

A dynamic data encoder for embedded systems

John P. T. Moore
jmoore@zedstar.org

Abstract

This paper describes a data encoding tool which utilises the "data is code" principle of symbolic expressions available in Lisp-like languages to allow the scripting of binary network protocols. The approach taken has its origins in the domain of games development where it is common place to embed a script language into a compiled program to allow customisation of a deployed binary. This dynamic approach provides specific flexibility when working on embedded systems as it reduces the amount of cross compilation and deploy cycles that occur following more traditional development approaches. In addition, the separation of how the data is encoded from the compiled application facilitates a concept known as extensibility of the network protocol.

1. Introduction

The International Standards Organisation (OSI) 7 Layer reference model provided an academic framework for the design of network protocols and standards [14, 15]. In comparison to the OSI model, the TCP/IP model adopted a more simplified approach where amongst other changes the Presentation Layer was consumed by the Application Layer. As such, a network applications programmer needs to consider how to structure their data when transferring it across an internet. A number of competing technologies have been developed and continue to develop in this area. When it comes to structuring data in a human-readable way, XML has dominated. However, approaches to structuring binary data range from serialising native data structures to transforming an abstract syntax into a more concise binary form. Binary protocols continue to play an important role in supporting network applications. Common uses include network games [8] and mobile communication. In addition, Google released their work on Protocol Buffers [3] which was created to address issues they faced in the area of high performance computing [1]. In this paper we discuss a tool known as Packedobjects¹, a cross platform bit stuffing tool targeting embedded devices. The software has the following features:

- Uses a high-level abstract syntax to describe network protocols
- Allows scriptable network protocols
- Produces platform independent concise binary protocols

¹<http://gitorious.org/packedobjects>

[Copyright notice will appear here once 'preprint' option is removed.]

1.1 Extension language

Packedobjects is available as a module for GNU Guile² which in turn is available as a C library [13]. By linking with this library you gain access to a Scheme interpreter which amongst other things will allow manipulation of structured data in the form of symbolic expressions (s-expressions). This approach of embedding an interpreter allows a separation of the network code and the compiled C program. Being able to script a binary protocol means we can dynamically alter its structure without the need to recompile the main program. This facilitates a development cycle which reduces the amount of cross compilation that would be required for an embedded device if we exclusively used the traditional programming languages such as C and C++. Often, the traditional methods (see figure 1) which require a compiler to transform an abstract syntax into program code are untested in cross compilation environments and therefore may not work. In addition to this added flexibility we also obtain a degree of extensibility of the network protocol.

1.2 Extensibility

The concept of extensibility can be confusing, especially to those who have worked exclusively with text-based network protocols that highly structure the data they communicate. With this extra structure comes flexibility. It allows an application to receive a message and silently ignore parts of the message it does not understand or recognise. XML is a good example of a technology which facilitates this approach. The structure or tags placed around the data within the message provide all the information required to understand the real payload. This approach is in direct conflict for a protocol designer who strives to minimise every bit of information communicated. Although binary protocols can still follow a similar tag based approach it is common to try and further optimise the solution so only the minimal amount of data required to be decoded successfully is actually communicated. The challenge is producing binary protocols which are not fragile or easily broken by simple changes in the protocol definition. This future proof design approach is known as extensibility. Thus, extensibility refers to the way that network communications can continue between parties A and B even though party B may have updated the way it communicates. The following will illustrate a simple example. Party A uses the following protocol:

```
(define protocol-version-1
  '(foo choice
    (message-A boolean)
    (message-B boolean)))
```

Party B updates its protocol to the following:

```
(define protocol-version-2
  '(foo choice
    (message-A boolean)
    (message-B boolean)))
```

²<http://www.gnu.org/software/guile/>

```
(message-C boolean))
```

At this point parties A and B may produce incompatible encodings. For example, it is not possible for party B to communicate message-C with party A because party A has no knowledge of such a message. Party A would be expecting an encoding based on a choice between 2 messages. Encoding standards such as Packed Encoding Rules (PER) handle this situation using special notation in the protocol syntax to indicate the likelihood that specific parts of the specification will be extended and then encode some extra structure to facilitate this [7]. Packedobjects does not require this. What is required is that both parties obtain the same version-2 protocol. In this case, even though party A will never use message-C, it can still receive the message and can choose to silently ignore it. The party A program does not need to be recompiled because the protocol is available via a Scheme script which can be dynamically loaded or bootstrapped over a simple HTTP request. This provides a more stable upgrade option for deployed devices allowing them to continue working with restricted functionality until a software upgrade can be authorised by the user. The ability to maintain communication across mass-deployed devices can be a key goal in the domain of embedded communication technologies.

Having introduced some of the key design goals of Packedobjects, we will first provide an overview of some other relevant techniques for producing binary protocols and then go on to describe Packedobjects in more detail.

2. Related work

Serialising data structures for transmission across a network is a common technique. The programmer might have to handle differences in byte ordering if communication takes place across different hardware platforms. In addition, the protocol designer is restricted to describing the network protocol in terms of the native data structures available in the language used. Although languages such as Erlang provide excellent support for working with binary data [4], an alternative more abstract approach may be taken.

