Federico Pizzuti

Protocol Buffers for Standard ML

Computer Science Tripos – Part II

Jesus College

May 14, 2015

Proforma

Name:	**Federico Pizzuti**
College:	**Jesus College**
Project Title:	**Protocol Buffers for Standard ML**
Examination:	**Computer Science Tripos – Part II, June 2015**
Word Count:	**10217**
Project Originator:	Prof. Lawrence Paulson
Supervisor:	Prof. Lawrence Paulson

Original Aims of the Project

To provide a serialisation library for the Standard ML programming language implementing the Google's Protocol Buffers standard, related functionality to access deserialised messages in a idiomatic way and a code generator to limit the amount of boilerplate code the library user has to write. The project proposal also mentioned support for Remote Procedure Calls hooks in message definitions, and particular optimisations on the serialised data. Both these features were later abandoned during the project development, as the focus moved from the low level serialisation features towards ease of use and quality of code improvements.

Work Completed

A subset of the Protocol Buffers' serialisation standard has been implemented: this includes a parser for the proto2 language for schema definitions, and all the prescribed field types with the exception of Remote Procedure Calls and packed entries. A small functional lenses library is integrated in the project and provides the means for writing idiomatic message accessing functions. A dynamic code generator is capable of producing the code required to use functional lenses given a parsed schema definition, with the option to do so at runtime.

Special Difficulties

No special difficulty was encountered

Declaration

I, Federico Pizzuti of Jesus College, being a candidate for Part II of the Computer Science Tripos, hereby declare that this dissertation and the work described in it are my own work, unaided except as may be specified below, and that the dissertation does not contain material that has already been used to any substantial extent for a comparable purpose.

Signed

Date

Contents

Chapter 1

Introduction

This chapter explains the overall motivation behind the project. It then reviews some of the existing work in the field, highlighting the unique aspects behind Protocol Buffers for Standard ML

1.1 Motivation

The Standard ML language is a pearl hidden in the mud. It is a simple language defined by clear and concise rules relying on a functional paradigm to be able to express complex abstractions in few lines of code. Very efficient open source implementations are available, some of which are capable of exploiting the parallelism offered by many-core machines. In spite of these great qualities, the language is not used outside some academic circles, mostly because of the lack of key software libraries enabling common tasks.

In the last decade, the computing community has seen an immense growth in the presence of high speed-low cost Internet access. This has led to the creation of a large number of Internet services, and as such one of the key capabilities for a language to succeed is the availability of serialisation libraries, enabling programs to communicate with the aforementioned services.

Among the standards available, Google's Protocol Buffers stand out for efficiency and ease of use. It describes a binary serialisation format aimed at minimizing message size, and it is designed to map well to the primitive types present in most programming languages. Protocol Buffers implementations generally come with code generation tools that allow a native integration of the system within the facilities of the hosting language, allowing the programmer to interact with it in a consistent way with the rest of the application.It is a schema based protocol, defining a formal language to describe data definitions. It is used extensively at Google and similar environments where convenient communication between programs written in a multitude of languages is required.

A library that allows Standard ML code to leverage the power of Protocol Buffers would have the power of ameliorating the lack of support that plagues the Standard ML community. It could be used in conjunction with the standard library's networking functionality to access on-line services that communicate using Protocol Buffers, or it can

be used locally as a serialisation library to save structured data to disk, without having to write ad-hoc code every time.

The developed library has the potential to impact programming education: In institutions where Standard ML is taught as first language, it would empower students with a tool to help their understanding of serialization and network based communication

Personally, my interest in the project comes from the fact that it offers me the chance to write code pertinent to the areas of Computer Science I like the most: first, it is to be implemented in a functional language, and functional programming is my main interest.It allows me to deal with compilers related subjects such as parsing and code generation without forcing me to write a compiler for a full-blown language, or an incomplete toy project. And finally, it presents me with the challenge of writing low-level code in a functional language.

1.2 Related work

1.2.1 Analysis of similar projects

There are a great number of Protocol Buffers implementations, covering many different languages, and authored by different parties. Amongst the more commonly used ones are the Java and C++ protobuf libraries, both developed by Google. They rely on mapping Protocol Buffer's messages to classes and objects by providing a command line tool, protoc, which is capable of generating source code from schema definitions. In the Java implementation, the deserialisation code is not included in the corresponding message's class, but rather in a companion factory object. The approach of providing a code generation tool is not unique to OOP languages: the Haskell version, which targets a significantly different language than the previous ones, also follows a similar technique, generating record types in place of classes.

Another interesting library that stands out is the "Piqi project", targeting the OCaml and Erlang languages. It offers its own schema definition through the "piqi" language, and it is capable of employing a variety of serialisation standards, including JSON and Protocol Buffers. Interestingly, it supports functionality for generating .proto files from .piqi files, and vice versa. It too relies on ahead of time code generation, mapping groups of related piqi messages to an OCaml module.

1.2.2 Key differences

Protocol Buffers for Standard ML aims to be similar to the currently existing alternatives, with one key difference. Instead of relying on ahead of time code generation, it relies on a dynamic internal representation based on algebraic data types. This means that there is no need for the extra tool to be used during development, and that the system is capable of processing new or updated definitions on the fly. It will also be possible for users to inspect the structure of a newly parsed schema without resorting to language level reflection. Moreover, suitable abstraction layers will be provided to offer type safety when

interacting with the deserialised messages. A code generation tool can still be provided, to reduce the amount of boilerplate code needed to use the above mentioned abstractions.

1.3 Achievements

The serialisation library implements a subset of the Protocol Buffers standard. It supports most of the features specified in the developer's guide for the proto2 schema definition language, with the exception of remote procedure calls hooks on messages and support for a particular size optimisation know as "packed repeated fields", which allows the user to trade encoding away speed in order to reduce the serialised binary's size

In order to provide comfortable and idiomatic access to deserialised message instances, the project relies functional lenses. This is a technique used in most modern functional languages for record access, and it was found to be particularly suited to the problem at hand. However, usage of functional lenses generally requires the user to write some boilerplate code in order to start using the features. To minimize the problem, a code generator has been implemented, capable of mechanically producing the code required to use functional languages from message definitions.

Chapter 2

Preparation

This chapter begins with a quick explanation of the background information required by the reader in order to understand the project's design and implementation, followed by a summary of the preliminary research and preparation.

2.1 Protocol Buffers

2.1.1 What are they

Protocol Buffers is a standard for data serialisation, a set of multi-language libraries and associated tools developed at Google, first released in July 2008. It aims to support the exchange of structured data between programs written in different languages, while relieving the programmer from relying on custom serialisation code. In the standard use case, the author of the programs relying on Protocol Buffers to communicate would download, for each language employed, a specific runtime library and a code generation tool. They then proceed to write a set of ".proto" files, which describe the format of the data to be serialised in a formal language. This is done through the definition of a series of "messages". Each message is a named collection of fields, and each field is composed by a modifier (such as "required" or "optional"), a type(such as "int32", "string", or the name of a previously defined message), an identifying name and a unique numerical key. Then these .proto files are supplied to the language specific code generation tool, which will produce source files that implement the structured data type in the way that is idiomatic for the targeted language (as an example, the C implementation generates structs, the C++ one generate classes, the Javascript one code to construct suitable dictionaries...). Generally, together with a representation of the message, a pair of functions are provided to serialise and deserialise an instance of it from a bytestream (the exact specifics are, as with the rest, all language dependant).

The Protocol Buffers standard has been designed with the priority of minimizing the size of the encoded messages: this explains several of the design choices made, such as relying on a binary format (sacrificing human readability) and using particular length-efficient encodings for numerical values. As such, they excel when used as a medium for request/responses with remote services

2.1.2 Language and binary format

What follows here is a quick and minimal example of two protocol buffer messages, defined in a file. While this example is by no means sufficient to represent the schema definition language to its full extent, it will be sufficient to understand the rest of the dissertation.

```
1  //This is a comment
2  //Example.proto
3
4  message Address {
5
6  required int32 number = 1;//Create a required field
7  //containing a 32-bit integer called x, with key "1"
8  required string streetName = 2;
9  }
10
11 message Person {
12 required string firstName = 1;
13 required string lastName = 2;
14 //The next field is marked as optional:
15 //it may or may not be found in the de-serialisation
16 optional string middleName =3;
17 //This is an example of a message-typed field
18 required Address home = 4;
19 }
```

Before proceeding to list the various allowed types, is important to cover the binary format of a Protocol Buffer. From now own, I will use the name "proto message" to refer to the de-serialised and logical entity, and the name "wire message" to refer to the serialised instance of a proto message.

A wire message is a binary stream of (tag,value) pairs. Each of these pairs represents one of the fields described in the matching proto message. There is no constraint on the ordering of the pairs, nor there is an header that specifies the name of the message or other identifying metadata. As such, it is impossible to deserialise a wire message without prior knowledge of its definition.

Each value that appears in the wire message (included the key), is encoded in a specific way. The used formats are:

- Fixed-length encoding: the traditional encoding for a value: 2's complement for numerical types, ASCII for characters in strings. Wire messages are always little endian.
- Run-length encoding: the length of the value is prepended to the value itself.
- Variable length integer encoding.This variable length representation is derived from the fixed representation of the value: all leading zeros are removed, and the remaining bits are gathered in groups of 7 (leading padding zeros for the group containing the Most Significant Bit are added if necessary). Then, to all the groups is added one more bit: a 0 if this is the group with the LSB, a 1 in the other cases. Finally, these 8-bit groups are simply written down in order as a byte stream. To decode,

simply reverse the process: start reading bytes that begin with a 1,stop when a beginning 0 is met. Remove the leading bit from each byte, concatenate all the bits.

Each of the above encodings is identified by a numerical value (0 for variable length integers, 1 for fixed 64-bit values, 2 for length-delimited values and 5 for 32-bit values). The encoding identifier is then combined with the key, using the formula:

```
tag = key << 3 | type.
```

Finally, for length delimited values, a variable length integer indicating the length of the field's payload is written after the tag.

