Executable rules of encoding

John P. T. Moore
The University of West London
St. Mary's Road, London W5 5RF, UK
moorejo@uwl.ac.uk

ABSTRACT

Encoding rules can be found in telecommunications standards documents. These documents try to explain in an unambiguous manner how a developer might transform a protocol described in an abstract language into a more concrete binary form by following a set of rules. They are often difficult to read and difficult to understand.

In this paper we show how a dynamic language can be used to help describe and understand a formal process such as applying a set of encoding rules to some data. We achieve this by taking the homoiconic approach of describing the encoding process in the abstract language used to describe protocols and then execute our result in a Scheme REPL.

Categories and Subject Descriptors

D.3.2 [**Programming Languages**]: Language Classifications Applicative (functional) languages

General Terms

Design, Languages

Keywords

Lisp, Scheme, Packedobjects

1. INTRODUCTION

In this paper we describe the encoding process of the Packedobjects tool [6] by specifying the process in its own abstract language and experimenting with the result in a Scheme REPL. This approach provides a more dynamic and interactive way of formalising the process compared to the traditional approach of reading and interpreting standards documents such as ITU-T X.691 [3]. Before illustrating an example we will provide some background to the work.

1.1 Background

Packedobjects is heavily influenced by Abstract Syntax Notation One (ASN.1) [2] and as such uses an abstract language to describe network protocols. Data that is described according to this abstract language must be transformed into something more suitable for communication across a network by applying a set of rules such as Packed Encoding Rules (PER) [3]. Packedobjects is designed to encode data as concisely as possible and therefore is based on the unaligned variant of PER. The main difference between Packedobjects and tools based on ASN.1/PER is that Packedobjects adopts a more dynamic approach. Instead of having to parse an abstract language and generate high-level code, Packedobjects uses s-expressions to describe protocols. Simple list manipulation takes place to transform a protocol into something suitable for passing to a low-level C based encoder. This approach lets us script network communication and allows us to easily make changes to deployed applications - a key feature in embedded computing. Using Scheme we can embed this functionality into more traditional C applications. We will illustrate a simple example of encoding a UK postcode and then show how the data is encoded.

1.2 Packedobjects

Postcodes in the UK are alphanumeric and consist of an outward and inward part separated by a space. The following code will define their structure using Packedobjects.

```
(define 1letter '(1letter string (size 1)))
(define 2letters '(2letters string (size 2)))
(define 1digit '(1digit integer (range 0 9)))
(define 2digits '(2digits integer (range 0 99)))

;; to preserve unique namespace of A9A sequence
(define anotherletter
  (cons 'anotherletter (cdr 1letter)))

(define proto
  `(postcode sequence
       (outward choice
          (A9 sequence ,1letter ,1digit)
          (A99 sequence ,1letter ,2digits)
          (AA9 sequence ,2letters ,1digit)
          (AA99 sequence ,2letters ,2digits)
          (A9A sequence ,1letter ,1digit ,anotherletter)
          (AA9A sequence ,2letters ,1digit ,1letter))
       (inward sequence ,1digit ,2letters)))
```

The first section of code defines some string and integer data types with suitable constraints on their size and range respectively. We reuse these definitions in our protocol by using simple in-built mechanisms available in Scheme, in this case unquote. The protocol shows the outward and inward compound data types. The outward data type consists of another compound data type representing a choice of postcode formats. To encode the postcode "HA9 0WS" we supply the following data.

```
(define data
  '(postcode
    (outward
      (AA9
       (2letters "HA")
       (1digit 9)))
     (inward
      (1digit 0)
      (2letters "WS"))))
```

Encoding the data produces a string.

```
guile> (string->list (encode proto data))
(#\R #\space #\310 #\W #\246)
```

To understand how we arrived at this string we need to describe our rules of encoding. There are a number of ways to do this including using a textual description of the process or by using a more concise mathematical approach, for example Z notation [1]. However, we already have an abstract language which is flexible enough to apply to a range of scenarios including something as obscure as shopping at a supermarket to more practical problems such as representing system time [4]. Moreover, the result of what we describe in our abstract language is executable within the REPL allowing us to experiment and confirm our understanding of the process. We will first describe how various data types are encoded and then show how this applies to our postcode example.