Describing network protocols in a way that is independent of the application programming language used introduces some complexity. Ultimately this abstract syntax will need to be represented by the programming language. The traditional way of handling this is not dynamic. A compiler is used to transform the abstract syntax into native language code. Typically the code generated will be combined with application specific code and linked with vendor supplied code. Figure 1 summarises the approach which is taken by numerous Abstract Syntax Notation 1 (ASN.1) tools [2, 10]. ASN.1 originates from the world of telecommunications [5]. The philosophy of ASN.1 is to provide a rich abstract syntax to describe network protocols and this syntax should be transferred into binary before transmission. Different techniques, or encoding rules, can be applied to make this transition from abstract syntax to binary. The abstract syntax allows the protocol designer to think at a higher level and provides a common ground between application developers working in different programming languages. Protocol Buffers adopts a similar approach where the abstract syntax used to describe a message is transformed into Classes together with methods to set, query and encode the data. Packedobjects takes a more dynamic approach which uses s-expressions from the Scheme programming language to exploit the concept of "data is code" and therefore bypasses the need for a compiler. In keeping with a minimalist tradition adopted by Scheme, Packedobjects uses a simplified subset of the ASN.1 standard when describing protocols. By simplifying the abstract syntax we can provide a dynamic runtime representation within an s-expression which encourages exploration in the read-eval-print loop (REPL).

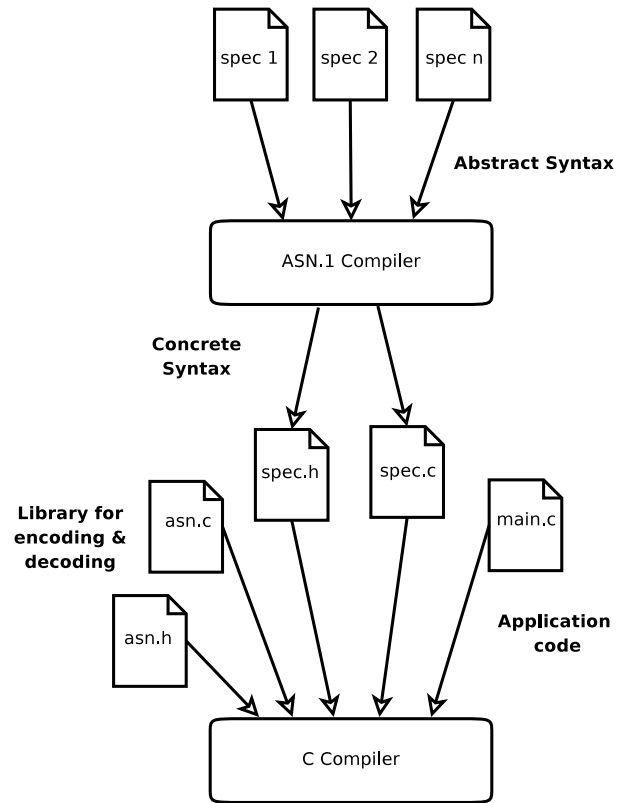


Figure 1. ASN.1 to C compiler

The main disadvantage of adopting such a dynamic approach is a loss of runtime performance, the significance of which depends on the type of embedded device used and the nature of the network [9]. For example, Packedobjects has been tested successfully on various embedded Linux devices which resemble the performance of desktop computers from perhaps a decade ago. The software has also been tested over latency bound networks where CPU performance has little relevance [12].

The following subsections will contrast the abstract syntax used by ASN.1, Protocol Buffers and Packedobjects using an example of a baseball scorecard.

2.1 ASN.1

```

-- Baseball Card Abstract Syntax (BCAS)
BCAS DEFINITIONS ::= BEGIN
  BBCard ::= SEQUENCE {
    name IA5String (SIZE (1..60)),
    team IA5String (SIZE (1..60)),
    age INTEGER (1..100),
    position IA5String (SIZE (1..60)),
    handedness ENUMERATED {
      left-handed(0),
      right-handed(1),
      ambidextrous(2)},
    batting-average REAL
  }
END
  
```

2.2 Protocol Buffers

```

// Baseball Card Abstract Syntax (BCAS)
message BBCard {
  required string name = 1;
  
```

```

required string team = 2;
required int32 age = 3;
required string position = 4;
enum Handedness {
  LEFT_HANDED = 0;
  RIGHT_HANDED = 1;
  AMBIDEXTROUS = 2;
}
required float batting_average = 5;
}

```

2.3 Packedobjects

```

;; Baseball Card Abstract Syntax (BCAS)
(define bocard
  '(bocard
    sequence
    (name string (size 1 60))
    (team string (size 1 60))
    (age integer (range 1 100))
    (position string (size 1 60))
    (handedness enumerated
      (left-handed
       right-handed
       ambidextrous))
    (batting-average
     sequence
     (mantissa integer ())
     (base enumerated (2 10))
     (exponent integer ())))))

```

Protocol Buffers uses a syntax closer to a traditional programming language than the other two approaches. Both ASN.1 and Packedobjects use constraints on the integer and string data types to help reduce the encoding size. All three approaches provide additional types which include the ability to nest types and specify repetition. The following section will formally introduce the complete syntax of Packedobjects.

3. Domain specific language

The language we represent using an s-expression can be referred to as a Domain Specific Language (DSL) with its purpose to specify structured data for communication across a computer network. The subsections that follow will introduce the DSL by formally specifying the grammar for the protocol description language together with the grammar for the data that needs to be communicated.