The possible field types are

- int32, int64, uint32, uint64 : Traditional integer values represented with variable length integer, so smaller values have a shorter encoding.
- bool: encoded as a varint, with true $\mapsto$ 1 and false $\mapsto$ 0 bit count only refers therefore to their full size while deserialised. The uintXs are to be interpreted as unsigned values only
- fixed32, fixed64: Standard integer values encoded within the respective fixed sizes
- sint32, sint64: Signed integer values, before being serialised they are remapped using a zigzag function: $0 \mapsto 0, 1 \mapsto 1, -1 \mapsto 2, 2 \mapsto 3, -2 \mapsto 4$.... Mapping common negative values to positive ones reduces the average size of the encoded varint: in a straight varint encoding, all negative numbers must be represented with the full number of bytes, as the sign bit is always 1 for them. Using the zigzag function allows therefore to reap the benefit of varint size reduction even with negative values present
- sfixed32, sfixed64: similar to the above, only that they are encoded as fixed size instead of using varints.
- float, double: IEEE floating point values. Both are encoded as fixed size values, 32-bit for float and 64-bit for double
- bytes, string: length-delimited values. Bytes simply indicates a sequence of bytes. String are ASCII based and therefore represented exactly the same as bytes.
- message: This is not a real type, in the sense that one would specify the actual name of the message. Sub messages are included simply by treating their encoded byte stream as a bytes field.

In addition to each type, each field may hold a modifier. This is drawn from a few possible alternatives:

- required: This is the default if no modifier is present.It indicates that the field must be present in an encoded message, and must be unique. If neither of these conditions is met, then the deserialiser has to fail.

- optional: A field must be present 0 or 1 times
- repeated: The field must be present at least 1 time. If indeed more than one copy of the same field is encountered, the last one seen is the visible one to the host language, and the others are carried silently along.

2.2 Functional Lenses

2.2.1 Motivation

One of the core objectives of this project was to provide a convenient and idiomatic way to interact with the deserialised messages. In a traditional imperative language, this would be quite straightforward: mapping messages to structures/objects would automatically provide the required functionality to access and update their fields.

However,this is not the case in a functional language like Standard ML. Moreover, writing some functionality to emulate imperative access and update is generally undesirable, as it would clash with the otherwise side-effect free style that is pervasive within the language. At first, this problem was addressed by providing two simple functions, get and set, which would have type like

```
1 val get : protoMessage * string -> 'a
2 val set : protoMessage * string * 'a -> protoMessage
```

Get simply looks up in the a proto message for a field given a name, and set looks for a field given a name and returns a new *protoMessage* with the field value updated. This seems fine at first, however, as soon as the code logic gets more involved, problems start to arise. The following example should illustrate the issue:

```
1  val name = get(person,"name")
2  //or, in imperative parlance, name = person.name
3  val federico = set(person,"name","federico")
4  //person.name = "federico";
5  (*Simple "0-th level" access works fine*)
6
7  val motherName = get(get(person,"mother"),"name")
8  //motherName = person.mother.name;
9  val federicosMother = set(get(federico,"mother"),"name","barbara")
10 //federico.mother.name = "barabara";
11 (*One level deep, the nesting starts to become quite complex*)
12
13 (*Lets try to append Mrs to my mother's name...*)
14 val federicosMotherMrs = set(get(federico,"mother"),"name",
15 "Mrs " ++ (get(get(federico,"mother"),"name")))
16 //federico.mother.name = "Mrs. " ++ federico.mother.name
17 (*Ok, writing this last line tired me out...and probably
18 there is some bug in it too!*)
```

It is obvious that this simple method just does not scale. After a more thorough observation, it is possible to notice the root cause of the problem: A whole new level of nesting

is required every time we need to read a field from a contained object, and this leads to intolerable amounts of boilerplate and repetition.

And while it is possible to ameliorate the situation somewhat by breaking up each line with let definitions, this does not reduce the overall amount of boilerplate code that needs to be written. To a functional programmer, the problem is that the functions get and set do not compose well, and a more flexible abstraction is needed.

2.2.2 Functional getters and setters

Functional lenses are the abstraction that fills our needs perfectly. Their curious name refers to the fact that a lens is an entity that can be used to "focus" on a sub-part of a greater structure. This gives us a method of easily traversing, inspecting and updating the contents of some complex structure in a general method that minimizes the amount of code required.

The first step that needs to be taken to derive them is to try to characterize the nature of the elements we are dealing with. The two basic blocks of the new abstraction are the above mentioned pair of functions: the getter and the setter. We can characterise them as two functions with types

```
1 get : 'a -> 'b
2 set : 'b * 'a -> 'a
```

This pair of functions is what is called a ('a,'b) lens, where 'a and 'b are the matching type variables for the functions. A lens therefore can be used to implement the concept of an accessible field, and it can be used by simply projecting out the get or set component and apply it to the particular instance we wish to access. Using this new approach, we then define two new functions, *getL* and *setL*, which take a lens and apply the relevant component to the supplied parameter (this allows us to "skip a step", since most of the time we'll want to apply lenses to actual values)

```
1 getL : ('a,'b) lens -> 'a -> 'b
2 setL : ('a,'b) lens -> 'b * 'a -> 'a
```

We now want to devise a way to combine together two lenses to get a new one. This is akin to "chaining dots" in traditional imperative field access, and it is exactly what is needed to resolve the nesting problem that was highlighted in the previous section. From the type of get, we can see that there is only one sensible way to compose lenses:

```
1 compose : ('a,'b) lens * ('b,'c) lens -> ('a,'c) lens
```

Such an operation can be described intuitively as: we know how to access a field of type 'b from an object of type 'a and we also know how to access a field 'c from an object of type 'b, therefore we know how to access a field of type 'c from an object of type 'a, passing trough the 'b component. A similar although a bit convoluted process allows us to compose the set members, completing the operation. If we were to replace compose with an infix symbolic operator, we could write code like

```
1 val motherName = getL(mother +> name) federico
2 val renamedMother = setL (mother +> name) (federico,"barbara")
```

The actual definition of compose will be presented in the implementation chapter, together with the presentation of a variety of useful function that will allow for convenient lenses usage by trying to recreate an imperative-like syntax, and with code that is capable to generate lenses on the fly for proto message instances given their definition.

2.2.3 Lens laws

Some readers might be wondering whether composing lenses as presented above always results in some new lens who follows the common sense "getter-setter" behaviour we wished to achieve: their type signatures seem to describe well the intended result, but this is not enough: Standard ML is a language that allows side effects to happen, so no guaranteed can be made. Moreover and even without mutation involved, it is easy to see how one could create some particular lenses instances whose behaviour is not what was expected. Consider for example a "faulty lens", whose setter works correctly, but the getter incorrectly always returns a constant value. Such faulty lens is still a lens by the definition given above, but its behaviour would be highly puzzling to the user, especially with respect to composition and chaining. In order to rule out these possibilities, there are three rules that every lens needs to obey, known as the lens laws:

```
1 getL l (setL l (value,x)) = value
2 setL l (getL l x ,x) = x
3 setL l (a,setL l (b,x)) = a
```

In prose, this means

1. If a field is set and then immediately read, the result is what was set
2. If a field is read and then set to the value that was read, is as if nothing changed
3. If a field is set twice in a row, it is as if only the second update happened

If a lens respects these laws, then it is safe to say to assume that it is well-behaved. Moreover, it can be proved that the result of composing lawful lenses is always lawful.

It must be stressed that the lens laws are not enforcible in the Standard ML language, and it must be care of the author of a particular lens to assure that these are respected.

2.3 Project preparation

2.3.1 Language and environment

In order to be able to implement the project, some research had to be done regarding the programming language to be used and the available libraries. My previous knowledge of Standard ML was good, as it has been taught in the first year here at Cambridge, but incomplete, since the complex and extensive module system has been avoided entirely. It was through learning those advanced language features that I could precisely work out the structure of the program as a whole, as discussed later in the design subsection.

In addition to the module system, significant amount of time has been spent in looking for libraries that could simplify common tasks, such as a Map structure to represent

the key-value pairs, a parser library that could simply the task of writing the parser for the .proto files and a "Dependency Manager" tool that could organize source code imports automatically (Similarly to C, in Standard ML to reference code in other source files one must include them in the current source. However, unlike C, Standard ML does not support conditional compilation. This means that, the most complex files, with many interconnected dependencies, by default import many copies of the referenced files). However, the amount of publicly available, documented, functioning libraries for Standard ML is almost zero, and for these tasks I ended up writing code myself. While this might seem unprofessional, it turned out to be far faster, and sufficient for the needs of the program, to reimplement those features on my own that trying to engage in "code archeology" trying to infer how to use this old undocumented libraries. The only case where a legitimate library was available is in the parser generator, SML-yacc. However, during the time of familiarisation with the language, I had written a simple parser combinator library, inspired by the Haskell library Parsec. Once again I have preferred to use my own code, and it was sufficient for the needs of the program, blending much better than the table-generating imperative code that SML yacc generates.

2.3.2 Research

The previous section has presented the specification of the Protocol Buffers standard and the theory behind functional lenses. The information on Protocol Buffers was drawn from Google's protocol buffer's development guide. In addition to the information presented previously, the development guide contains a chapter dedicated to the writers of protocol buffers implementations, covering topics such as what sort of malformed messages should a parser tolerate, or on how to choose which native types correspond to the proto types. In addition to this, I decided to take a look at the current open source implementations, particularly the Java and Haskell ones.

I have seen the different approaches that were used (such as Java relying extensively on inheritance, Haskell on algebraic data types) to form a better idea on the different ways to approach the overall problem. This also suggested a variety of ways to differ from the already existing projects, such as not relying on a code generation tools like the other implementations but instead handling everything dynamically within the runtime library, reducing the complexity of the user's work flow, while still relying on code generation to preserve the ease of use. Prof. Paulson was also very kind and forwarded me a short e-mail exchange he had with an engineer at Google, which was also advocating for a similar approach.

The research on the theory of lenses comes from my previous experience in writing software in Haskell and Scala. Both languages have a lenses library which simplifies greatly the amount of code required to access fields of records and algebraic data types, and I have used them with great satisfaction. Most of the materials comes from Edward Kmett's talks "Lenses,Folds, Traversals" and "Lenses: A functional imperative", which are available on YouTube, together with the documentation found on the library data-lenses for Haskell.

