PADS/Haskell
Sub-Byte Data Support

Sam Cowger

August 28, 2017

1 Booleans

PADS/Haskell now supports descriptions using bit-level Booleans ("BitBools"). They can be used in whatever circumstances may require them, but one common use would be to describe flags in a network packet. As an example, a description of flags in a TCP header (full specification here) would look like this:

```
data TCPFlags = TCPFlags {
    tcp_urg :: BitBool,
    tcp_ack :: BitBool,
    tcp_psh :: BitBool,
    tcp_rst :: BitBool,
    tcp_syn :: BitBool,
    tcp_fin :: BitBool
}
```

TCPFlags describes six bits in a row, whether they be in a single byte or across two bytes. Here, tcp_urg will be the "leftmost" bit of this set of six bits in data, while tcp_fin will be the "rightmost" bit. Put simply, order matters: the first bit you expect to encounter must be described first, and so on. This behavior has endianness ramifications, namely, the onus of determining endianness falls to the user; at this low of a level, significance of data cannot be inferred and endianness has no pertinence.

BitBools are represented in memory as Haskell Booleans, and the values they describe, once parsed from data, can be used in a Haskell program as such. If a program depended on a flag from a TCP header, it could be used as (e.g.) a predicate in an if statement with no additional syntactic gymnastics (see below, in 2, for an example of such a program).

1.1 Implementation Details

BitBools were my first step in extending PADS below the byte level, as the choices presented to the user in the interface (and thus cases necessary to account for in the implementation) were far fewer than for full support for numbers of arbitrarily long sequences of bits. The change which underpins them, and indeed the entire suite of sub-byte functionality of PADS, is an additional field in the Source record that represents the position of the next bit to be read. With this change, any source manipulation function could have access to both the current bit and the current byte containing it. Since PADS was constructed to operate on (read or consume) only as little as byte at a time, this was a reasonable and efficient choice within the existing infrastructure (details of how exactly it proves helpful can be found in 2.1, as they are more pertinent to my implementation of numbers).

2 Numbers

Support for numbers denoted by arbitrary amounts of bits came next. In the interests of emulating pure Haskell concrete syntax, i.e. in keeping with the PADS ethos, the types for these numbers are designed to mirror Haskell `Word`s of fixed bit widths. These new number types are `Bits8`, `Bits16`, `Bits32`, `Bits64`, and `BitField`. A `Bits8` can represent a number of up to 8 bits in data, `Bits16` can represent 16, and so on; a `BitField` can be used to represent a number of an arbitrarily large (i.e. likely >64) amount of bits, and is analogous to a Haskell `Integer` (itself an arbitrary precision number).

In a PADS description, each type takes as an argument the amount of bits by which the number in data is represented, i.e. `Bits16 13` would parse and return a number of 13 bits. Attempting to parse more bits than a specific `Bits` type can hold will trigger an error and the data will not be parsed. Since all types mirror `Word`s, all values they return are unsigned. Again, endianness is unaccounted for at this level; asking for 13 bits will return whatever value the next 13 bits in data represent. With these types, we now have the tools necessary to write a more complete description of a TCP packet header (however, this example does not define the fields for either options or data, as both require complexity not befitting this example):

```
data TCPPacket = TCPPacket {
    tcp_source :: Bits16 16,
    tcp_dest   :: Bits16 16,
    tcp_seqnum :: Bits32 32,
    tcp_acknum :: Bits32 32,
    tcp_offset :: Bits8 4,
    Bits8 6, -- reserved
    tcp_flags  :: TCPFlags,
    tcp_window :: Bits16 16,
    tcp_cksum  :: Bits16 16,
    tcp_urgptr :: Bits16 16
}
```

Notable in the above description is the sixth item, a type without a named variable. This acts as a literal, and will read and consume those six bits in the data without returning them to the user. This behavior is most compatible with the TCP header specification, which reserves these bits for future use but ascribes no current significance to them; from a user standpoint, it's easier and better to simply consume and ignore them than name and parse them.

Using a description similar to this, generating a list of all packets with the `rst` flag set and with an offset greater than five (i.e. with any options set) is simple, leveraging PADS's emulation of Haskell's record syntax:

```
getRstWithOpts :: [TCPPacket] -> [TCPPacket]
getRstWithOpts = filter (\p -> rst (tcp_flags p) && (tcp_offset p > 5))
```

2.1 Implementation Details

Excepting `BitField`s, which are `Integer`s, each type above is represented by the corresponding `Word` of a fixed amount of bits (`Bits8 == Word8, Bits16 == Word16,` etc.). As such, and as demonstrated in `getRstWithOpts`, each is manipulatable in Haskell as a regular value (of its corresponding type), similar to the behavior of `BitBool`s.

