00 | T01 | T02
data T1 = T10 | T11 | T12 | T13 | T14
data T2 = T20 | T21 | T22 | T23
```

“Digits” represented by `T0` are base three, `T1` is base five and `T2` is base four. The number 43 can be converted to a tuple of type `(T0, T1, T2)` in the following way (note that the fields are named `(a, b, c)`):

$$\begin{array}{l|l} 43 / (\text{base } T2 = 4) = 10 & c = (\text{remainder} = 3) = T23 \\ 10 / (\text{base } T1 = 5) = 2 & b = (\text{remainder} = 0) = T10 \\ 5 / (\text{base } T0 = 3) = 1 & a = (\text{remainder} = 2) = T02 \end{array}$$

So the tuple `(T02, T10, T23)` is produced. This can then be converted

back to a number using the following method:

$$\begin{array}{l|l} (T02 = 2) & 2 \\ 2 * (\text{base } T1 = 5) + (T10 = 0) & 10 \\ 10 * (\text{base } T2 = 4) + (T23 = 3) & 43 \end{array}$$

This method for converting tuples to and from integer representations is reversible, and every possible tuple of a given type has a unique integer representation. Figure 3.9 gives the code for a type-class instantiation in Enum for three-tuples.

```
instance (Enum a1, Enum a2, Enum a3, Bounded a1, Bounded a2, Bounded a3) =>
  Enum (a1, a2, a3) where
  toEnum p =
    let
      p1 = ((mod p ((fromEnum (maxBound :: a3)) + 1))
            (div p ((fromEnum (maxBound :: a3)) + 1)))
      p2 = ((mod (snd p1) ((fromEnum (maxBound :: a2)) + 1)),
            (div (snd p1) ((fromEnum (maxBound :: a2)) + 1)))
      p3 = ((mod (snd p2) ((fromEnum (maxBound :: a1)) + 1)),
            (div (snd p2) ((fromEnum (maxBound :: a1)) + 1)))
    in
      (toEnum (fst p3), toEnum (fst p2), toEnum (fst p1))
  fromEnum (b1, b2, b3) =
    let
      v1 = fromEnum b1
      v2 = (v1 * ((fromEnum (maxBound :: a2)) + 1)) + (fromEnum b2)
      v3 = (v2 * ((fromEnum (maxBound :: a3)) + 1)) + (fromEnum b3)
    in
      v3
```

‘fst’ returns the first element of a two element tuple, ‘snd’ returns the second element.

Figure 3.9: Enum instantiation for a three-tuple

Bounded

The `Bounded` type class specifies the minimum bound and the maximum bound of its member types. For example, using the `T0` type defined for the discussion of `Enum`:

```
> minBound :: T0
T00
```

```
> maxBound :: T0
T02
```

The `minBound` of a tuple is the tuple of the `minBounds` of its element types, likewise for the `maxBound` of a tuple. For example:

```
> minBound :: (T0, T0, T0)
(T00, T00, T00)
```

```
> maxBound :: (T0, T0, T0)
(T02, T02, T02)
```

As with `Read` and `Show` only up to five-tuples are defined in `Bounded`, so a function was written to automate the instantiation of larger tuples. The general form of the instantiation can be very easily adapted for new tuple sizes by changing the size of the pattern and return tuples. Figure 3.10 shows an instantiation for a three-tuple in `Bounded`.

```
instance (Bounded a1, Bounded a2, Bounded a3) =>
  Bounded (a1, a2, a3) where
  maxBound = (maxBound :: a1, maxBound :: a2, maxBound :: a3)
  minBound = (minBound :: a1, minBound :: a2, minBound :: a3)
```

Figure 3.10: Bounded instantiation for a three-tuple

Chapter 4

Testing

Testing of the utility generation functions can be performed simply by observing that the generated utilities perform in the manner expected. As they use general forms if they work for several different sizes of tuple it is a reasonable assumption that they work for all. A more formal testing method is required for testing the generation of types, however.

Type generation was tested on several different data-sets (Smyth 2004) to ensure that the correct types were being created for the representation of each data-set. After the types were generated the generated type representation list was checked to verify the types were correct.

Some of the data-sets the checking was done with were slightly modified by hand, i.e. changing “1” to “Yes” where appropriate, and fixing bad formatting (malformed delimiters, etc). None of this modification changed the meaning of the data.

For each data-set there is a description of the format of the data and the meaning of each column and an example of the data, followed by a description of the results of generating types based on that data-set.

4.1 Trained Monkey Data

The trained monkey data-set (Rehabilitation R&D Evaluation Unit 1991) is a simple data-set which catalogues information on several helper monkeys for disabled people. The first column contains the monkey’s name, the second column contains the years the monkey has been working for handicapped people, and the third column contains the number of tasks the monkey is ca-

pable of performing. Columns are delimited by tabs, and there is no missing data or lists. Table 4.1 gives a sample of the trained monkey data.

Hellion	10.0	28
Freeway	8.0	24
SuSu	6.5	28

Table 4.1: Sample trained monkey data

After reading in this data and finding types, the type representation list looks like