2.3.3 Requirements Analysis

The requirement analysis for this project was quite straightforward, as it has to match that of other .proto implementations: to be able to parse .proto files to generate message definitions, to use those definitions to serialise and deserialise protocol buffers into some Standard ML data type. In addition to this, the project must also offer functionality to easily interact with the objects in an idomatic way, through the use of functional lenses. Finally, as an additional feature, the library should include the ability to mechanically generate code when possible to reduce the boilerplate required by the user.

The requirements highlighted above are however still vague, and if they are to be fully completed, the project would quickly grow too big to be comfortably implemented in the time allotted. Therefore, each requirement has been better specified:

- The parser must be able to generate definitions from correct .proto sources, and refuse to parse malformed ones. However, it is not required to generate accurate, user friendly error messages.

- The serialiser and the deserialiser must correctly process well-formed messages, and have some basic level of error-tolerance. However they need to only implement a subset of the whole standard: in particular, support for remote procedure call stubs was not included, and deprecated features are not to be supported.

- The lens functionality has to provide type definition for lenses, together with a composition operation, and few other convenience functions. It does not support monadic compositions, as common with other lens libraries, nor other lens-like concepts such as prisms.

- The code generation functionality must be able to generate a Standard ML structure given a set of protocol buffers definitions. For each message a type is to be created, the serialisation and deserialisation functions, and for each field a lens. Each structure must be equipped with a matching signature, and the user can decided whether to use transparent or opaque matching

2.3.4 Design

In order to ensure a smooth implementation and avoid problems that could have jeopardized the success of this project, a clear division in modules was necessary. It was decided to map each feature to a module, and then eventually add other convenience modules to better group functionality. Then type signatures have been written for each module, in order to get a first glimpse at the structure of the code and to start defining inter-module contracts.

The main design decision was to structure the whole project as a small "technology stack": originally, the vision was to provide a top-module which would be the only one the user would ever interact with, keeping the internals of the library hidden. Instead, after the division in modules, it was clear that such an approach would have only limited the flexibility of the library. The final design resulted in the following "layers"

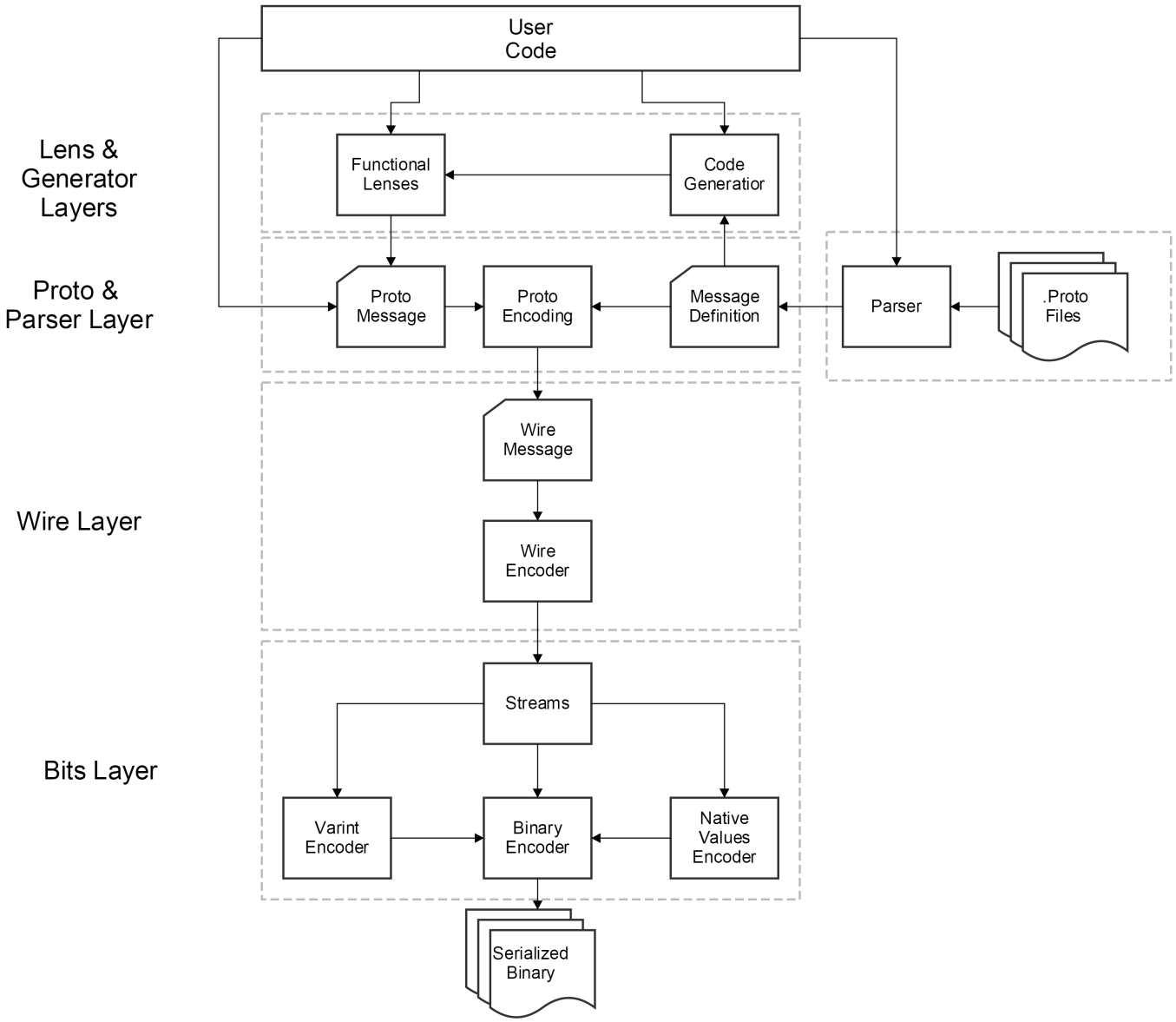

Figure 2.1: The layered design. Boxes denote components, arrows indicate usage

- Bits layer: functionality to read and write values from/to a byte stream in a variety of encodings, such as fixed, varint and length delimited. It also provides a stream abstraction, used to represent the target/source byestream
- Wire layer: contains an algebraic data type representation for a wire message. Using the functionality from Bits, it can be serialised/deserialised to/from a byte stream.
- Proto layer and Parser: provides an algebraic data type representation for proto messages, and for their definition. It includes the parser to generate the definition, functionality to non type-safe access to the internals of a proto message, and builds on the previous layer by providing functionality to convert between proto messages and wire messages.
- Lens layer: contains the code that implements lenses in a generalized way and

specific functions for manual lens generation on proto messages. In addition provides functions to automatically generate lenses for a given protocol message definition.

- Generator layer: This layer takes care of code generation. It offers functions that produce SML Structure and matching Signature from a set of proto messages definitions. For each message type an SML type is generated, and for each field lenses are generated. This is the top-most layer of the stack. It is possible to instruct the generator to produce opaque type signature, therefore hiding the implementation of proto messages behind new types

The idea behind this layering is that the user can freely choose the level of abstraction that is better suited for his purposes. For example, some might want to be given the power to perform low-level operations on proto messages, be able to write extensions to the serialiser, or simply wish to minimize the performance overhead that would otherwise be introduced by using more sophisticated abstractions. Others might wish to just operate at the lens layer, which will allow them to gain convenience and type safety without relying on code generation, and yet others might desire the type safety and encapsulation that is provided by the generator tool.

2.3.5 Previously existing code

As mentioned earlier in the research section, the project is entirely self-contained, and uses no previously existing code. The Parser.sml file was written before the design and implementation of the rest of the project as it was used as an exercise in moderately advanced Standard ML, however it was later found useful and therefore was ultimately included as part of the project.

Chapter 3

Implementation

This chapter presents the steps that were taken in implementing the project. It follows the layering highlighted before, which also mostly corresponds to the chronological order in which the code was written.

3.1 Bits layer

As the lowest layer of the program stack, this module offers a variety of services to convert between Standard ML values and the various encodings that can be found in the Protocol Buffers standard: fixed length 32-bit and 64-bit encoding for int and real types, variable length integer encoding for int and bool types, and run length encoding for lists of bytes. It also provides a stream abstraction to use as a medium for byte reading-writing. The relevant source code is in the following files: "Streams.sml" "Bits.sml" and "Varint.sml".

3.1.1 Streams

Many of the decoding algorithms used to deserialise encoded values from a wire message rely on a two step process: identify an initial prefix of the bytestream that contains the field, and then convert that initial prefix into the correct value. This paradigm works well if we assume an imperative, mutable environment: as the values are read an imperative side effect advances the position in the stream. This is the model adopted by the default streams in the Standard ML library, and it works well when used in conjunction with I/O facilities, which are imperative in nature. Relying on this style would however clash with the rest of the program, which is purely functional. As such, a quick replacement was put together

The Bits layer contains two signatures, called *Instream* and *Outstream*, which define the behaviour of immutable Streams. They are respectively characterised by the following functions

```
1 val read : 'a instream -> 'a * 'a instream
2 val write : 'a * 'a outstream -> 'a oustream
```

Using only read and write to interact with these streams would quickly become unwieldy, as they rely on pair parameters/return value: after each read, pattern matching must be used to separate the value from the modified stream, leading to some very cumbersome code every time more than one operation in a row is performed.

To address this problem, a variety of combinators/iterators must be provided by each stream, allowing for better compositionality. Examples of these are functions like *process*, which takes a consumer function and applies it repeatedly to and input stream, collecting the results in a list as they are constructed, or *readWhile*, which takes a predicate on stream elements and continues accumulating values while it holds true.

```
val process : ('a instream -> 'b * 'a instream)
-> 'b list * 'a instream
val readWhile : ('a -> 'b) -> 'a instream ->
'a list * 'a instream
```

Together with the above abstract definitions, a trivial implementation of these streams is provided through list: an input stream is read by taking the head element, while an output stream is written to by adding an element to the head. Conversion between the two lists is achieved by reversing. Since these *ListInstream* and *ListOutstream* are the only implementations provided, whenever a stream is needed in another module, they are referenced directly. If other streams implementations are provided in future, then it will be trivial to convert all stream using structures to functors parametrised by the respective stream module.