3.1 Protocol grammar

```

(protocol ::= (form))

(form) ::= (atomic type)
| (atomic type) (form)
| (compound type)
| (compound type) (form)

(atomic type) ::= ((id) integer (range))
| ((id) string (size))
| ((id) bit-string (size))
| ((id) octet-string (size))
| ((id) hex-string (size))
| ((id) numeric-string (size))
| ((id) enumerated (enum))
| ((id) boolean)
| ((id) null)

(compound type) ::= ((id) sequence (form))
| ((id) sequence-optional (form))
| ((id) sequence-of (form))
| ((id) choice (form))

```

```

(range) ::= ()
| (range min max)
| (range min (integer))
| (range (integer) max)
| (range (integer) (integer))

```

```

(size) ::= ()
| (size min max)
| (size min (integer))
| (size (integer) max)
| (size (integer) (integer))
| (size (integer))

```

```

(enum) ::= ((symbol))
| ((symbol) (enum))
| ((number))
| ((number) (enum))

```

```

(id) ::= (symbol)

```

3.2 Data grammar

```

(data) ::= (form)

```

```

(form) ::= (atomic value)
| (compound value)

```

```

(atomic value) ::= ((id) (string))
| ((id) (integer))
| ((id) (symbol))
| ((id) (boolean))
| ((id))

```

```

(compound value) ::= ((id) (normal form))
| ((id) (special form))

```

```

(normal form) ::= (form)
| (form) (normal form)

```

```

(special form) ::= ((form))
| ((form) (special form))

```

```

(id) ::= (symbol)

```

4. Data types

The DSL used by Packedobjects consists of two categories of data type: atomic and compound. An atomic data type specifies a single value to be encoded whereas a compound data type consists of one or more atomic and/or compound data types. The compound data types include the various sequence types and the choice type. The following subsections will introduce all the data types available. In section 5 we will go on to show how these types are transformed by the encoding process through example.

4.1 Integers

The integer data type is a core type. All other types are transformed into this type before being mapped onto the encoder. The integer type uses visible subtype constraints to optimise the encoding. Subtyping in this case is used to restrict the range of values allowed for an integer value. The ability to customise data types produces efficient encodings [6]. Not only are less bits sent across the communications link but also more optimised encoder/decoder implementations can be built to handle specific protocols. Constraints are specified using the range syntax, the effect of which is detailed in subsection 5.2.

4.2 Strings

There are various string types that differ according to the type of characters they represent and therefore the amount of bits they need when encoded. For example, a string containing only the characters one and zero requires just 1 bit, whereas a string containing characters which can represent hexadecimal requires 4 bits per character when encoded. The default string type encodes in 7 bits. There is also an 8 bit string type which could be used to contain non-string data. As with integers, the various string types employ subtyping to optimise the encodings. This time the constraints are specified using the size syntax and are used to restrict the length of strings. Subsections 5.3 - 5.7 detail the mechanics involved.

4.3 Sequences

The sequence type provides a useful way of logically grouping together named values so that each belongs to a unique name space. Although it has no impact on the data encoded it is an important mechanism for structuring data. From an implementation specific point of view, structuring the data in this nested way means we could use simple association lists without suffering too much from the $O(n)$ time to search within a name space.

It may not be feasible to can encode every value within a sequence. To handle this optionality we must use a variation of the sequence type that informs the encoder to include the required extra information. Subsection 5.9 illustrates this principle. In addition to encoding a small amount of extra data, the added flexibility of optionality will result in a loss of performance as we need to determine what is present in the sequence at runtime.

Another important feature of sequences is they may repeat. From a low-level encoding point of view this is straight forward. All we need to know is how many times the sequence repeats. From a higher-level perspective we must employ special handling of our values to group together each individual sequence. In subsection 3.2 we call this a special form. Here we show how each normal form is surrounded in an extra pair of brackets to denote this grouping. Subsection 5.10 provides an explanation through example.

4.4 Choice

Typically a network protocol will consist of a selection of different messages or Protocol Data Units (PDUs). The simple protocol in subsection 1.2 provides an example of this type of choice. In addition, within a PDU itself decisions may need to be made that selectively encode only parts of the message.

4.5 Enumeration

Enumeration is common in many high-level languages. The DSL used by Packedobjects restricts the sequence of values to be of type symbol or integer. An example of an enumeration of symbols is illustrated in subsection 5.12.

4.6 Boolean and null

Boolean and null types are both simple to use. A boolean type encodes a true or false value and a null type encodes no value as depicted in sections 5.13 and 5.14 respectively. Although a null type encodes no value its significance comes from its context. There are specific circumstances where no extra data needs to be encoded to convey information. An analogous scenario would be the acknowledgement system employed by the Transmission Control Protocol (TCP). It is possible that a TCP acknowledgement is explicitly sent across a network where no actual TCP data is communicated other than the TCP header itself. With our DSL we could have a choice of PDUs and one particular choice represents an acknowledgement. By making this choice we already have all the information we need, however, a choice still requires encoding

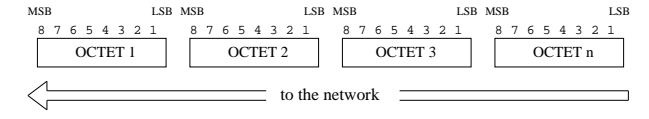


Figure 2. Order of bits and octets

at least one type. If the null type is used no extra data needs to be encoded. Section 7 provides a more esoteric example of using the null type.