The functions underpinning these types all rely on the same pieces of information ((a) the significance of the current bit, (b) the amount of bits to take, and (c) the bytes, as a Haskell `ByteString`, representing the data to be parsed), and on the same basic principle. As an example, let's take 17 bits from three bytes worth of data, with values [117,25,203], having already read the most significant two bits of the first byte (in Source, the `bit` field is therefore 5)

- We begin with the raw binary data, with the bits we intend to take highlighted:

7	6	5	4	3	2	1	0		7	6	5	4	3	2	1	0		7	6	5	4	3	2	1	0
0	1	1	1	0	1	0	1		0	0	0	1	1	0	0	1		1	1	0	0	1	0	1	1

- Partition the data into (1) the bytes that contain the bits to be parsed, white, and (2) the "rest" of the data's bytes, gray (alternating gray and white in (2) here signifies that it needs to be read for some of its bits, but also needs to be retained so the rest of its bits remain accessible; this partial byte read behavior is common):

<													1												>
																		<			2				>
7	6	5	4	3	2	1	0		7	6	5	4	3	2	1	0		7	6	5	4	3	2	1	0
0	1	1	1	0	1	0	1		0	0	0	1	1	0	0	1		1	1	0	0	1	0	1	1

- Convert (1) into a number (my implementation uses a foldr) by treating it as a value in base 256 (e.g. these bytes with values of [117,25,203] would become $117*256^2+25*256^1+203*256^0$, or 7,674,315). In this particular case, the number will consist of all bits in this data:

7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0
0	1	1	1	0	1	0	1	0	0	0	1	1	0	0	1	1	1	0	0	1	0	1	1

- Perform a bitwise right shift on this value, such that the rightmost insignificant bits (arising from the partitioning stage) are eliminated:

7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0
0	1	1	1	0	1	0	1	0	0	0	1	1	0	0	1	1	1	0	0	1	0	1	1

↓

					7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5
0	0	0	0	0	0	1	1	1	0	1	0	1	0	0	0	1	1	0	0	1	1	1	0

- Perform a bitwise AND on the resulting value with a mask of ones, such that the leftmost insignificant bits are eliminated:

					7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5
0	0	0	0	0	0	1	1	1	0	1	0	1	0	0	0	1	1	0	0	1	1	1	0

(.&.)

0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

↓

							5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5
0	0	0	0	0	0	0	1	1	0	1	0	1	0	0	0	1	1	0	0	1	1	1	0

- This remaining value is what we want, and is returned. The final byte ((2), from the partitioning stage) is left accessible, and the `bit` value in Source is set to 4.

Reusing a single function for each `Bits` type would seem the most programmer-friendly and polymorphic option, but the efficiency trade-offs were far from ideal; one function to operate on *any* number of bits meant returning a Haskell `Integer`, an inefficient type, and casting the returned values to the `Word` of the proper width. The preferable approach was to write individual functions for each `Bits` type, which I did. `BitBools`, however, share the implementation of `Bits8`, and always use its functions to take a single bit.

3 Non-Byte-Aligned Data

As a corollary to the above modifications, I implemented a basic set of alignment-agnostic types, for describing data that may come heavily compressed and in a non-byte-aligned format. These "NB" types include `CharNB`, `BytesNB`, `StringNB`, `StringFWNB` (fixed width string), and `StringCNB` (string with a user-specified terminating character). If these types prove unhelpful for a specific application, they can be used as starting points for other alignment-agnostic extensions to the language. Their interfaces are identical to the regular types they emulate, and they function identically on data that might happen to be byte-aligned. A contrived example of their use:

```
data StrangeData = StrangeData {
    offset1   :: Bits8 3,
    nbchar    :: CharNB,
    nbstringc :: StringCNB '\n',
    nbbytes   :: BytesNB 10,
    offset2   :: Bits8 5
}
```

3.1 Implementation Details

These types (other than `CharNB`, which is guaranteed to be 8 bits) leverage the same functionality as for `BitFields`, and as such, they involve the same process as described in 2.1. This makes them far less efficient than their byte-aligned counterparts, but this is a necessary evil. Parsing non-byte-aligned data (using a parsing engine meant to operate on one byte at a time) will never be as fast as doing so with regular data, and the relative lack of prevalence of non-byte-aligned data makes it not worth the trouble to entirely overhaul PADS to improve its behavior here.