3.1.2 Variable length integer encoding

Building on streams, is now possible to implement Variable length integer encoding and decoding. These functions have the following types and code

```
val encodeVarint : int -> Word8.word list
val decodeFromStream : Word8.word list -> int
val decodeFromStream : Word8.word ListInstream.instream
-> int * Word8.word ListInstream.instream

fun encodeVarint x = let fun revEncodeVarint x =
if (x <= 127)
then [zeroMSB x]
else (oneMSB x)::(encodeVarint (Bits.~>>(x,0w7)))
in
rev (revEncodeVarint x)
end

fun decodeVarint [] = raise VarintFormatException
| decodeVarint ls =
let fun decodeVarintF [] acc _ = acc
| decodeVarintF (x::xs) acc i =
let val newAcc = Bits.orb
(acc,iShiftR(iZeroMSB(Word8.toInt x),7*i)) in
decodeVarintF xs newAcc (i+1) end
in
```

```
22 decodeVarintF (rev ls) 0 0
23 end
24 fun decodeFromStream stream =
25 let val (ls,s1) = readWhile(isMSBOne,stream)
26 val (last,s2) = read(s1) in
27 (decodeVarint (ls @ [last]),s2)
28 end
```

This implementation does not rely on mutation,unlike other implementations of the algorithm that can be found on-line. Functions like *zeroMSB*,*oneMSB* are helpers that allow to easily manipulate the bits in their parameters.

The encoding function is quite straightforward: it simply keeps extracting a 7-bit value from the input number, sets the 8th bit to be a one, and keeps going until we are left with only a 7-bit value, to be added with a zero as an 8th bit. In order to avoid liner list access, the whole algorithm runs on a reversed list.

The decoding function is split in two steps: *decodeVarint* takes in a list input containing the encoding bytes. As a precondition, this list must contain just the right amount of bytes, no more, no less. The implementation simply reverses the list, reads the values one by one and keeps accumulating. When the list empty, then the accumulator is returned.

Finally, *decodeFromStream* takes in a byte *ListInstream*, reads off the bytes that encode a variable length integer, and then invokes the list-based *decodeVarint* to do the conversion. Notice that this function conforms to the type

```
1 val decodeFromStream :'a instream -> 'b * 'a instream
```

This means that is possible to call it in conjunction with the input stream iterator functions.

3.1.3 Other functionality

The remaining encodings are located in the Bits structure, together with helper functions. Standard ML does not have a fixed size integer type that is consisted across various environments (for example, the Poly/ML platform, on which the project has been developed, does not posses a Int64.int type, but only an Int32.int type). Therefore the decision was made to rely solely on the omnipresent default int type, and to truncate it if necessary during the conversion into a byte list. The integer to bytes functions are not all that interesting, but present the opportunity to illustrate another recurrent code pattern used throughout the code

```
1 fun fromTemplate(0,acc,_)= rev acc
2 | fromTemplate(n,acc,x) = fromTemplate(n-1,Word8.fromInt(x)::acc,~>>(x,0w8)
    )
3 fun fromInt32(x) = fromTemplate(4,[],x)
4 fun fromInt64(x) = fromTemplate(8,[],x)
5
6 fun toInt xs =
7 let fun inner([],v) = v
8 | inner((x::xs),v) = inner(xs,orb(<<(v,0w8),Word8.toInt(x)))
9 in
```

```
10 inner(rev xs,0)
11 end
```

The pattern consists in factoring out common features from similar functions in a "template" function, and rewrite the specific functions be a specialisation of the template by fixing parameters. The problem faced here was that the *fromInt32* and *fromInt64* functions (that encode integers into bytes) differ only in the number of iterations/bytes to be generated. This is only a very rudimentary example, but this pattern scales well, particularly in combination with higher order functions.

As for the algorithms themselves, they are very simple: in the encode case bytes are generated by grabbing 8-bit values off the given value, accumulating them into the result list, and the decode case simply loops through the supplied list of bytes, ORing them with the result value. As before, the accumulator lists are created backwards for O(1) access and therefore need to be reversed before being returned.

3.2 Wire layer

The Wire layer contains data structures that implement Protocol Buffer's wire messages, and functionality to encode and decode wire messages. The main reason for its existence is to decouple the low-level functionality of the Bits layer from the logical representation of messages in the proto layer. The relevant files are "Wire.sml" and "WireEncoding.sml"

3.2.1 Data structures

The wire message data type aims to give a structured representation of the contents of a binary encoded message. This representation is not strictly necessary, and it is quite possible to restructure the program in a way to directly encode Proto layer messages into the binary format.

However, by introducing this intermediate step, we significantly simplify the encoding/decoding code, by splitting it in two phases: going from messages to (tag,value) pairs, and going from there to bytestreams. Several types are defined at this layer, mirroring the layout of a serialised message.

At the most basic level there is the *wireValue* algebraic data type, with five possible constructors, one per field type. Upon that a *wireField* type is declared, realised as an (int,field) pair, where the int key is the numerical field id. Finally, the *wireMessage* type is declared, aliasing a *wireField* list.

```
1 datatype wireValue = Varint of int
2 | I32 of int
3 | I64 of int
4 | Bytes of Word8.word list
5 | WireMessage of (int * wireValue) list
6 type wireField = int * wireValue
7 type wireMessage = wireField list
```

In addition to the above definitions, several convenience constructors and accessors were written. I will omit them as they are pretty straightforward, single line functions.

3.2.2 Encoding

The rest of the layer, contained in the *WireEncoding* structure, takes care of providing the mapping between *wireMessage* instances and serialised bytestreams. Two main functions are exported by the module

```
val encodeWireMessage : Wire.wireMessage -> Word8.word list
val decodeWireMessage : Word8.word ListInstream.instream
-> Wire.wireMessage
```

Similarly to the Variable length integer conversion case, the encoding function takes an instance of the source value and generates a list of bytes, while the decode function generates the value from a byte instream. The code for the encoding function is:

```
fun encodeWireField(k,Varint(x)) = makeTag(k,tagVarint)::Varint.
    encodeVarint(x)
| encodeWireField(k,I32(x)) = makeTag(k,tagI32)::Bits.fromInt32(x)
(* ... more of the same ...*)
and encodeWireMessage mp =
let fun inner((k,v),acc) = acc @ encodeWireField(k,v) in
(foldl inner [] mp)
end
```

As it is possible to see, *encodeWireMessage* does nothing more than calling *encodeWireField* on each item, and then concatenates the results.

This is also a common pattern throughout the code base: whenever some value is computed by traversing the structure of a message, a function that processes a single field is written, and the message level function is just a repeated application of the field level ones. Covering the specifics, to encode a field, the tag is computed by ORing a type-specific number with the key, and then prepending it to the encoded version of the field's value

As for decoding, the top level function is again just a combinator (this time leveraging the process function from the Stream structure). Field decoding simply works by reading a field tag, switching on the type contained in it, reading the amount of bytes specified, and assembling the wireField from it.

```
fun decodeWireField stream =
let val (tag,rest) = read(stream)
val (fieldNum,fieldType) = tagToNumAndType(tag)
val (v,rest2) =
(case fieldType of
tagVarint => decodeWireVarint(rest)
| tagI32 => decodeWireInt32(rest)
(*... omitted ...*)
in ((fieldNum,v),rest2) end
fun decodeWireMessage stream = process(decodeWireField,stream)
```

3.3 Proto layer and Parser

The Proto layer is the largest module of the system, and the last of the core components of the library. It deals with defining the representation of deserialised Proto messages and a mirroring data structure to represent the definition of a message as appearing into a ".proto" file. It includes a parser to build those definitions, and code that can convert between Proto and Wire messages, completing the serialisation code.

3.3.1 Data structures

The internal representation of a Proto message can be found in the "Proto.sml" source file. Similarly to the definitions shown for the Wire layer, a message is defined as a list of fields (enriched with some extra message-level information such as default values for absent fields), and each field is defined as a *protoValue* accompanied by some metadata (field key, modifier and name). At the end of the chain is the definition of the *protoValue*: an algebraic data type is used to cover all possible cased defined in the standard.

These data structures however are not sufficient for our needs, as they can only represent complete instances of messages, but not their abstract definition (in Object Oriented Programming terms, these are objects, but we need a way to represent classes too). Therefore, a totally symmetric datatype, called *protoMessageType* is defined. It is exactly the same as a *protoMessage*, with its *protoField* instances changed to *protoFieldType* instances. Following the pattern, these in turn are nothing but *protoFields* with their *protoValues* replaced by *protoValueTypes*, an algebraic data type used to express the different constructors of a *protoValue* without supplying a parameter, derived by making each *protoValue* constructor 0-arity and prefixing its name with a capital T

```
1 datatype protoType = TDouble | TFloat
2 | TInt32 | TInt64
3 | TString of String, | TBytes of ....
4 (... etc etc ..)
```

The rest of the file provides convenience functions for accessing *protoMessage*'s entries by name, and conversion functions capable of deriving a*protoMessageType* from a *protoMessage* instance.

3.3.2 Parsing

We now turn at the problem of generating *protoMessageTypes* from ".proto" files. In order to achieve this, a parser is needed. As explained in the preparation section, a parser combinator library was written before the start of the project proper, and it was later decided to incorporate it within the code base. Throughout the explanation of the parser's workings, it is assumed that the source code contained in the file has been read completely into a list of characters. Doing this removes the burden of mixing I/O code with the otherwise pure parser code.

Definition of Parser

Writing a hand crafted parser is no easy task. Most of the more powerful and efficient ways to write a parser rely on symbol tables and state machines that are not meant to be designed by humans. The alternative to these is generally to write a recursive descent parser, with dedicated functions for each syntactic element. However, functional languages allow for a third alternative, which maximizes human friendliness by allowing for a very declarative style: the parser combinator approach.

