Everything counts in small amounts

John P. T. Moore,
Thames Valley University, UK,
moorejo@tvu.ac.uk

Abstract

This paper describes an encoding tool which utilises the "data is code" principle of symbolic expressions available in Lisp-like languages to allow the scripting of tightly-packed, cross-platform network protocols. 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 without requiring special handling.

1 Introduction

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 [1], an alternative more abstract approach is often 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 and involves code generation. 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. This is the approach of Abstract Syntax Notation One (ASN.1) [2] which originates from the world of telecommunications. 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. Google's Protocol Buffers[1] 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.

In this paper we describe Packedobjects [7], a tool which takes a more dynamic approach. Packedobjects uses s-expressions from the Scheme programming language to exploit the concept of "data is code" and therefore bypasses the need for code generation. 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).

A novel aspect of the tool originates from its foundations as an extension language. In the following sections we will introduce this concept and also explain how this supports a feature such as extensibility. The remaining sections of the paper will describe the language used by Packedobjects as well as illustrate the encoding process. Finally we will present a simple example, describe some challenges and then conclude.

2 Extension language

Packedobjects is available[2] as a module for GNU Guile which in turn is available[3] as a C library. 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

[1] http://code.google.com/apis/protocolbuffers/
[2] http://gitorious.org/packedobjects/
[3] http://www.gnu.org/software/guile/

Proceedings of SIMPAR 2010 Workshops
Intl. Conf. on SIMULATION, MODELING and PROGRAMMING for AUTONOMOUS ROBOTS
Darmstadt (Germany) November 15-16, 2010
ISBN 978-3-00-032863-3
pp. 278-283

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 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.

3 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)
     (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 [3]. 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 novel aspects of the tool we will now describe the language used.

4 Integer Encoding Rules

The domain specifc language (DSL)[4] 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 process of encoding types involves combining a protocol and some data and transforming this into an integer form. The integer form is then transformed into a core form before being supplied to the low-level encoder [6]. In the following subsections we will describe the transformation which we call Integer Encoding Rules. The integer form can be summarised as follows

```
(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

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

where we show a choice between the signed and unsigned representation and z which represents the number of bits required to encode value n.

Integers may be unconstrained, semi-constrained or constrained. All unconstrained and semi-constrained integers require a length encoding to represent the number of bytes required to encode the value. In the following subsections we will first describe length encoding and then show by example how other types are handled.

[4] http://zedstar.org/packedobjects/#Protocol-grammar

Proceedings of SIMPAR 2010 Workshops
Intl. Conf. on SIMULATION, MODELING and PROGRAMMING for AUTONOMOUS ROBOTS
Darmstadt (Germany) November 15-16, 2010
ISBN 978-3-00-032863-3
pp. 278-283

4.1 Length encoding

Values are encoded within 8, 16 or 32 bit ranges. This choice of 3 sizes can be represented with 2 bits using the following function:

```
(lambda (n)
  (cond
   ((and (>= n SCHAR_MIN) (<= n SCHAR_MAX))
    `(unsigned (bits 2) 0))
   ((and (>= n SHRT_MIN) (<= SHRT_MAX))
    `(unsigned (bits 2) 1))
   ((and (>= n INT_MIN) (<= INT_MAX))
    `(unsigned (bits 2) 2))))
```

Currently Packedobjects supports 32 bit architectures however, the 2 bits used to encode the length category can support an additional value for 64 bit platforms.

4.2 Unconstrained 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. 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. Given the following protocol

```
(foo integer (range min max))
```

and corresponding data

```
(foo 1066)
```

we combine the protocol and data together to form

```
(integer (range min max) 1066)
```

As this example has no range limits it is unconstrained and will need to be encoded as a signed value. It is transformed into a list containing the following:

```
((unsigned (bits 2) 1)
 (signed (bits 16) 1066))
```

The first item in the list specifies which length category the value will be encoded in. The second item in the list is the value encoded in the number of bits dictated by this category.

4.3 Semi-constrained integers

Encoding a semi-constrained integer follows a similar approach to an unconstrained integer except that a lower bound restricts the size of the value we encode. The offset result will always be positive and is therefore encoded as an unsigned variant. For example, the protocol

```
(foo integer (range -1000 max))
```

is combined with the value

```
(foo -1000)
```

to produce

```
(integer (range -1000 max) -1000)
```

Instead of encoding the value -1000 we first subtract the lower bound. This allows us to encode the result as a positive value as follows:

```
((unsigned (bits 2) 0)
 (unsigned (bits 8) 0))
```

4.4 Constrained integers

Encoding a constrained integer bypasses the need to provide a length encoding. We determine the number of bits required to encode the value based on the range. For example the protocol

