PADS/Haskell
Data Generation and Serialization

Sam Cowger

August 18, 2018

1 Generation

1.1 Implementation and Design Choices

Initial efforts at generation took the form of a standalone module that would convert PADS descriptions into a generation-friendly AST, then walk over that to produce a disk representation of the generated data. This approach had its limitations, such as relying on a PADS quasi-quotation being bound to a variable, eliminating automatic creation of parsing and printing functionality and forcing the user to choose between these or generation. Additionally, it was unable to handle the arbitrary Haskell expressions present in some PADS descriptions.

PADS relies on Template Haskell to use a PADS description to create a customized Haskell parsing function. Generation now takes the same approach, allowing for the flexibility and power that the earlier version lacked. Generated generation code, structurally speaking, mirrors generated parsing code. For example, we might have the description

```
data Foo = Foo { x :: Int, y :: Bytes <|x * 4|> }
```

to describe an (ASCII) `Int` followed by a `ByteString` whose length is four times the value of that `Int`. The following are the parser and generator that PADS will create for this description, with metadata manipulation code, uniqueness of generated names, and extra parentheses removed for readability:

Parser	**Generator**

`foo_parseM = let`
`  foo x1 x2 = ... metadata manipulation ...`
`  in do (x, x_md) <- int_parseM`
`        (y, y_md) <- bytes_parseM (x * 4)`
`        return (Foo x y, foo x_md y_md)`
	`foo_genM = do`
	`  x <- int_genM`
	`  y <- bytes_genM (x * 4)`
	`  return (Foo x y)`

The functions have clearly similar structures. Both easily incorporate evaluation of pure Haskell and the ability to refer to a previously-defined (and by extension previously-parsed or previously-generated) variable. By not dealing with the metadata that the parsers need to carry, generation is able to be more lightweight.

Much like parsing requires manually implemented parsers for base types (e.g. int_parseM), the new generation functionality requires the same, with a suite of new base generators created in CoreBaseTypes.hs.

1.2 User Interface

Rules for generator names mirror those for parsers: lowercase the first letter of the type and append "_genM" (if the type comes from a qualified import, e.g. `BE.Int8`, the generator name will be qualified as well, e.g. `BE.int8_genM`).

Generation procedures run in the "PadsGen" monad, a custom Reader monad carrying a random value generator, allowing reuse of the same generator throughout the procedure. As such, any call to a generator needs to be wrapped in the function `runPadsGen (:: PadsGen a -> IO a)`, which creates the random generator and supplies it to the procedure (e.g. `runPadsGen foo_genM`). Language.Pads.Padsc exports `runPadsGen`.

A type `Foo` will usually have a generator of type `PadsGen Foo`. However, for datatypes with type parameters (e.g. `data List a = ...`), the generator will expect generators as arguments (i.e. `list_genM :: PadsGen a -> PadsGen (List a)`) in order to generate data. These effectively allow instantiation of the datatype. Similarly, for types with expression parameters (e.g. `Bytes`), the generators expect expressions as arguments (i.e. `bytes_genM :: Int -> PadsGen Bytes`).

As part of the new generation functionality, users have the ability to override generation for individual fields of record types. For instance, the above example could become

```
data Bar = Bar { x :: Int generator <|randNumBound 100|>, y :: Bytes <|x * 4|> }
```

to lower the upper bound of generated values of `x` to 100. This presents a useful level of customization, for example, in generating network packets - the length field in a packet might accommodate numbers of up to 32 bits, but if many packets are fewer than 1,500 bytes in length then standard generation will result in unrealistically large data:

Unrealistic Generation	**More Realistic Generation**

`data Packet = Packet {`
`  len :: Word32,`
`  ... other fields ...`
`  body :: Bytes len`
`}`
	`data Packet = Packet {`
	`  len :: Word32 generator <|randNumBound 1500|>,`
	`  ... other fields ...`
	`  body :: Bytes len`
	`}`

Additionally, users can (generally must) provide their own generation functions for PADS declarations made using the `obtain` keyword. A user might have the description

```
type Hex = obtain Word from String using <|(h2w, w2h)|> generator word_genM
```

to describe a mapping between `Word`s and `String`s representing hex values. This example illustrates how sound `obtain` declarations can be created where default generation behavior is impossible to derive automatically.

The types `Hex` (a type synonym for `Word`) and `String` are not isomorphic, since logically any hex value can be converted to a string but not all strings represent valid hex. PADS would need to generate a `Word` to generate a `Hex`, but since `Word` is not a PADS base type, no `Word` generator already exists. (`Hex` can be a type synonym for it nonetheless because `Word` exists in the underlying Haskell context.) Here, then, the only way to generate data is for the user to provide their own generator, via the `generator` keyword. By default, generators for `obtain` declarations will trigger an error.

Users can also (once again optionally) provide custom generators for type synonyms, e.g. with the declaration `type Foo = Bar generator <|myFoo_genM|>`. The type of `myFoo_genM` ought to match whatever type `bar_genM` would have (in this case, `PadsGen Bar`).