Let's first supply a definition for a parser type: a function that takes the current parsing state (a datatype that contains global information to the current parsing run, such as the remaining source code, the line and column position, and so on...) and either fails, providing some error-related data, or succeeds, by returning the parsed value (not necessarily a string) and the parsing state on which the next parsing step has to run. The Standard ML definition is:

```
1 type 'a parser = parserState -> ('a * parserState,string)result
```

The result is a type that behaves similarly to the Standard Library 'a option type, with the difference that it carries values on both of its possible constructors (Success and Failure).

Parser combinators

The idea behind parser combinators is than to provide the user with a relatively small number of ad-hoc primitive parsers, and a large number of functions (the combinators) that can build new, sophisticated parsers from simpler ones. In the implementation written for this project, there is only one primitive parser, *anySymbol*, of type char parser. All the others low level parsers, such as *letter, alphaNum, word, number, integer* and *float* are build using the combinators provided. To show the flexibility of this technique, let us first look at some of the combinators signatures:

```
1 val many : 'a parser -> ('a list) parser
2 val many1 : 'a parser -> ('a list)parser
3 val until : 'a parser * 'b parser -> ('b list) parser
4 val wrap : 'a parser * 'b parser * 'c parser -> 'b parser
```

The above collection is just a small sample,with many more available "Parser.sml" source file.

Let's consider the parser combinator *many*, which will create a parser that accepts any number of repetitions of its argument, collected in order in a list, and its variant *many1*, which requires at least one value. A potential use case for this is the definition of number: instead of writing it as an explicit recursive parsing of each individual digit, all that must be done is to write:

```
1 val number = many1(digit)
```

A more involved example could be a parser for quoted string literals, such as "hello world!"

```
1 val stringLiteral =
2 let val quote = symbol #"\""
3 val p = wrap(quote,until(quote,anySymbol),quote) in
4 lift (String.implode) p end
```

This allows us to showcase more functionality. First, look at the definition at line 3 *until(quote, anySymbol)* means "collect any symbol until a quote is met". The outer wrap call stands for "parse the middle parameter, sandwiched by the first two, which are to be ignored". The second part of the function sees the use of lift, familiar to Haskellers. This very important function is best explained by showing its type signature.

```
1 val lift : ('a -> 'b) -> 'a parser -> 'b parser
```

Thanks to lift we can apply some transformation to the result of a parser without having to first run the parser proper.

ProtoParser

With parser combinators in place, it was easy to implement a particular parser for a *protoMessageType*. This was written by piecemeal, starting with parsers for values, then fields, and finally full messages. The code is quite straightforward so no snippets are presented, but can be found in the file "ProtoParser.sml". The only caveat was handling out of order declarations, but this is easily done by introducing a second step of processing: the algorithm scans all the parsed definitions and when a "hole" is found due to a forward declaration, the right sub-message definition is written in.

3.3.3 Encoding

We now turn to the last component of the Proto layer, the *ProtoEncoding* module, containing functions to transform Proto messages into Wire messages and vice versa. The encoding part is quite straightforward, as a proto message is a "richer" than a wire one. By that I mean that the only thing needed is to loop through every field in the proto message, discard all the metadata but the key and only mapping the value to the correct wire container.

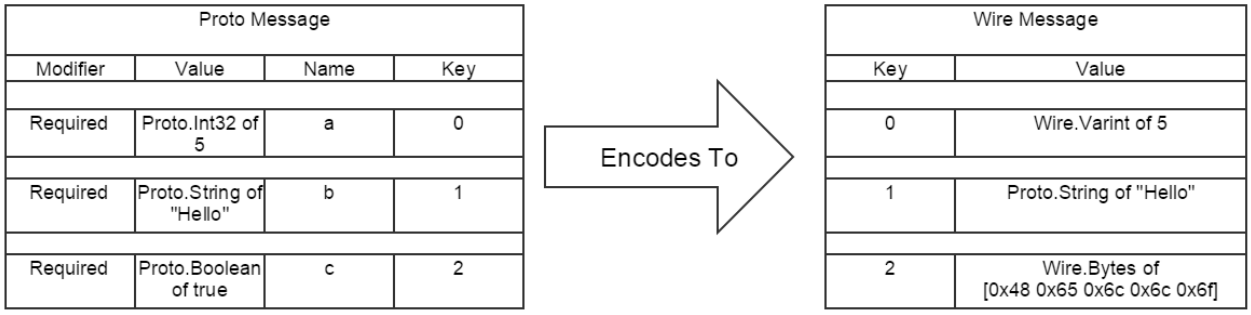

Figure 3.1: Proto to Wire encoding

In the case of *SInt32, SInt64, Sfixed32* and *SFixed64*, the value passed to the wire level constructor needs first to be ZigZag encoded, as explained in the preparation stage. The code for the encoding is contained in two simple functions, located in the file "ZigZag.sml"

```
1 fun encodeZigZag32 n = xorb(<<(n,0w1),~>>(n,0w31))
2 fun encodeZigZag64 n = xorb(<<(n,0w1),~>>(n,0w64)
```

Similarly, floating point numbers need to be casted into an int value of the corresponding bit-length, as wire messages employ integers to represent fixed sized values.

The decoding functionality is slightly more complex, since we are "upcasting" from a bare wire message to a complete proto one. In order to achieve this, a *protoMessageType* needs to be supplied as a parameter. The algorithm then iterates for each field in the *protoMessageType*, extracting the corresponding value in the Wire message, and then trying to build the correct *protoValue* from that. In case of a mismatch (for example, if the definition wants an *Int32*, but the entry in the wire message has been constructed as a *Byte*), the conversion fails and an exception is thrown. The algorithms will also fail if the wire message is lacking some prescribed field (the only exception to this is if a default value for that entry has been specified in the ".proto" file). If the wire message has more fields than expected, the decoding is still considered correct. This is in order to allow for message retro-compatibility.

This completes the Proto layer, and with it the portion of Protocol Buffers standard implemented in this project. We now turn to higher layers, which are concerned with usability and general "quality of life" improvements.

3.4 Lens layer

This layer is divided among two main components: the code implementing functional lenses, in the file "Lens.sml" file, and some proto message specific lens construction code in "ProtoLens.sml". Let us first concentrate on the former.

3.4.1 Functional lenses

The first design decision to be made is how to represent lenses. Across the literature and implementations, three different approaches have been followed. The first, is to simply use a getter and setter pair. The second one, is to rely on a single-constructor algebraic data type definition. The third is to represent it as a single function, that takes the instance the lens wants to be applied to and returns a pair with the getter and setter (partially) applied to it. The three approaches are listed below, in order

```
1 type ('a,'b) lens = ('a -> 'b) * ('b * 'a -> 'a)
2 datatype ('a,'b) lens = Lens of ('a -> 'b) * ('b * 'a -> 'a)
3 type ('a, 'b) lens = 'a -> 'b * ('b * 'a -> 'a)
```

Of the three approaches, the second one was chosen, through a process of elimination. The third one, while theoretically elegant, is needlessly complicated. The first and second one are fundamentally identical, but the latter is easier on the Hindly-Milner type inference engine: as soon as the Lens constructor is met in the code, the typing is solved. On the other hand, working on a pair can introduce ambiguity, particularly in functions that use only one of the two values.

Having sorted out the representation for lenses, we can now proceed to implement *getL* and *setL*, the functions that allow direct application of a lens onto an object. In the code, they are simply called get and set. Another useful function is provided, called

modify, which takes a function and updates the field's value with the result of the function applied to it.

```
1 fun get (Lens(g,_)) x = g(x)
2 fun set (Lens(_,s)) x = s(x)
3 fun modify f (Lens(g,s)) x = s(f(g(x)),x)
```

We can now turn to compose, the function that allows us to chain together lenses.

```
1 val compose : ('a,'b) lens * ('b,'c) lens -> ('a,'c) lens
2 fun compose (a_b,b_c) = Lens (
3 fn a => get b_c(get a_b a),
4 fn (c,a) => set a_b (set b_c (c,get a_b a),a))
```

The code shown above might be a bit confusing, so let's dissect it step by step. From the type, we can see that takes two lenses whose types must "line up". The body of the function is just an application of Lens to two lambdas.

The first of these is the getter, and it is simply constructed by function composition of the parameters' getters. The setter part is more complicated. We can deduce the parameter of the lambda from the desired *('a,'c) lens* result. Moreover, as a composition of setters is intended, we know that there will be two set calls, and once again the types alignment up forces the order of the setters' calls. However, we cannot call the inner setter without somehow converting an 'a into a 'b, and this is done in the only way possible: by applying the getter.

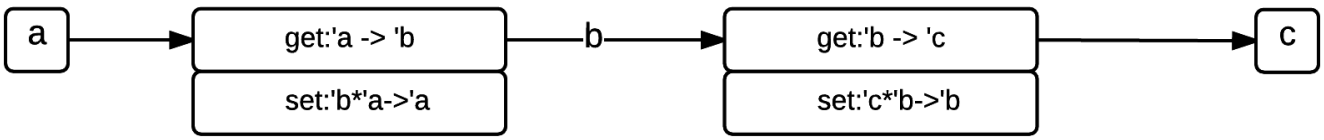

Figure 3.2: Composition of getters

Finally, the file contains many other convenience functions, such as infix symbolic aliases for the functions shown above, and lenses for common types, such as a first and second component lenses for pairs, and head and tail lenses for lists.