2. DEFINITION OF RULES

Describing our rules of encoding should allow others to implement corresponding encoding software. The Packedobjects Reference manual [6] provides examples of this process for all data types. The manual also formally describes the abstract language used. In the following subsections we will now use this abstract language as the language to describe the encoding process.

2.1 Integers

We will start by defining how integers are represented. This forms the basis of how all other types are encoded. Integers are mapped to a low-level encoder [5] based on the range of values they represent. They can be represented with three functions corresponding to three categories of integer as follows:

```
(define (semi-constrained-integer lb)
  `(n integer (range ,lb max)))

(define (constrained-integer lb ub)
  `(n integer (range ,lb ,ub)))

(define (unconstrained-integer)
  '(n integer (range min max)))
```

The Packedobjects manual provides further details of how these integer forms are transformed into a core form suitable for the low-level encoder.

2.2 Strings

Using our integer definitions we can describe how a string is encoded:

```
(define (string)
  `(s sequence-of
      ,(constrained-integer 0 127)))
```

We show how the abstract language of Packedobjects is easily employed to describe how a string consists of a sequence of characters whose values must be encoded within a specified range. From this definition we can show how three categories of string are described:

```
(define (semi-constrained-string)
  `(semi-constrained-string sequence
           ,(semi-constrained-integer)
           ,(string)))

(define (constrained-string)
  `(constrained-string sequence
           ,(constrained-integer)
           ,(string)))

(define (fixed-length-string)
  (string))
```

The above shows that all strings which are not fixed in length require a length to be encoded and this length is represented as either a semi-constrained integer or constrained integer accordingly. Packedobjects has five types of string which includes string, octet-string, bit-string, hex-string, and numeric-string. In terms of encoding they only differ by the range of values they can represent. For example a bit-string is expressed as follows:

```
(define (bit-string)
  `(bit-string sequence-of
         ,(constrained-integer 0 1)))
```

This time we can see that the string is constrained to be an integer with either the value zero or one. As with any string type there are three categories representing semi-constrained, constrained and fixed length strings.

2.3 Atomic types

Both strings and integers are atomic types. Packedobjects has other atomic types which includes enumerated, boolean and null. An enumerated type can be defined as follows:

```
(define (enumerated len)
  (constrained-integer 0 len))
```

The above represents how Packedobjects allows a single choice to be made from a list or enumeration of values and that this choice is encoded from zero. Encoding a boolean is equally straight forward:

```
(define (boolean)
  (constrained-integer 0 1))
```

This time we know both bounds for the constrained integer as opposed to just knowing the lower bound in the enumeration example. A null type does not require any value to be encoded.

2.4 Compound types

Compound types must contain other types and provide a mechanism for representing choice and repetition. In addition, they allow the creation of complex nested protocol definitions. There are various sequence types which include sequence, sequence-optional and sequence-of. No encoding is required to represent a sequence. A sequence-optional requires a bitmap to indicate which values of a sequence are present. It can be described as follows:

```
(define (sequence-optional)
  (bit-string))
```

Here we see that this bitmap can be represented as a sequence of characters which in this case is simply a bit-string. For example this definition is expanded into:

```
(bit-string sequence-of (n integer (range 0 1)))
```

A sequence-of type requires a value to be encoded to represent how many times the sequence repeats. This can be defined as follows:

```
(define (sequence-of)
  `(semi-constrained-integer 0))
```

A sequence-of is unbounded in size therefore we represent the value as a semi-constrained integer which has a lower bound of zero. In addition to the different types of sequence the other compound data type is choice. The encoding of a choice is very similar to an enumeration apart from the first item in a list of choices is encoded from one. A choice can be defined as follows:

```
(define (choice len)
  (constrained-integer 1 len))
```

As we can see a choice uses a constrained integer that begins at one.