A number of functions useful for user-defined generators (all of which run in the PadsGen monad), including the function `randNumBound` used above, are exported by the module Language.Pads.Generation (in the file Generation.hs), in turn exported by Language.Pads.Padsc. When users create their own generators, it is incumbent on them to ensure the type of their custom generator matches whatever type the automatically-derived version would have, since type signatures are not yet automatically created for default generators.

None of the changes made to add generation functionality result in a loss of backwards compatibility with preexisting PADS descriptions. Generation functions will be derived for such descriptions, but can be ignored. The additional `generator` keyword, the only change to the concrete syntax, is optional.

NB: While generating random values uniformly distributed across a default range might often result in unusable or unrealistic data, such data could still be valuable in fuzz testing, e.g. in applications that need to ensure correctness for any examples of a given format of data, not just realistic ones.

2 Serialization

2.1 Implementation and Design Choices

PADS has the ability to print in-memory representations of data through use of `_printFL` functions. These functions, however, are limited in their ability to convert sub-byte data to its original on-disk representation. New serialization functions deal in "`Chunks`" instead, a datatype representing either an ASCII character, or a value and the amount of significant bits of that value. Since each Chunk specifies (implicitly or explicitly) how many bits of its value are relevant, they can be used to easily describe and join together pieces of data that do not align to byte boundaries, such as flags of a network packet.

Parsing considers the bits in a byte from most to least significant (i.e. left to right, in the way numbers are usually written), when working with sub-byte data descriptions. For example, with P representing the "position" of the parser (i.e. the next bit the parser will consume):

	P							
0	1	1	0	1	1	0	1	...
2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0	...

⇒

					P			
0	1	1	0	1	1	0	1	...
2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0	...

This represents a snapshot of parsing behavior within a single byte, where the byte as a whole represents the decimal value 109. By consuming the shaded 4 bits, the parser returns the decimal value 13.

Serialization works in harmony, creating as output what the parser would expect as input. This means bytes are "packed" with bits beginning with the most significant. If one serialized decimal 1 as type `Bits8 1`, i.e. as a single bit of data, then (absent any other data being serialized before or after it) the result would be a byte with only the most significant bit activated, as a whole representing not 1 but 128. This behavior allows for, among other things, correct parsing and serialization of network packets, whose specifications conform to these same ideas of bit significance and arrangement.

In the same vein as parsing, serialization function structure closely mirrors that of the preexisting printer functions. The generated _printFL function and the new generated _serialize function for the previous Foo example:

Printing

```
foo_printFL (rep, md) = case (rep, md) of
  (Foo {x = x, y = y}, (_, Foo_imd {x_md = x_md, y_md = y_md}))
    -> concatFL [int_printFL (x, x_md), (bytes_printFL (x * 4)) (y, y_md)] }
```

Serialization

```
foo_serialize rep = case rep of
  Foo x y -> cConcat [int_serialize x, (bytes_serialize (x * 4)) y]
```

All serialization functions result not directly in `Chunk` lists but rather in `CList`s, an abstraction used to represent a `Chunk` list. The type of `CList` is `[Chunk] -> [Chunk]`. It's what some call a "difference list" or the "ShowS trick," which is generally speaking a representation of a list as a function awaiting an argument to append to itself (used in the implementation of `shows`).

`cConcat` (`:: [CList] -> CList`, used above) joins together multiple `CList`s into a single `CList` by folding over the list with `cAppend`. `cAppend` (`:: CList -> CList -> CList`) joins together two `CList`s into a single `CList` using function composition, where `(cAppend c1 c2)` == `(c1 . c2)`. Using these functions, `CList`s can be treated as regular Haskell lists, while avoiding the potentially $O(n^2)$ operation of appending a number of lists together in favor of the $O(n)$ operation of executing a composition of n functions.

The printer functions' inclusion of metadata theoretically allows for more robust printing functionality (e.g. manipulating the metadata from a parse and providing it in some format along with the printed result), but in practice the metadata is discarded when printing, and it is disregarded entirely when serializing. Serialization functions are meant to be a more strict translation from Haskell structures to on-disk representation; metadata wouldn't appear in such a translation.

2.2 User Interface

Similarly to generation functions, serialization functions for a particular type are named by lowercasing the first letter of the type and appending "_serialize". Any type's serialization function will convert it to a `CList`, an abstraction described above.

For some type `Baz`, the generated serialization function will likely have type `Baz -> CList`. However, similarly to generation, if a datatype has type parameters, then its serializer will expect serializers as arguments (e.g. `list_serialize :: (a -> CList) -> MyList a -> CList`). For datatypes with expression parameters, serializers will expect expressions as arguments (e.g. `bytes_serialize :: Int -> Bytes -> CList`).

The function `fromCL` (`:: CList -> [Chunk]`) converts from `CList` to `Chunk` list, and the function `fromChunks` (`:: [Chunk] -> ByteString`) converts from `Chunk` list to `ByteString`. (Serialization is a pure computation.) These functions, together with a particular "_serialize" function, work together to translate data from in-memory to on-disk representation.

For example, `(fromChunks . fromCL . baz_serialize) myBaz`, where `myBaz` is an instance of `Baz`, is a direct and point-free way to perform such a translation.