```
[("String", []), ("Float", []), ("Int", [])]
```

Although the data set does not require any new types to be generated or contain any lists, each column contains a different base type. This example shows that base types are being found correctly.

4.2 Childhood Respiratory Disease Data

The child respiratory disease data-set (Smyth 2002) contains statistics on the forced expiratory volume (the amount of air expelled from the lungs with constant effort) of children between the ages of six and twenty-two. The first column is purely numerical ID numbers, the second column is age as an integer, the third column is continuous data representing the litres of air forcefully expelled from the lungs in a second, the fourth column is height to the nearest half inch, the fifth column contains gender (male/female), and the sixth column contains whether or not a parent is a smoker (non/current). The data is tab delimited and contains no missing data or lists. Table 4.2 gives a sample of the childhood respiratory disease data.

The type representation list generated from this data is

```
[("Int", []), ("Int", []), ("Float", []), ("Float", []),
 ("T0", ["Female", "Male"]) ("T1", ["Current", "Non"])]
```

It can be seen from this that base types were found correctly, as well as the appropriate types being generated for the representation of all categories in the two categorical columns.

351	9	1.708	57	Female	Non
1701	8	2.336	58	Female	Non
44601	12	2.304	66.5	Male	Current
49201	11	3.206	63.5	Male	Non
72552	10	3.038	65	Female	Current

Table 4.2: Sample childhood respiratory disease data

4.3 Cholesterol Level Data

The cholesterol level data-set (Ryan et al. 1985) contains details on the blood cholesterol data of patients after heart attacks. For the sake of this experiment only two columns were used, the first is a patient ID and the second is a list of blood cholesterol levels two days, four days and fourteen days after the heart attack (note that the fourteen day cholesterol data is missing in some cases). The data is tab delimited and contains no missing data (lists with the fourteenth day missing simply contain two elements). Lists are opened and closed by square brackets. Table 4.3 gives a sample of the cholesterol level data.

1	[270 218 156]
2	[236 234]
3	[210 214 242]
4	[142 116]
5	[280 200]

Table 4.3: Sample cholesterol level data

The type representation list which is generated from looking at this data is:

```
[("Int", []), ("[Int]", [])]
```

It can be seen that the base type of the first column was correctly found. The second column was also correctly recognised as a column of lists, and the base list type was correctly found.

4.4 Missing Person Data

Allison (2006) showed how Inductive Programming could be used with a case study of Bayesian Networks. The data-set used for this contained information on missing people (Koester 2001). There are several basic type fields in the data-set, but of particular interest are the fields for which new types are defined in the paper. These new types are:

```
data Tipe = Alzheimers | Child | Despondent | Hiker | Other |  
          Retarded | Psychotic  
data Topography = Mountains | Piedmont | Tidewater  
data Urban = Rural | Suburban | Urban
```

Tests were performed using the missing person data set to ensure the types generated correspond to the types manually defined. Generating types using this data set results in the following type representation list:

```
[("T0", ["Alzheimers", "Child", "Despondent", "Hiker", "Other",  
        "Psychotic", "Retarded"]), ("Int", []), ("T1", ["Black",  
        "White"]), ("T2", ["Female", "Male"]), ("T3", ["Mountains",  
        "Piedmont", "Tidewater"]), ("T4", ["Rural", "Suburban",  
        "Urban"]), ("Float", []), ("Float", []), ("T5", ["Dead",  
        "Hurt", "Well"]), ("T6", ["Find", "Invalid", "Suspended"]),  
("T7", ["Air", "Dog", "Ground", "Law", "Local"]), ("T8",  
["Brush", "Field", "Linear", "Water", "Woods"]), ("Float",  
[]), ("Float" [])]
```

It can be observed from this that the types T0, T3 and T4 correspond to the manually defined types. This shows that the generated types are (aside from in name) the same as the manually defined types, which demonstrates the type generation is serving its purpose.

Chapter 5

Conclusion

As mentioned in the introduction, one of the necessary tasks in the use of any data-set for machine learning is the creation of new variable types to describe the data. Given that IP aims to encapsulate and generalise as much as possible of machine learning, it is beneficial for the creation of these variable types – along with several functions and type class instantiations – to also be generalised.

This thesis showed that the experimental Haskell meta-programming extension Template Haskell can be used to generate the required variable types. This was supported by a simple but effective approach to finding out the kinds of data which a target file contains. It was shown that generating types in this manner produces types which can be considered “correct”.

In addition to the creation of new variable types methods were presented for generating `zip` and `unzip` functions of arbitrary sizes and generating instantiations for large tuples in the standard Haskell type classes `Read`, `Show`, `Enum` and `Bounded`.

5.1 Further Work

Despite the achievements of the project there is still future investigation and work which could be undertaken in a similar vein. This work includes:

- Further investigation into the suitability of alternative Haskell meta-programming strategies such as Haskell generics (Lämmel and Peyton Jones 2003, 2004, 2005) or Generic HASKELL (Clarke et al. 2002). An initial aim of this project was to investigate all possibilities in this

respect, but due to time constraints only a superficial investigation was done before Template Haskell was chosen.

- There are further type classes for which large tuple instantiation could be generated. The selection of type classes to generate instantiations for is as-needed, and this thesis simply presented the ones most commonly required.
- The instantiation of newly generated types in appropriate user-defined type classes may be of assistance. This could build on the work previously done by Lynagh (2003a).
- After type generation has been done there are data-sets for which there is still a considerable amount of work that needs to be done to get the data into a format where each record can be parsed. It would be extremely useful if there were general utilities for formatting the data for reading into the data-types generated by this project.

Bibliography

- Allison, L. (2003). Types and classes of machine learning and data mining. In *25th Australasian Computer Science Conference (ACSC2003)*.
- Allison, L. (2005). Models for machine learning and data mining in functional programming. *J. Functional Programming*, 15(1):15–32.
- Allison, L. (2006). A programming paradigm for machine learning, with a case study of bayesian networks. In *29th Australasian Computer Science Conference (ACSC)*.
- Clarke, D., Jeuring, J., and Lö, A. (2002). The Generic HVSHELL user’s guide. Technical report, Institute of Information and Computing Sciences, Utrecht University.
- Crawley, M. J. (2002). *Statistical Computing: An Introduction to Data Analysis Using S-Plus*. Wiley.
- Hammond, K., Loogen, R., and Berhold, J. (2003). Automatic skeletons in Template Haskell. In *Proceedings of the 2003 Workshop on High Level Parallel Programming, Paris, France*. Available from <http://www.haskell.org/th>.
- Koester, R. J. (2001). Virginia dataset on lost-person behaviour, Author’s website: <http://www.dbs-sar.com/>.
- Lämmel, R. and Peyton Jones, S. (2003). Scrap your boilerplate: A practical design pattern for generic programming. In *Proceedings of the 2003 ACM SIGPLAN International Workshop on Types in Language Design and Implementation*, pages 26 – 37.

- Lämmel, R. and Peyton Jones, S. (2004). Scrap more boilerplate: Reflection, zips, and generalised casts. In *Proceedings of the Ninth ACM SIGPLAN International Conference on Functional Programming*, pages 244 – 255.
- Lämmel, R. and Peyton Jones, S. (2005). Scrap your boilerplate with class: extensible generic functions. In *Proceedings of the Tenth ACM SIGPLAN International Conference on Function Programming*, pages 204 – 215.
- Lynagh, I. (2003a). Template Haskell: A report from the field. Available from <http://www.haskell.org/th>.
- Lynagh, I. (2003b). Template Haskell website. <http://www.haskell.org/th/>.
- Lynagh, I. (2003c). Unrolling and simplifying expressions with Template Haskell. Available from <http://www.haskell.org/th>.
- Milner, R. (1978). A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375.
- Peyton Jones, S., editor (2003). *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press.
- Rabhi, F. and Gorlatch, S., editors (2003). *Patterns and Skeletons for Parallel and Distributed Computing*. Springer.
- Rehabilitation R&D Evaluation Unit (1991). An evaluation of capuchin monkeys trained to help severely disabled individuals. *Journal of Rehabilitation Research and Development*, 28(2):91 – 96.
- Ryan, B. F., Joiner, B. L., and Ryan, T. A. (1985). *Minitab handbook*. Duxberry Press. Table 1.1.
- Seefried, S., Chakravarty, M., and Keller, G. (2004). Optimising embedded DSLs using Template Haskell. Available from <http://www.haskell.org/th>.
- Sheard, T. and Peyton Jones, S. (2002). Template meta-programming for Haskell. In *Proceedings of the 2002 ACM SIGPLAN Workshop on Haskell*, pages 1 – 16.
- Sheard, T. and Peyton Jones, S. (2003). Notes on Template Haskell version 2, Available from <http://www.haskell.org/ghc/>.

- Smyth, G. (2002). Childhood respiratory disease data, Retrieved 5 Nov 2006, from <http://www.statsci.org/data/general/fev.html>. Data comes from the Childhood Respiratory Disease Study in 1980 in East Boston, Massachusetts.
- Smyth, G. (2004). Australasian data and story library, Retrived 5 Nov 2006, from <http://www.statsci.org/data/index.html>.
- Tarski, A. (1956). The concept of truth in formalized languages. In *Logic, Semantics, Mathematics*. Oxford: Clarendon Press.
- The GHC Team (2002). The glorious Glasgow Haskell compilation system user's guide, Version 6.4.2, Available from <http://www.haskell.org/ghc>.
- The R Development Core Team (2006). R language definition, Version 2.3.0.
- Venables, W. N., Smith, D., and The R Development Core Team (2006). An introduction to R, Version 2.3.0.
- Wadler, P. and Blott, S. (1989). How to make ad-hoc polymorphism less ad-hoc. In *Proc. 16th ACM Symposium on Principles of Programming Languages*. ACM Press.
- Witten, I. H. and Frank, E. (2005). *Data Mining: Practical Machine Learning Tools and Techniques*. Morgan Kaufmann, second edition.