3.4.2 Proto specific lenses

The second part of the layer takes care of generating lenses for accessing fields in proto messages. A general lens constructing function called *makeFieldLens* is provided. It works by taking a string value representing a field name, and creates a *(protoMessage,protoValue)* lens that accesses the corresponding entry in a *protoMessage*. This function is not meant for direct use: it is meant to offer a common foundation over which a series of native ML typed functions are build, by wrapping some type converting functions over *makeFieldLens*. By interacting only with these, it is possible to assure a certain degree of type safety. Limitations still exist at this level because some particular Protocol Buffers value

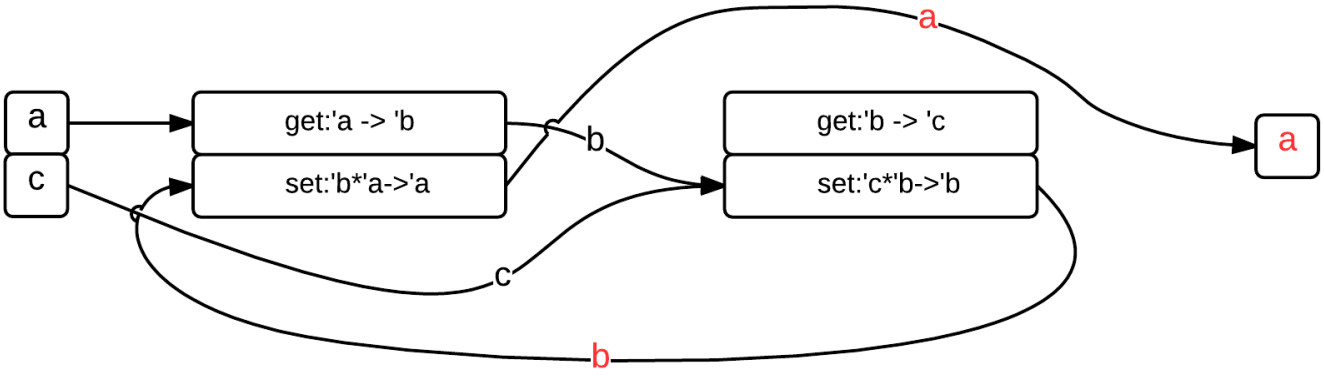

Figure 3.3: Composition of setters. Modified values are in red.

are not supported in Standard ML (such as particularly sized integers). Moreover, fields which contain a *protoMessage* instance are at this point still partially safe, as this level does not assign different types to differently named *protoMessage* instances.

3.5 Generator layer

At the top of the library stack sits the generator layer. This component has two main responsibilities: to offer a convenient point of access for the user, who could ideally use the whole library just interacting with a handful of functions exposed by this module, and to build a type safe interface built on the almost-safe lens layer.

Both of the outlined objectives can be achieved by generating suitable Standard ML code. Given a set of protocol message definitions, it is possible to produce a structure/signature pair that completely captures their behaviour while hiding their proto buffer nature. The procedure follows these steps:

- For each *protoMessageType* in the definition set, with name N, a type *protoMessage* type alias N is defined in the structure, and an opaque type N is defined in the signature. This provides the desired implementation hiding.

- For each field defined in the *protoMessageType* of N a correspondingly native typed lens variable is generated through the use of *makeFieldLens* (for example, a field of type *SInt32* would lead to a generated lens of type int). If the field has type *protoMessage* with name M, then instead of generating a lens from N to *protoMessageType*, we generate a lens from N to M. If M is not present in the set of definitions provided, an error is thrown. This provides type safe access, even in the presence of *protoMessage* fields. The name of the generated lens is a concatenation of the message name N and the field name, following a camel case scheme.

- Two serialisation functions are then created. These are built from the composition of the Proto layer encoding and the Wire layer encoding. These functions are kept

hidden.

- Finally, the procedure's invoker has a variety of options to customize the result. It can choose whether to use transparent matching or opaque matching for the signature. If transparent matching is employed, then it is possible to operate on values of aliased type N with functions that expect *protoMessages*. If opaque matching is selected, then even though values of type N are ultimately implemented as *protoMessage* instances, the type checker will disregard this fact, forcing all the interactions through the generated code.

Other minor options include the name of the file where the generated code is saved, and a boolean flag indicating whether the code is to be immediately loaded into the interpreter. This means that the Generator layer can be used both as a code generation tool and as a way to generate new message definitions at runtime.

Chapter 4

Evaluation

4.1 Planning

Before any evaluation can take place one must first identify which characteristics of the system are to be measured. The two key properties of the system highlighted in the requirement analysis where the correctness of the code implementing the Protocol Buffers standard and ease of use provided by the lenses and generator abstractions.

The former can be verified through a series of serialisation/deserialisation tests, comparing the result of encoding messages with the examples contained in the Protocol Buffers' developer guide.

The latter is harder to evaluate: the only sensible approach is to write two alternative version of the same code, one using the abstractions and the other only the basic method provided. In doing this however one must try to be as impartial as possible, and not purposely craft code that would suit one's agenda of proving his system better. As such, the choice has fallen on the code necessary to configure a simple message with very little nesting. If the utility of lenses is clear in this restricted scenario, then the abstraction is deemed successful, as the benefit of using lenses can only increase with the complexity of the code.

Other evaluation methods were considered, but ultimately discarded. To give insight on the selection process, consider two other obvious tests which could have been carried out: a timing test to measure the performance of the serialisation code and a compatibility test relying on interfacing an example program with existing services that communicate with protocol buffers.

The timing test was deemed irrelevant, as no performance requirement was specified during the initial analysis. While performing other tests the system behaved smoothly, and so no further examinations where performed.

Moreover, the lack of a similar Standard ML library makes any comparison hard. Another issue with timing is that performance varies greatly depending on the hosting Standard ML platform, and that in an actual use case probably the bottleneck would be placed in the network/file communication.

The exclusion of the other test was not motivated by a choice as before, but rather forced. The available services which communicate using Protocol Buffers rely on some advanced features of the standard that were left out (as covered in the Preparation chap-

ter). The implementation of a mock service in Java was considered at some point, but discarded due to timing reasons.

4.2 Correctness

I will now present three tests used to ensure the correctness of the serialisation component of the library. Many other similar tests have been made, but I believe this small sample will be a sufficient representative.

All these tests follow the same structure: a string of code from the Protocol Buffers' developer guide is parsed, the message's fields are set to some specified value, the message is then encoded and decoded. We then compare the encoded stream with the binary representation on the guide, and then the deserialised message with the serialised one. In both cases, if the program is correct, the values should be equal.

First test: single integer field message, showing varint encoding

```
1  val sourceGoogle = "message Test1 = { required int32 a = 1; }"
2  val result = ProtoParser.parseMessages(sourceGoogle)
3  val msgDef = case result of
4  Parser.Success(x) => hd x
5  | Parser.Failure _ => raise TestFailedException
6  val a = ProtoLens.intLens "a"
7  val msg = set a (150,Proto.bareMessage msgDef)
8  val en = ProtoEncoding.encodeMessageToStream(msg)
9  val dec = ProtoEncoding.decodeMessageFromStream(msgDef,en)
10 (* querying the interpreter *)
11 >msg (*pre encoding message*)
12 val it = Message ("Test1", [], [Field (Required, Int32 150, "a", 1)])
13 >dec (*decoded message*)
14 val it = Message ("Test1", [], [Field (Required, Int32 150, "a", 1)])
15 >en (*finally, the bitstream*)
16 val it = [0wx8, 0wx96, 0wx1]
17 (*Developer guide's bitstream found in section Encoding->Simple Message:
18 08 96 01*)
```

Lines 1 to 9 contain the code used to generate the test, the remaining are the queries and responses of the Poly/ML interactive system. We can see that *msg*, the message that encoded, is indeed equal in value to its decoding *dec*. Moreover, the serialised bytestrem *en* is exactly the same as the one prescribed in the developer guide.

Test 2, a string value. The code is structurally identical to that above, so I'll just provide a shortened version.

```
1  val sourceGoogle2 = "message Test2 = { required string b = 2; }"
2  (*... as before *)
3  val en2 = ProtoEncoding.encodeMessageToStream(msg2)
4  val dec2 = ProtoEncoding.decodeMessageFromStream(msgDef2,en2)
5  (* interpreter *)
6  >msg2
7  val it = Message ("Test2", [], [Field (Required, String "testing", "b", 2)
       ])
8  >dec2
```

```
val it = Message ("Test2", [], [Field (Required, String "testing", "b", 2)
    ])
>en2
val it = [0wx12, 0wx7, 0wx74, 0wx65, 0wx73, 0wx74, 0wx69, 0wx6E, 0wx67]
(*Developer guide's bitstream found in section Encoding->Strings
12 07 74 65 73 74 69 6e 67
```

Finally, the third test involves some message nesting

```
val sourceGoogle3 = "message Test3 = { required Test1 c = 3; }"
(*...skipping directly to interpreter querying...*)
>msg3
val it = Message ("Test3", [], [Field(Required,ProtoMessageValue
(Message ("Test1", [], [Field (Required, Int32 150, "a", 1)])), "c",3)])
>dec3
val it = Message ("Test3", [], [Field(Required,ProtoMessageValue
(Message ("Test1", [], [Field (Required, Int32 150, "a", 1)])), "c",3)])
>en3
val it = [0wx1A, 0wx3, 0wx8, 0wx96, 0wx1]
(*Developer guide's bitstream found in section Encoding->Embedded Messages
1a 03 08 96 01 *)
```

The joint code of these tests is found in "TestRun.sml", and the above results can be replicated simply by querying variable values as shown.

4.3 Ease of use

In order to evaluate the improvement of code quality brought by the usage of functional lenses, two code samples of similar purpose are presented, one relying on the native proto message access capacity and the other using functional lenses.

In the example that follows, three message types are used to model a simplistic situation. The messages have both atomic fields and embedded messages.

```
message Country = {
required string countryName = 1;
}
message Maker = {
required string makerName = 1;
required Country makerCountry = 2;
}

message Car = {
required string modelName = 1;
required int32 year = 2;
required Maker modelMaker = 3;
}
```

In the example, a car object will be created in memory, and its fields will be populated with some values. This type of code is ubiquitous in applications relying on Protocol Buffers.