5. The encoding process

The goal of the encoding process is to produce a tightly packed byte stream by combining a protocol and data. Thus protocol + data \rightarrow encoded data. Bit manipulation however, is sometimes viewed as the practice of hackers [16]. In this section we will describe the techniques used by Packedobjects to pack bits, starting from a high level perspective and working down to the lower-level integer encoder.

5.1 Higher layer encoding

All data types are expressed within a flat list structure suitable to be mapped to the low-level encoder. Each list item represents an individual call to the encoder with a form summarised as

```
(integer (range x y) n)
```

where x and y restrict the range of values n may take. We can then determine whether we need to encode signed or unsigned values and how many bits are required to encode this range. The resulting core form can be expressed as

```
(signed | unsigned (bits z) n)
```

where we show a choice between the signed or unsigned representation and z which represents the number of bits required to encode value n . This core form contains all the information we need to map to our low-level encoder. Before describing the low-level encoding process we will describe how each data type is converted to these core forms.

Some types require a length encoding, one which has been optimised to minimise the number of bytes required. Depending on the size of the value to be encoded, the first two bits of the encoding indicate whether 1 byte, 2 bytes or 4 bytes are required to represent the length. Figure 2 shows the order in which we encode our bits. The left-most bit in an octet is the Most Significant Bit (MSB). This byte order follows the “Big Endian” format. Examples of each encoding method given will refer to bits within an octet using this format.

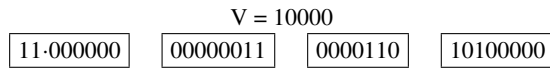
Returning to our length encoding, values less than or equal to 127 can be encoded as a non negative binary integer using bits 7 through to 1, and bit 8 is set to zero. For example to encode a length with a value of 7 we use

```
V = 7
0-0000111
```

If the value is greater than 127 and less than 16384, the value is encoded (using 2 octets) as a non negative binary integer starting from bit 6 of the first octet. The first 2 bits (bit 8 and bit 7) are set to equal 1 and 0 respectively. So to encode the value 1696 we get

```
V = 1696
10-000110 10100000
```

If the value is greater than or equal to 16384, then the first 2 bits are both set to equal 1. This leaves the remaining 30 bits to encode a value. To encode a value of 100,000 we get



Therefore length encodings are restricted to values less than 2^{30} . In practical terms this restriction should not be encountered and would indicate the data is not highly structured. Packedobjects has not been designed to handle large chunks of unstructured data.

Having introduced how length values are encoded we will now describe how to encode integers. The integer data type provides the basis for encoding all other types. In the subsections that follow we will show this relationship as we describe the remaining types.

5.2 integer

As previously mentioned in section 4.1, all integer types are expressed within a range. From this we decide the kind of integer to encode. Integers may be signed or unsigned as well as unconstrained, semi-constrained and constrained categorised as follows

signed	unconstrained	semi-constrained	constrained
unsigned	...	semi-constrained	constrained

All unconstrained and semi-constrained integers require a length encoding for the number of bytes required to represent the value. For example, the protocol

```
(foo integer ())
```

is combined with the value

```
(foo 1696)
```

to produce

```
(integer () 1696)
```

This example has no range therefore it is unconstrained. It is transformed into a list containing the following

```
((unsigned (bits 8) 2)
 (signed (bits 16) 1696))
```

The first item in the list represents the number of bytes required to encode the value. As with all integer length encodings, it is a small value and therefore encoded within a single byte. The second list item specifies the value 1696 to be encoded within 16 bits or 2 bytes.

Encoding a semi-constrained integer follows a similar approach except that a lower bound reduces the size of value we need to encode. For example, the protocol

```
(foo integer (range 1650 max))
```

is combined with the value

```
(foo 1696)
```

to produce

```
(integer (range 1650 max) 1696)
```

This example has a lower bound at 1650. Instead of encoding the value 1696 we encode $1696 - 1650$ as follows:

```
((unsigned (bits 8) 1)
 (unsigned (bits 8) 46))
```

Note the value 46 is encoded as an unsigned integer due to the positive lower bound.

Encoding a constrained integer is the most efficient way of encoding this data type. It bypasses the need to provide a length encoding. We determine the number of bits required to encode the value reduced by the lower bound and based on the supplied range. For example, the protocol

```
(foo integer (range 0 2000))
```

is combined with the value

```
(foo 1696)
```

to produce

```
(integer (range 0 2000) 1696)
```

This is transformed into

```
((unsigned (bits 11) 1696))
```

In this case the lower bound is 0 and therefore has no effect on the value encoded.