3. EXECUTION OF RULES

Now that we have defined some rules for encoding we can try and describe the encoding of our postcode example and then reinforce our understanding of this description by supplying values in the REPL. We can describe our encoding as a sequence as follows:

```
(define proto
  `(encoding sequence
        ,(contains ,(choice 6))
        ,(contains ,(fixed-length-string))
        ,(contains ,(constrained-integer 0 9))
        ,(contains ,(constrained-integer 0 9))
        ,(contains ,(fixed-length-string))))
```

To preserve a unique namespace within the sequence we wrap each entry in its own sequence. This can be achieved by a simple macro such as:

```
(define-syntax contains
  (syntax-rules ()
    ((_ thing)
     `(,(gensym "s") sequence thing))))
```

It is important to note, we have concisely described the mechanics behind the encoding process which is suitable information for someone implementing an encoder but not what a protocol designer would do. They would be focusing on how data is structured at a higher-level rather than how it is encoded.

Now that we have defined our encoding using our high-level definitions we will pretty-print the result to obtain the expanded protocol and use this version instead:

```
(define proto
  '(encoding
    sequence
    (s151 sequence (n integer (range 1 6)))
    (s150 sequence
          (s sequence-of (n integer (range 0 127))))
    (s149 sequence (n integer (range 0 9)))
    (s148 sequence (n integer (range 0 9)))
    (s147 sequence
          (s sequence-of (n integer (range 0 127))))))
```

In this version we can see how we wrapped each part of the encoding sequence in its own sequence with a unique identifier generated by a call to the gensym function. From this fully expanded version we can now specify some values:

```
(define data
  '(encoding
    (s151 (n 3))
    (s150 (s ((n 72)) ((n 65))))
    (s149 (n 9))
    (s148 (n 0))
    (s147 (s ((n 87)) ((n 83))))))
```

Now that we have defined both the protocol and values we can encode the values and also decode the result to confirm correctness:

```
guile> (pretty-print (decode proto (encode proto data)))
(encoding
  (s151 (n 3))
  (s150 (s ((n 72)) ((n 65))))
  (s149 (n 9))
  (s148 (n 0))
  (s147 (s ((n 87)) ((n 83)))))
```

4. CONCLUSIONS

In this paper we have described a dynamic way of describing a set of encoding rules which allows experimentation within the Scheme REPL. This approach not only allows us to concisely specify the mechanics of the encoding but also allows us to reinforce our understanding by executing the result. The language used to describe our encoding rules is the same language we used to describe network protocols. This demonstrates the flexible nature of the abstract language and allows us to work with a syntax we are already familiar with. We introduced our own high-level functions to allow concise descriptions to be written. These functions are eventually expanded into the correct abstract language required for use by the Packedobjects tool. The outcome is we are able to offer an alternative approach to the traditional method of formally describing encoding rules using standards documents.

5. REFERENCES

[1] A.~Cerone. Representing ASN. 1 in Z. In *Proceedings of the Australasian information security workshop conference on ACSW frontiers 2003-Volume 21*, pages 9–16. Australian Computer Society, Inc., 2003.

[2] International Telecommunication Union. Specification of Abstract Syntax Notation One (ASN.1). ITU-T Recommendation X.208, 1988.

[3] International Telecommunication Union. Abstract Syntax Notation One (ASN.1): Specification of Packed Encoding Rules (PER). ITU-T Recommendation X.691, July 2002.

[4] J.~Moore. Get stuffed: Tightly packed abstract protocols in Scheme. The 10^{th} Scheme and Functional Programming Workshop, 2009.

[5] J.~Moore. Everything counts in small amounts. International Workshop on Dynamic languages for Robotic and Sensor systems (DYROS), November 2010.

[6] J.~Moore. Packedobjects Reference Manual. http://zedstar.org/packedobjects/, Aug. 2010.