In the following code snippets, common initialisation stages are also skipped, so that the reader may concentrate better on the relevant code. The joint code of this example can be found in the file "Comparison.sml"

First, the traditional approach:

```
1  val country = setStringValue("countryName","Italy",Proto.bareMessage
       countryDef)
2  val maker1 = setStringValue("makerName","Fiat",Proto.bareMessage makerDef)
3  val maker2 = setProtoMessageValue("makerCountry",country,maker1)
4  val car1 = setStringValue("modelName","Panda",Proto.bareMessage carDef)
5  val car2 = setIntValue("year",2008,car1)
6  val car3 = setProtoMessageValue("modelMaker",maker2,car2)
7  (*Let's now attempt to increment the car's year*)
8  val car4 = setIntValue("year",getIntValue("year",car3),car3)
9  (*And let's update the maker, who changed name!*)
10 val car5 = setProtoMessageValue
11 ("modelMaker",
12 setStringValue("makerName","FCA",getProtoMessageValue("modelMaker",car4))
13 ,car4)
14 (*Let's attempt to append a ! to the current country name*)
15 val currentCountry = getProtoMessageValue("makerCountry",
       getProtoMessageValue("modelMaker",car5))
16 val changedName = String.concat[getStringValue("countryName",currentCountry
       ),"!"]
17 val changedCountry = setStringValue("countryName",changedName,
       currentCountry)
18 val changedMaker = setProtoMessageValue("makerCountry",changedCountry,
19 getProtoMessageValue("modelMaker",car5))
20 val car6 = setProtoMessageValue("modelMaker",changedMaker,car5)
```

It is not too important to follow the details of the code above. Just notice that the first few lines of code seem to be pretty straightforward. That is because all the accessed fields belong directly to the root object. However, as soon as we try to perform a modification of children fields (such as changing a car's maker name), we are forced to introduce repetition. In fact, for each level we go deeper within the structure, a whole level of wrapping must be introduced.

This is especially obvious from line 14 onwards, where all the code is written just to append an exclamation mark to the name of the country.First, the country must be read out, then the modification is performed on its name field, and finally the car object needs to be rebuilt with the updated components.

Now, the same code using lenses:

```
1 (*First, lenses needs to be built.*)
2 val countryName = stringLens "countryName"
3 val makerName = stringLens "makerName"
4 val makerCountry = protoMessageLens "makerCountry"
5 val modelName = stringLens "modelName"
6 val year = intLens "year"
7 val modelMaker = protoMessageLens "modelMaker"
```

The lines above are the declarations of the lenses subsequently used. This code has been automatically generated from the message definitions.

```
1  (*now, we can procede to build things*)
2  val country = set countryName ("Italy",Proto.bareMessage countryDef)
3  val maker =
4  sequence[setM makerName "Fiat",setM makerCountry country]
5  (Proto.bareMessage makerDef)
6  val car =
7  sequence[setM modelName "Panda",setM year 2008, setM modelMaker maker]
8  (Proto.bareMessage carDef)
9  (*Increment car's year*)
10 val car2 = modify (fn x => x + 1) year car
11 (*And now update the maker*)
12 val car3 = set (modelMaker +> makerName) ("FCA",car2)
13 (*Apped ! to country name*)
14 val car4 = modify (fn x => String.concat[x,"!"])
15 (modelMaker +> makerCountry +> countryName)
```

The first modification, on line 2, does not seem to be much better than the equivalent code with explicit setters. That is the nature of lenses: their benefits are more apparent as the code gets complex.

Lines 3 and 5 make us of a combinator functions, *sequence*, which iteratively applies setters, saves us from writing multiple incremental definitions of car.In order to use *sequence*, we must rely on *getM*, the curried version of the getter.

Line 18 showcases another common lens accessor: *modify*, which updates a field with the result of a function applied to that field itself (this is akin to a generalized version of the imperative "+=" operation). In order to access inner fields, lenses are composed by the plus-greater then operator, which attempts to mimic imperative dot access. Notice how lines 14-15 are sufficient to replace lines 15 to 20 of the previous example, leading to a great reduction in nesting and code size.

From the above example, it is possible to conclude that using functional lenses, particularly in conjunction with an automated code generator, can reduce significantly the amount of code required to carry out simple field access tasks.

Regarding code clarity, it is possible that at first the more explicit approach, albeit longer and noisier, might be easier to comprehend than code relying on lenses, especially if symbolic operators are used. However, after a period of familiarisation with the library and its keywords, this new approach might result simpler and more pleasurable to work with.

Chapter 5

Conclusion

5.1 Project review

Protocol Buffers for Standard ML was successfully implemented: it supports the required features specified during project design. A core subset of the Protocol Buffers standard is supported and idiomatic interfacing with Proto messages is achieved through the use the abstraction known as functional lenses. Moreover, automated code generation has been implemented as an extra feature.

The final work does however distance itself from the original proposal, where emphasis was put on implementing the full standard rather than on usability: in order to successfully implement functional languages and code generation, support for Remote Procedure Calling and the ability to employ specific optimisations for the serialisation had to be sacrificed.

No particular issue was encountered during the project's realisation. In the project's proposal there was mention of potential problems due to Standard ML having poor support for low level programming. This fear turned out to be false, as all the features required were indeed offered by the environment.

There are not many changes I would suggest should the project be reimplemented. However, I would advise changing slightly the formulation of lenses presented in this project: all of the functions operating on them should be curried by default: especially for setters, expecting tuples parameters does limit significantly the amount of composition possible (even the examples in the evaluation section rely on curried versions). This is not a significant, change, as it is mostly a matter of swapping the names of the curried and uncurried versions.

5.2 Possible extensions

While the project is deemed successful, there is still plenty of room for more work to be done on it. A key area where expansion is possible is finishing the implementation of the protocol buffers language: the core subset supported by the current implementation is probably insufficient for communication with pre-existing commercial quality services.

Another possible route is to support those features which were cut from the project during development: providing support for Remote Procedure Calls stubs (and possibly integrating their invocation within the functional lenses abstraction) and supporting a variety of optimisations during the serialisation step, allowing the user to choose which to employ. Such an extensions fits naturally within the scope of a Part II project, and could be taken by a student in the next future.

Finally, some more possible extensions include offering support for other schema definition languages, such as XML based schemas, and different serialisation formats, such as JSON.This would ideally come close to what the "piqi" project does: being able to arbitrary choose a front-end schema definition language and a back-end serialised format, offering a comprehensive set of tools for data exchange in Standard ML.

Appendix A

Code examples

This appendix contains excerpts of code which may be of interest to the examiners. Note that each section only contains a selected fragment of code contained in each file.

A.1 Varint.sml

```
1  fun zeroMSB x = Word8.andb(Word8.fromInt(x),0w127)
2  fun oneMSB x = Word8.orb(Word8.andb(Word8.fromInt(x),0w127),0w128)
3  fun iZeroMSB x = Bits.andb(x,127)
4  fun iOneMSB x = Bits.orb(x,128)
5  fun iShiftR(x,y) = Bits.<<(x,Word.fromInt y)
6  fun isMSBOne x = Word8.>(Word8.andb(0w128, x),0w0)
7
8  fun encodeVarint x = let fun inner x =
9    if (x <= 127)
10   then [zeroMSB x]
11   else (oneMSB x)::(encodeVarint (Bits.~>>(x,0w7)))
12   in
13      inner x
14   end
15
16 fun decodeVarint [] = raise VarintFormatException
17   | decodeVarint ls =
18       let fun decodeVarintF [] acc _ = acc
19             | decodeVarintF (x::xs) acc i =
20             let val newAcc = Bits.orb(acc,iShiftR(iZeroMSB(Word8.toInt x)
                   ,7*i)) in
21               decodeVarintF xs newAcc (i+1)
22             end
23     in
24       decodeVarintF (rev ls) 0 0
25     end
```

A.2 WireEncoding.sml

```
1  fun makeTagW(k,t) = Word8.orb(Word8.<<(k,0w3),t)
2  fun makeTag(key,typ) = makeTagW(Word8.fromInt key,Word8.fromInt typ)
3
4  fun encodeWireField(k,Varint(x)) = makeTag(k,tagVarint)::(Varint.
       encodeVarint(x))
5    | encodeWireField(k,I32(x)) = makeTag(k,tagI32)::Bits.fromInt32(x)
6    | encodeWireField(k,I64(x)) = makeTag(k,tagI64)::Bits.fromInt64(x)
7    | encodeWireField(k,Bytes(x)) =
8        makeTag(k,tagLenDelimited)::(Varint.encodeVarint(List.length(x))) @ x
9    | encodeWireField(k,WireMessage(x)) = encodeWireField(k,Bytes(
         encodeWireMessage x))
10 and encodeWireMessage mp =
11   let fun inner((k,v),acc) = acc @ encodeWireField(k,v) in
12     (foldl inner [] mp)
13   end
14 fun tagToNumAndType n =
15   let val fieldType = Word8.toInt(Word8.andb(n,0w7))
16       val fieldNum =  Word8.toInt(Word8.~>>(Word8.andb(n,0w248),0w3)) in
17       (fieldNum,fieldType)
18     end
19 fun decodeWireField stream =
20   let val (tag,rest) = read(stream)
21     val (fieldNum,fieldType) = tagToNumAndType(tag)
22     val (v,rest2) = (case fieldType of
23             0 => decodeWireVarint(rest)
24             | 5 => decodeWireInt32(rest)
25             | 1 => decodeWireInt64(rest)
26             | 2 => decodeWireBytes(rest)
27             | x => raise UnknownFieldType(x)) in
28               ((fieldNum,v),rest2) end
29 fun decodeWireMessage stream = process(decodeWireField,stream)
30 fun decodeWireMessageLs ls = decodeWireMessage(ListInstream.fromList ls)
```

A.3 Lens.sml

```
fun get (Lens(g,_)) x = g(x)
fun set (Lens(_,s)) x = s(x)
fun modify f (Lens(g,s)) x = s(f(g(x)),x)
fun compose (a_b,b_c) = Lens (
  fn a => get b_c(get a_b a),
  fn (c,a) => set a_b (set b_c (c,get a_b a),a))
fun bind (l1,f) = fn x =>
  case l1(x) of
    (newState,v) => f(v) newState
fun pure v = fn x => (x,v)
fun sequence [] x = x
| sequence (f::fs) x = sequence fs (f x)
val op+>> = bind
fun chain(f,g) = fn x => g(f(x))
fun chain3(f,g,h) = fn x => h(g(f(x)))
fun chain4(f,g,h,i) = fn x => i(h(g(f(x))))
fun +>(x,y) = compose(x,y)
fun <+(x,y) = compose(y,x)
```