5.3 string

Depending on size constraints all strings must be semi-constrained, constrained or fixed in length. Only fixed length strings do not require a length encoding. A semi-constrained string has no size constraint on the upper bound. For example, the protocol

```
(foo string ())
```

is combined with the value

```
(foo "foobar")
```

to produce

```
(string () "foobar")
```

There is no size constraint therefore a length encoding is required. We transform our string into integer form with the length plus the characters in decimal form as follows

```
((integer (range 0 max) 6)
 (integer (range 0 127) 102)
 (integer (range 0 127) 111)
 (integer (range 0 127) 111)
 (integer (range 0 127) 98)
 (integer (range 0 127) 97)
 (integer (range 0 127) 114))
```

From this form we can apply the same techniques described for integer encoding to obtain

```
((unsigned (bits 8) 1)
 (unsigned (bits 8) 6)
 (unsigned (bits 7) 102)
 (unsigned (bits 7) 111)
 (unsigned (bits 7) 111)
 (unsigned (bits 7) 98)
 (unsigned (bits 7) 97)
 (unsigned (bits 7) 114))
```

A constrained string has a more concise length encoding. For example, the protocol

```
(foo string (size 1 10))
```

is combined with the value

```
(foo "foobar")
```

to produce

```
(string (size 1 10) "foobar")
```

From this we obtain

```
((integer (range 1 10) 6)
 (integer (range 0 127) 102)
 (integer (range 0 127) 111)
 (integer (range 0 127) 111)
 (integer (range 0 127) 98)
 (integer (range 0 127) 97)
 (integer (range 0 127) 114))
```

This in turn produces

```
((unsigned (bits 4) 5)
 (unsigned (bits 7) 102)
 (unsigned (bits 7) 111)
 (unsigned (bits 7) 111)
 (unsigned (bits 7) 98)
 (unsigned (bits 7) 97)
 (unsigned (bits 7) 114))
```

Note in this example there is a lower bound range of 1 which means the encoded length value is reduced to 5.

A fixed length string is the most concise as it requires no length encoding. For example, the protocol

```
(foo string (size 6))
```

is combined with the value

```
(foo "foobar")
```

to produce

```
(string (size 6) "foobar")
```

The transformation to core form requires no length encoding and only contains values for each character as follows

```
((unsigned (bits 7) 102)
 (unsigned (bits 7) 111)
 (unsigned (bits 7) 111)
 (unsigned (bits 7) 98)
 (unsigned (bits 7) 97)
 (unsigned (bits 7) 114))
```

5.4 bit-string

A bit-string encoding follows the same techniques as a string encoding, however, each character can be encoded within 1 bit. For example, the protocol

```
(foo bit-string (size 1 10))
```

is combined with the value

```
(foo "101010")
```

to produce

```
(bit-string (size 1 10) "101010")
```

This is then mapped to

```
((integer (range 1 10) 6)
 (integer (range 0 1) 1)
 (integer (range 0 1) 0)
 (integer (range 0 1) 1)
 (integer (range 0 1) 0)
 (integer (range 0 1) 1)
 (integer (range 0 1) 0))
```

Note how we must convert the character version of one and zero to its numeric equivalent. From this form we simply apply the same transformation techniques as previously described to obtain

```
((unsigned (bits 4) 5)
 (unsigned (bits 1) 1)
 (unsigned (bits 1) 0)
 (unsigned (bits 1) 1)
 (unsigned (bits 1) 0)
 (unsigned (bits 1) 1)
 (unsigned (bits 1) 0))
```

5.5 octet-string

An octet-string encoding is exactly the same as a string encoding but instead of using 7 bits we use 8 bits. For example, the protocol

```
(foo octet-string (size 6))
```

is combined with the value

```
(foo "foobar")
```

to produce

```
(octet-string (size 6) "foobar")
```

Represented in integer form we get

```
((integer (range 0 255) 102)
 (integer (range 0 255) 111)
 (integer (range 0 255) 111)
 (integer (range 0 255) 98)
 (integer (range 0 255) 97)
 (integer (range 0 255) 114))
```

From this we arrive at the core form

```
((unsigned (bits 8) 102)
 (unsigned (bits 8) 111)
 (unsigned (bits 8) 111)
 (unsigned (bits 8) 98)
 (unsigned (bits 8) 97)
 (unsigned (bits 8) 114))
```

5.6 hex-string

A hex-string encoding follows a very similar method to a bit-string encoding except that we have a larger range of values to encode corresponding to the valid characters of hexadecimal. For example, the protocol

```
(foo hex-string (size 1 10))
```

is combined with the value

```
(foo "AFAFAF")
```

to produce

```
(hex-string (size 1 10) "AFAFAF")
```

Represented in integer form we get

```
((integer (range 1 10) 6)
 (integer (range 0 15) 10)
 (integer (range 0 15) 15)
 (integer (range 0 15) 10)
 (integer (range 0 15) 15)
 (integer (range 0 15) 10)
 (integer (range 0 15) 15))
```

Again, we apply the same methods previously described to arrive at the following core form

```
((unsigned (bits 4) 5)
 (unsigned (bits 4) 10)
 (unsigned (bits 4) 15)
 (unsigned (bits 4) 10)
 (unsigned (bits 4) 15)
 (unsigned (bits 4) 10)
 (unsigned (bits 4) 15))
```

5.7 numeric-string

The encoding of a numeric-string follows the same principles previously described and transforms into exactly the same core form as a hex-string.

5.8 sequence

A sequence has no impact on the encoding output and therefore does not map to a core form.