```
(foo integer (range -100 100))
```

is combined with the value

```
(foo 100)
```

to produce

```
(integer (range -100 100) 100)
```

This is transformed into

```
((unsigned (bits 8) 200))
```

As with semi-constrained integers, the lower-bound means we will always encode positive values.

4.5 String types

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. Rather than show how every string type is first converted to integer form and then to core form, we will provide one example to illustrate the principles involved. For example, the protocol

Proceedings of SIMPAR 2010 Workshops
Intl. Conf. on SIMULATION, MODELING and PROGRAMMING for AUTONOMOUS ROBOTS
Darmstadt (Germany) November 15-16, 2010
ISBN 978-3-00-032863-3
pp. 278-283

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

is combined with the value

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

to produce

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

This is then transformed into integer form where the first item in the list represents the length encoding followed by each character converted to a decimal value.

```
((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))
```

The length is encoded as a constrained integer as specified in section 4.4. From this form we convert to core form as follows

```
((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))
```

The end result of supplying this list to the low-level encoder is a string containing just 2 bytes.

4.6 Enumerated types

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 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))
```

4.7 Boolean types

A boolean type encodes a true or false value and 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))
```

4.8 Null types

A null type encodes no value and therefore does not require any call to the encoder. 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.

4.9 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.

4.10 Sequences with optionality

It may not be feasible to 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. 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. For example, if we have the following protocol

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

and supply

Proceedings of SIMPAR 2010 Workshops
Intl. Conf. on SIMULATION, MODELING and PROGRAMMING for AUTONOMOUS ROBOTS
Darmstadt (Germany) November 15-16, 2010
ISBN 978-3-00-032863-3
pp. 278-283

```
(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 4.7.

4.11 Sequences that repeat

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 the example that follows we can see how each sequence of values foo and bar are surrounded by an extra pair of brackets to denote this grouping. Encoding this sequence requires a value to represent how many times it repeats. This value is encoded as a semi-constrained integer as illustrated in subsection 4.3. 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.

4.12 Making choices

Typically a network protocol will consist of a selection of different messages or Protocol Data Units (PDUs). The simple protocol in section 3 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. 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 4.4 and then mapped to a core form as follows

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

The following section will provide a complete example of the encode and decode process.

5 Example

We will describe a simple phone book entry by first defining the protocol and then encoding and decoding the values within the Scheme interpreter.

Proceedings of SIMPAR 2010 Workshops
Intl. Conf. on SIMULATION, MODELING and PROGRAMMING for AUTONOMOUS ROBOTS
Darmstadt (Germany) November 15-16, 2010
ISBN 978-3-00-032863-3
pp. 278-283

```
(use-modules (packedobjects packedobjects))
(use-modules (ice-9 pretty-print))

(define protocol
  '(person sequence-optional
     (name string (size 1 100))
     (id integer (range 1 max))
     (email string (size 3 320))
     (phone-number sequence-of
       (number numeric-string (size 8 20))
       (type enumerated (mobile home work)))))

(define data
  '(person
    (name "John Doe")
    (id 1234)
    (email "johnd@example.com")))
```

From the interpreter we can try out our code

```
guile> (define pdusize 100)
guile> (define pdu (encode protocol data pdusize))
guile> (string-length pdu)
27
guile> (pretty-print (decode protocol pdu pdusize))
(person
  (name "John Doe")
  (id 1234)
  (email "johnd@example.com"))
guile>
```

This example shows the use of the sequence-optional type which allowed us to add an entry for John Doe without including a phone number. The reference manual [7] contains a variety of other examples. Before concluding we will highlight some challenges developing the tool.

6 Challenges

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 [4]. For example, Packedobjects has been tested successfully on various embedded Linux devices[5] 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 [5].

Another potential difficulty is providing sophisticated error and exception handling without directly impacting on performance. Scheme has a powerful macro system which could be used to map to the s-expression-based DSL. This could help provide a consistent approach to handling errors.

7 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.

[5] http://zedstar.org/packedobjects/#Embedded-Linux

References

[1] 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.

[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] L. Kleinrock. The Latency/Bandwidth Tradeoff in Gigabit Networks. *IEEE Communications Magazine*, 30(4):36–40, Apr. 1992.

[5] 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.

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

[7] J. P. T. Moore. Packedobjects Reference Manual. http://zedstar.org/packedobjects/, Aug. 2010.

Proceedings of SIMPAR 2010 Workshops
Intl. Conf. on SIMULATION, MODELING and PROGRAMMING for AUTONOMOUS ROBOTS
Darmstadt (Germany) November 15-16, 2010
ISBN 978-3-00-032863-3
pp. 278-283