A.4 Parser.sml

```
fun anySymbol (PS(ss,pos)) = case isOver ss of
              false => let val c = current ss in
                    Success(c,PS(next ss,modifyPosition(c,pos)))
                  end
            | true => Failure "End of stream"
fun pSymbol (p,msg) state =
  case anySymbol state of
    Failure _ => Failure msg
  | Success(x,nstate) => if p(x)
               then Success(x,nstate)
               else Failure msg
fun symbol c = pSymbol(fn x => x = c,String.concat[Char.toString c,"
   expected"])
val alpha = pSymbol(Char.isAlpha,"letter expected")
val alphaNum = pSymbol(Char.isAlphaNum,"letter or digit expected")
val digit = pSymbol(Char.isDigit,"digit expected")
val whitespace = pSymbol(Char.isSpace,"whitespace expected")
fun many p state =
  case (many1 p) state of
    Success x => Success x
  | Failure _ => Success([],state)
fun until(test,p) state =
  let fun inner(test,p,accum) state =
    case test state of
      Success _ => Success(rev accum,state)
    | Failure _ => (case p state of
              Failure x => Failure x
```

```
27              | Success (x,rest) => inner(test,p,x::accum) rest)
28      in
29        inner(test,p,[]) state
30      end
```

A.5 ProtoLensGen.sml

```
1  fun lensNameFromType x = String.concat ["ProtoLens.",nativeNameFromType x,"
       Lens"]
2  datatype capitalization = NO | UPPERFIRST
3  fun cleanName (upper,x) =
4    let val s = StringManipulation.replaceNumbersWithLetters x in
5      case upper of
6        UPPERFIRST => StringManipulation.upperFirst(s)
7      | NO => s
8    end
9  fun generateLeftSide (msgName,fieldName) = ["val ",cleanName (NO,msgName),
       cleanName (UPPERFIRST,fieldName)]
10 fun generateValueRightSide (fieldType,fieldName) = [" = ", lensNameFromType
       fieldType, " \"",fieldName,"\"\n"]
11 fun generateTypeRightSide'(fieldType,specialNames) =
12   case fieldType of
13     TProtoMessage msg => (case List.find (fn x => Proto.messageDefName(msg)
           = x) specialNames of
14                 NONE => "protoMessage"
15               | SOME(n) => n)
16   | t => nativeNameFromType t
17 fun generateTypeRightSide (fieldType,specialNames) =
18   [" : ", generateTypeRightSide'(fieldType,specialNames)," ", "ProtoLens.
       protoLens\n"]
19 fun generateFieldLensCode msgName (FieldDef(_,fieldType,fieldName,_)) =
20   String.concat (generateLeftSide(msgName,fieldName) @
       generateValueRightSide(fieldType,fieldName))
21 fun generateFieldLensTypeCode (msgName,specialNames) (FieldDef(_,fieldType,
       fieldName,_)) =
22   String.concat(generateLeftSide(msgName,fieldName) @ generateTypeRightSide
       (fieldType,specialNames))
23 fun generateMessageLensCode (MessageDef(name,_,fields)) =
24   String.concat(map (generateFieldLensCode name) fields)
25 fun generateMessageLensTypeCode (MessageDef(name,_,fields),ls) =
26   String.concat(map (generateFieldLensTypeCode (name,ls)) fields)
27 fun packUpStructure(structName,definedTypes,structBody,structSig,isOpaque)
       =
28   String.concat["structure ", structName, " = \n",
29         "struct\n",
30         definedTypes,
31         structBody,
32         "\nend", if isOpaque then ":>" else ":",
33         "sig\n",
34         definedTypes,
35         structSig,
```

```
        "\nend"]
fun genStructMessageDef (msgDef, specialNames)=
  let val name = messageDefName msgDef in
      (String.concat["type ",name," = Proto.protoMessage"],
      generateMessageLensCode(msgDef),
      generateMessageLensTypeCode(msgDef,specialNames))
    end
fun combineTriples f ((x1,y1,z1),(x2,y2,z2)) =
  (f(x1,x2),f(y1,y2),f(z1,z2))
fun triple x = (x,x,x)
fun generateStructureBodyFromMessages(msgs, specialNames) =
  let val ls = map (fn x => genStructMessageDef(x,specialNames)) msgs
    fun f(x,y) = String.concat[x,"\n",y] in
    foldl (combineTriples f) (triple "") ls
  end
fun generateStructure(name, msgs) =
  let val specialNames = map messageDefName msgs
    val generated = generateStructureBodyFromMessages(msgs,specialNames) in
      case generated of
        (types,code,spec) => packUpStructure(name,types,code,spec,false)
    end
```

Federico Pizzuti
Jesus
fp296

Diploma in Computer Science Project Proposal

Protocol Buffers for Standard ML

22-10-2014

Project Originator: Federico Pizzuti

Resources Required: See Resource Required Section

Project Supervisor: Prof. Lawrence Paulson

Signature:

Director of Studies: Prof. Cecilia Mascolo

Signature:

Overseers: David Greaves and John Daugman

Signatures:

Introduction and Description of the Work

Protocol Buffers are a data interchange format developed at Google. It aims to be a language agnostic, compact and efficient way to exchange messages between different systems, in a way that is similar to JSON and XML. To make use of Protocol Buffers, a developer must first write down the description files for the data structures (from now on referred as messages) he wishes to represent. The language used is simple yet powerful enough to describe a wide array of basic datatypes, such as many different representation for integer and floating point numbers, strings and other embedded messages. It allows for different sorts of optimizations to be used during the serialization of messages, amongst which are optimisations for speed or code size, and the serialisation used is itself rich in tecniques such as different types of integer encodings according to their intended usage. The language is designed so that it is possible to extend messages while keeping backwords compatibility.Once the developer has written or otherwise obtained the .proto files, these are then passed to a language-specific tool that will generate the code needed to implement the protocol buffer in that specific language, and to be able to interact within that language environment in a idiomatic way. Most implementations require a small language-dependant runtime to be included in the projects using protocol buffers. Lastly, protocol buffers offer the ability to define messages-based RPC interfaces and stubs, to be used with an arbitrary rpc implementation. The objective of this Part II project is to implement the tool and runtime for the Standard ML language, and specifically for the Poly/ML system. The language of choice for the tool is Standard ML itself.

Resources Required

No particular resource is required, besides a computer capable of running the Poly/ML system. I expect to perform most of the work on my personal computer, using some online repository system (probably github) for back-up purposes.

Starting Point

At the current time, I have read the serialisation format and the API of the Java implementation of protocol buffers, and skimmed the Haskell one to see how the problem is handled in a more functional language. I have not written any code yet. I am relatively comfortable with the Standard ML programming language, but I expect to have to study it's tools (particulalry the sml-yacc tool, which I expect to use for the creation of the parser for .proto files).

Substance and Structure of the Project

As previously stated, the project will mainly consist of two parts: the tool and the runtime. At this point, it is important to explain the two main "strategies" that can be used to implement the code generation: one is to use some particular representation for all messages, and to write library functions to serialize-deserialize messages from/into that common format. The tool would then have to create the code to generate the particular arrangment for each message, but then the code for serialisation/deserialisation would all be contained in the runtime. Accessor functions to the message's field would be created by the tool. The whole thing would be wrapped into an ML structure, provided with a corrisponding signature, to abstract away the implementation. While this simplifies significantly the amount of generation that the tool has to do, it means that a runtime library needs to be shipped to make use of the protocol buffers. A second strategy would be to generate "ad-hoc" serialisation code and data structures for each message. While doing this will lead to more code being generated and duplicated, it would also mean that no runtime would be required as it is all "in each structure". Again, a signature would be generated at the same time and therefore the rest of the application would not need to know exactly which implementation has been used. I intend to support both strategies, each corrisponding to the optimisations "optimise_for=CODE" and "optimise_for=SPEED" in the protocol buffer's language. Regarding the RPC features, I intend to implement the stub and interface creation, togheter with the implementation of a simple RPC system, to demonstrate it's effectiveness. A possible extension is to allow the library to work with other formats such as JSON and XML. In order to test the project is functioning correctly, it might be required for me to write some small test applications, to show the project's succes. It would be ideal to write something along the lines of a simple chat server-client, with the clients written in a variety of languages (probably SML and Java) which use a common set of proto files. I am currently looking for some enstablished online service which offers a Protocol Buffer API: if that is the case, I could use that service (with their .proto files) to show the project success.

Success Criteria

- The tool must be able to create Standard ML structure definitions implementing messages given the protocol buffer format, and to serialise/deserialise them correctly.
- The code generated should be able to be used in a Standard ML program in an idiomatic and natural way.
- The tool must be able to offer different optimisations (CODE and SPEED), dimostrating the difference in outcomes dependently on which one is chosen.

- The runtime should offer facility to employ an RPC system, though careful design of signatures to be matched
- The runtime should provide a simple RCP implementation for the above-specified signatures.

Timetable and Milestones

As suggested in the pink book, I will divide my timetable in ten forthinight-long packages

Weeks 1 and 2

By the end of week one, I expect project design to be complete, and the details of the internal "default" representation of a protocol buffer message should be worked out. By week 2, the tool's parser should be complete.

Weeks 3 and 4

In this weeks, I intend to write the bulk of the serialisation/deserialisation code with respect to the internal representation.

Weeks 5 and 6

With the internal representation's serialisation code complete, I can now write the code generation part of the tool for protocol messages leveraging it. I intend to be able to take a "core subset" of a .proto file and generate an ML structure implementing it by the end of week 6

Weeks 7 and 8

In these weeks I intend to work on the ad-hoc method of code generation. I intend to mostly leverage the already written internal representation code, readapting it for the ad-hoc case. At this point, I will start writing the test program to show the project's success.

Weeks 9 and 10

Add support in the tool and runtime for RPC stubs. Start working on the implementation of a simple RPC

Weeks 11 and 12

Finish off the RPC implementation. Enrich the sample application with some service to demonstrate the functionality of the RPC system

Weeks 13 and 14

Testing, fixing, and catching up if something went wrong earlier.

Weeks 15 and 16

Dissertation writing

Weeks 17 and 18

Dissertation writing

Weeks 19 and 20

Dissertation writing and handing in