5.9 sequence-optional

A sequence-optional encoding requires an addition value to be encoded to represent which items of the sequence have been supplied. For example, if we have the following protocol

```
(foobar sequence-optional
 (foo boolean)
 (bar boolean)
 (baz boolean))
```

and supply

```
(foobar
 (foo #t)
 (baz #t))
```

we obtain

```
((integer (range 0 7) 5)
 (integer (range 0 1) 1)
 (integer (range 0 1) 1))
```

The first item of the list encodes the value 5 as a constrained integer to represent the bitmap 101 which informs us that the second item in the sequence was not supplied. The integers are then mapped to the core form

```
((unsigned (bits 3) 5)
 (unsigned (bits 1) 1)
 (unsigned (bits 1) 1))
```

This example also used the boolean type which is explained in subsection 5.13.

5.10 sequence-of

A sequence-of encoding requires a value to represent how many times the sequence repeats. This value is encoded as a semi-constrained integer. For example, if we have the following protocol

```
(foobar sequence-of
 (foo boolean)
 (bar boolean))
```

and supply

```
(foobar
 ((foo #t)
 (bar #t))
 ((foo #f)
 (bar #f))
 ((foo #t)
 (bar #f)))
```

we obtain

```
((integer (range 0 max) 3)
 (integer (range 0 1) 1)
 (integer (range 0 1) 1)
 (integer (range 0 1) 0)
 (integer (range 0 1) 0)
 (integer (range 0 1) 1)
 (integer (range 0 1) 0))
```

Note how the first item informs us that the sequence repeats 3 times and is encoded without an upper bound. Applying the usual integer transformation techniques we end up with

```
((unsigned (bits 8) 1)
 (unsigned (bits 8) 3)
 (unsigned (bits 1) 1)
 (unsigned (bits 1) 1)
 (unsigned (bits 1) 0)
 (unsigned (bits 1) 0)
 (unsigned (bits 1) 1)
 (unsigned (bits 1) 0))
```

The first two items represent a semi-constrained integer encoding of the value 3. The remaining items represent the boolean values repeatedly encoded as part of the sequence.

5.11 choice

A choice encoding requires an index value to be encoded corresponding to the position of the chosen item in the sequence. For example, if we have the following protocol

```
(foobar choice
 (foo boolean)
 (bar boolean))
```

and we supply

```
(foobar
 (bar #f))
```

we obtain

```
((integer (range 1 2) 2))
 (integer (range 0 1) 0))
```

The first item in the list indicates the second choice was made in the sequence. The choice index is encoded as a constrained integer as described in subsection 5.2 and then mapped to a core form as follows

```
((unsigned (bits 1) 1)
 (unsigned (bits 1) 0))
```

5.12 enumerated

An enumerated encoding is very similar to a choice encoding, however we count the first item from 0. For example, if we have the following protocol

```
(foobar enumerated
 (foo bar baz))
```

and supply

```
(foobar bar)
```

we obtain

```
((integer (range 0 2) 1))
```

From this we can easily obtain the core form

```
((unsigned (bits 2) 1))
```

5.13 boolean

A boolean encoding concisely maps to 1 bit. For example, if we have the following protocol

```
(foo boolean)
```

and supply

```
(foo #f)
```

we obtain

```
((integer (range 0 1) 0))
```

From this we arrive at the core form

```
((unsigned (bits 1) 0))
```

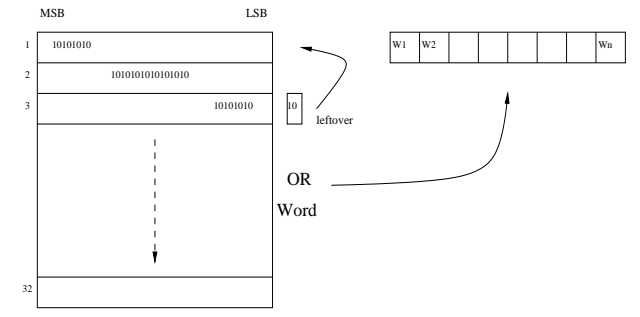


Figure 3. Encode buffers

5.14 null

A null encoding does not require any call to the encoder.

5.15 Lower layer encoding

Having shown how we represent everything through integer form we will now go on and explain our low-level integer encoder. The encoder works using two buffers [11]. One is fixed in size corresponding to the number of bits within a word; the other can be dynamically allocated. The word size is determined by the hardware platform and equals 32 bits in the example given. The fixed buffer can be visualised as an array of 32 words as depicted in figure 3. The dynamic buffer is typically created to accommodate the largest PDU so it effectively operates as a statically allocated piece of memory. The fixed buffer is used to construct the bit sequences before they are copied across to the dynamic buffer. A bit sequence is copied to the appropriate word of the fixed buffer and then shifted into position. The bits are aligned so that after an OR operation on the array of words a single word is produced which can then be copied across to the dynamic buffer. This sequence is illustrated in figure 3. To begin with the bit pattern "10101010" is copied to the first word in the fixed buffer. The eight bit pattern must be shifted so that it follows the network byte order and therefore has its most significant bit (MSB) at bit position 32. The next bit pattern "1010101010101010" is added to the second word and shifted so that its MSB starts at bit 24. This leaves room for only eight more bits to be added within the third word. If, for example, the ten bit pattern "1010101010" is to be added, then the eight most significant bits would be copied to the remaining room in the fixed buffer. The entire fixed buffer then has its contents OR'ed and the resulting word is copied to the dynamic buffer. The two bits left over are put back into the fixed buffer starting from the MSB of the first word. The pseudo code for the encode algorithm is provided in figure 4. The algorithm makes only two tests to see whether a word boundary is crossed in the fixed buffer and whether a full word exists already. The algorithm is recursive. It calls itself whenever there is a value left over to encode after a full word has been copied to the dynamic buffer.

6. The decoding process

The high level view of decoding can be summarised as protocol + encoded data → data, where traversal of the protocol acts as the driving force behind the process. Calls to the low-level decoder request the number of bits to use and whether or not the integer will be signed or unsigned. Similar to the encoding process we generate a series of calls to the low-level decoder within a list. Unlike the encoding process, which can use a flat list structure, the decoding process uses a list which emulates the protocol's structure. Additional information is also required to reconstruct the id of the value. For example, given the protocol

```

BEGIN /* encode */

    accept a number and a bit length

    IF the bit pattern is unable to fit into the space
    available in the current word of the fixed buffer THEN
        fit as many bits in as possible
        OR the fixed buffer
        copy the result to the dynamic buffer
        reset the fixed buffer
        GOTO BEGIN to encode the leftover bit pattern
    RETURN
ENDIF

    copy the number to the correct position in the
    next word of the fixed buffer

    IF we have a full word already THEN
        OR the fixed buffer
        copy the result to the dynamic buffer
        reset the fixed buffer
    ENDIF

END /* encode */

```

Figure 4. Encode algorithm

```

(foo integer (range 1 100))

we need to combine the id "foo" with the result of mapping

((integer (range 1 100))

to the low-level decoder. Using a similar method as described for
the encoder we transform this integer representation into its core
form

((unsigned (bits 7))

Assuming the result of this call to the decoder produces the value
10, we need to combine this back with its id to achieve the end
result of

(foo 10)

```

All data types are mapped through this process where they are first represented in integer form before being transformed into the core form required by the low-level decoder. To save on repetition we will not describe the process for all data types, as we did for encoding data, but rather go on to illustrate how the low-level decoder works.

6.1 Lower layer process

The decoder algorithm, depicted in figure 6, is slightly more straight forward than the encoder algorithm. A PDU is decoded by masking off the desired bits to form a value. The size of a word determines the size of the window which is placed over the data to decode. The window moves in word sized increments over the PDU. Provisions must be made to handle bit patterns that cross over word boundaries. Values obtained from different words must be merged together to form a single bit pattern. Figure 5 shows that the area between two words can contain the desired bit pattern. As with the encode algorithm, just two test conditions exist: one to examine whether a new word should be fetched from the PDU buffer and one to examine if the value to extract lies between two word boundaries. By storing a copy of the last word obtained, it may be possible to avoid fetching a word each time the routine is called.

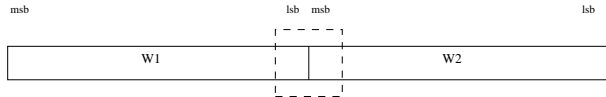


Figure 5. Decode window

```

BEGIN /* decode */

  accept a bit length

  IF current bit position has reached a word boundary THEN
    fetch a word from the buffer
    store a copy of the word
  ENDIF

  mask out (AND) the bit pattern from current word

  IF bit pattern crosses word boundary THEN
    fetch the next word from the buffer
    store a copy of the word
    obtain the missing part of the bit pattern
    merge (OR) the two bit patterns together
  ENDIF

  return the result

END */decode */

```

Figure 6. Decode algorithm

7. Functional abstraction

Although it is important to explain the integer encoding and decoding methods used by Packedobjects, we should not lose sight of the tool’s philosophy on abstraction [13]. As briefly mentioned in section 2, using an abstract language to describe the way data is packed into bytes allows us to think at a level that is completely disconnected from the mechanics of the underlying process. To illustrate this we will describe a real life activity such as shopping for food and drinks and encode our representation into bytes. As a network protocol this example is contrived, however, it does illustrate how much flexibility we have using our s-expression based DSL. In particular it will highlight powerful inbuilt data manipulation techniques such as quasiquote, unquote and unquote-splicing available as part of the Scheme language.

```

(define booze
  '(sequence-of
    (beer null)
    (nibbles null)))

```

We start by defining a sequence-of type containing null types. The sequence-of type is a compound data type which consists of a repeating sequence of other data types, in this case a sequence of two null types. The null type is one of several atomic data types available. It is an unusual data type in that it requires no value.

```

(define grub
  '(sequence
    (pizza null)
    (salad null)))

```

In addition to our drinks we should have some food. In this case we use the sequence data type which specifies both pizza and salad.

```

(define trolley
  '(trolley sequence-optional
    (food ,@grub)
    (drink ,@booze)))

```

```

(define basket
  '(basket choice
    (food ,@grub)
    (drink ,@booze)))

```

To carry our food and drink we could use a trolley or use a basket. The trolley is large enough to carry both but we must choose between the food or drink if we use a basket. The sequence-optional data type is a flexible type which is similar to the sequence type but each member is optional. Therefore, using the trolley we could decide to only pack some food. Using the basket we must choose between either the food or drink. The choice data type is a compound data type which enforces this restriction. In both cases, we have used unquote-splicing to reuse our definitions of food and drink and therefore are able to produce concise protocol descriptions. Having defined a protocol we must specify values according to their description.

```

(define thirsty
  '(basket
    (drink
      ((beer) (nibbles))
      ((beer) (nibbles))
      ((beer) (nibbles))
      ((beer) (nibbles)))))

```

```

(define hungry
  '(basket
    (food
      (pizza)
      (salad))))

```

```

(define thirsty+hungry
  '(trolley
    ,(cadr hungry)
    ,(cadr thirsty)))

```

Depending on our mood, we might be thirsty, hungry or both. In the case of being both thirsty and hungry we will use a trolley. This example illustrates the use of unquote to reuse our definitions of being hungry and thirsty. Note how we apply *cadr* to symbolise removing the basket from the items so we can place them in the trolley instead. The output of encoding our data is a tightly packed byte stream. In this case just two bytes are required to represent

```

(trolley
  (food (pizza) (salad))
  (drink ((beer) (nibbles))
         ((beer) (nibbles))
         ((beer) (nibbles))
         ((beer) (nibbles))))

```

8. Conclusion

The designer of a network protocol must make a number of choices. The choices taken will have an impact on the size and structure of the data communicated. In some cases it is necessary to try and encode the data as efficiently as possible, in which case a binary format may be used. Similar to the way we might migrate from a low-level language and think about a problem in a high-level language, the protocol designer should not think in terms of a low-level binary format. Instead the designer should use a more expressive alternative, one that will still produce equivalent concise binary output. In this paper we presented Packedobjects, a tool which provides such an alternative where developers are allowed to express their protocol using an abstract syntax. As with similar tools, an application produced takes the form of a compiled binary. However, with Packedobjects a Scheme interpreter is embedded into the application to provide the ability to represent a network protocol and its

values using an s-expression. By exploiting the concept of "data is code" we eliminate the need for using a compiler to transfer the abstract syntax into a concrete syntax which is usable in the native programming language. The benefits we gain from this approach are amplified in the tool's target application area of embedded systems. We are able to script the network protocol on an embedded device without the extra complication typically present when cross compilation is required. In addition, the separation of the data encoding and decoding process away from the compiled application facilitates extensibility of the communication. This in turn provides the opportunity to maintain communication across mass-deployed devices even when a change in protocol occurs. Such flexibility can be a key goal for embedded communication technologies. The main disadvantage of taking such a dynamic approach is the negative impact on performance that might occur, the significance of which depends on the nature of both the hardware and network used.

References

- [1] J. Dean and S. Ghemawat. MapReduce: a flexible data processing tool. *Commun. ACM*, 53(1):72–77, 2010. ISSN 0001-0782. doi: <http://doi.acm.org/10.1145/1629175.1629198>.
- [2] O. Dubuisson. *ASN. 1 Communication between Heterogeneous Systems*. Morgan Kaufmann, 2001.
- [3] Google. Protocol Buffers. <http://code.google.com/p/protobuf/>, July 2007. URL <http://code.google.com/p/protobuf/>.
- [4] P. Gustafsson and K. F. Sagonas. Bit-level binaries and generalized comprehensions in Erlang. In K. F. Sagonas and J. Armstrong, editors, *Erlang Workshop*, pages 1–8. ACM, 2005. ISBN 1-59593-066-3. URL <http://doi.acm.org/10.1145/1088361.1088363>.
- [5] International Telecommunication Union. Specification of Abstract Syntax Notation One (ASN.1). ITU-T Recommendation X.208, 1988.
- [6] International Telecommunication Union. Abstract Syntax Notation One (ASN.1): Constraint Specification. ITU-T Recommendation X.682, July 2002.
- [7] International Telecommunication Union. Abstract Syntax Notation One (ASN.1): Specification of Packed Encoding Rules (PER). ITU-T Recommendation X.691, July 2002.
- [8] P. Isensee. *Bit Packing: A Network Compression Technique*, chapter chapter 6, pages 571–578. Games Programming Gems 4. Charles River Media, 2004.
- [9] L. Kleinrock. The Latency/Bandwidth Tradeoff in Gigabit Networks. *IEEE Communications Magazine*, 30(4):36–40, Apr. 1992.
- [10] J. Larmouth. *ASN.1 Complete*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2000. ISBN 0-12-233435-3.
- [11] J. Moore. *On the Performance of Unaligned Packed Encoding Rules when Applied to a Non-optimised Protocol Specification*. PhD thesis, University of Sheffield, 2001.
- [12] J. P. T. Moore. Thumbtribes: Low Bandwidth, Location-Aware Communication. In M. S. Obaidat, V. P. Lecha, and R. F. S. Caldeirinha, editors, *WINSYS*, pages 197–202. INSTICC Press, 2007. ISBN 978-989-81111-14-2.
- [13] J. P. T. Moore. Get stuffed: Tightly packed abstract protocols in Scheme. The 10th Scheme and Functional Programming Workshop, 2009.
- [14] OSI. Information Technology – Open Systems Interconnection – Basic Reference Model. DIS 7498-1, ISO/IEC, 1984. ITU-T Rec. X.200.
- [15] A. Tanenbaum. *Computer Networks*. Prentice hall PTR, 2002.
- [16] H. Warren. *Hacker's Delight*. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 2002.