Best Title in the Universe

Victor Cacciari Miraldo

No Institute Given

Abstract. stuff

1 Introduction

The majority of version control systems handle patches in a non-structured way. They see a file as a list of lines that can be inserted, deleted or modified, with no regard to the semantics of that specific file. The immediate consequence of such design decision is that we, humans, have to solve a large number of conflicts that arise from, in fact, no conflicting edits. Implementing a tool that knows the semantics of any file we happen to need, however, is no simple task, specially given the plethora of file formats we see nowadays.

```
items      ,qty ,unit
wheat-flour,2   ,cp
eggs       ,2   ,units
```

Alice

```
items      ,qty ,unit
wheat-flour,1   ,cp
eggs       ,2   ,units
```

Original

```
items      ,qty ,unit
cake-flour ,1   ,cp
eggs       ,2   ,units
sugar      ,1   ,tsp
```

Bob

Fig. 1. CSV files

It is not hard to see that Alice's and Bob's edits, in figure 1 do *not* conflict. However, the diff3 [cite!] tool will flag them as such. Using generic programming techniques we can do a better job at identifying actual conflicts. The problem is twofold, however: (A) how to parse things generically and (B) how to diff over the results of these parsers and merge them properly.

We begin by explaining how we can write parsers in a very generic fashion, using type-level *grammar combinators*. This approach has lots of advantages, for instance, it is easily invertible to generate pretty-printers. The exposition of our library is guided by writing a CSV parser as an example.

Once we can solve the parsing problem in an elegant fashion, we address the diffing problem. Namelly, how can we take two values returned by a *grammar* parser and represent the differences between them.

To summarize our contributions,

- We present a generic parsing and pretty printing library built on Haskell's type-level, using a similar technique as in [1].
- We model the notion of *patch* generically, and prove it's correctness in Agda.

2 Grammar Combinators

Our parser combinator library seeks to generate a parser and a pretty printer from the same specification. It is similar to [3] and [2] in its idea, but drastically different in implementation. We achieve our generic parsers by encoding our combinators on the type-level, then having diffent instances of a class *HasParser* for each of them. We were inpired by [1], who first used this technique to build API-driven web servers.

Sticking to CSV as our *to-go* example, we can write its grammar in BNF as shown in figure 2, it can be read as *A CSV file consists in many lines.* From this description, it is already expected that a CSV parser will return a [*Line*], given an input file in the CSV format. But the converse should also be easy! Given a [*Line*], knowing that it represents a CSV file, we should be able to print it as such.

$$
\begin{aligned}
CSV &::= (line\ '\backslash n')^* \\
line &::= string \\
&\quad\mid\ string\ ','\ line
\end{aligned}
$$

Fig. 2. CSV grammar

The idea here lies in the fact that a BNF already looks like a Haskell type declaration, and at the same time, acts as the *type* of a language. We want to have type constructors that mimick the BNF syntax and allow us to define instances by induction on their structure. The CSV grammar gives us the following datatype.

```
type CSV  = [Line]
data Line = One  String
          | More String Line
```

Our parser combinators follow the applicative style, with some precedences swapped. With least precedense we have $a :\!<\!*\!> b$, which will parse a then b, in sequence, and return (*Result* a, *Result* b). Together with its forgetfull versions $a :*\!> b$ and $a :\!<\!* b$, wich will ignore the result from the left, or right, respectively. These type combinators are right associative. Then we have the choice combinator $a :\!<\!|\!> b$, which will try to parse a. If it fails, it proceeds by trying b, its result is *Either* (*Result* a) (*Result* b). And last we have a *tagging* combinator $k :\!<\!\$\!> a$ which injects the result of a into a datatype k. We will explain this combinator in more detail in section 2.1. We stress that a exhaustive description of the grammar combinators is beyond the scope of this paper. We refer the reader to the hackage documentation PUT IT ONLINE! for that.

A few high level combinators are also available, for example *VMany* a and *HMany* a. The parsing behaviour of both is the same, they will parse as many a s as possible. On the pretty-printing side, however, one will concat it's arguments vertically whereas the other will concat it's arguments horizontally.

Henceforth, the CSV grammar written using our combinators looks like figure 3.

Using `-XDataKinds` GHC extension we can have type-level strings, which are suitable for defining combinators such as *Sym* `","`, that parses the symbol `","`.

```
type CSVParser = VMany LineParser
type LineParser = Line :<$> Iden
                       :<|> Iden :<*> Sym "," :*> LineParser
csvParser :: Parser (Result CSVParser)
csvParser = genParser (Proxy :: Proxy CSVParser)
```

Fig. 3. CSV Parser

Here, *genParser* and *Result* are definitions provided by the class *HasParser*, which is defined for all grammar combinators,

```
class HasParser a where
  type Result a :: *
  genParser     :: Proxy a → Parser (Result a)
```

And, for illustration purposes, the instance for *VMany* is defined as:

```
instance (HasParser a) ⇒ HasParser (VMany a) where
  type Result (VMany a) = [Result a]
  genParser _ = many (genParser (Proxy :: Proxy a))
```

Note that we need to keep using these proxies (from `Data.Proxy`) around so GHC can choose the correct instance to fetch *genParser* from.

The attentive reader might have noticed a few problems with the CSV parser presented in figure 3. If we try to compile that code GHC will complain about a recursive type synonym. That is very easy to fix, however. We just wrap the recursive calls in a newtype and provide a cannonical instance[1] for it.

```
type LineParserA = Line :<$> Iden
                        :<|> Iden :<*> Sym "," :*> LineParser
newtype LineParser = LP LineParserA
instance HasParser LineParser where
  type Result LineParser = Result LineParserA
  genParser _ = genParser (Proxy :: Proxy LineParserA)
```

2.1 The Tagging Combinator

Tagging is the least intuitive of the combinators, for it deserves its own section. The reason for including it in the library is to provide the user with a way to generate hiw own datatypes instead of standard[2] ones. The important observation is that any Haskell type is isomorphic, by definition, to a sum-of-products, which can be expressed using standard types.

The parser instance for a :<$> b is defined as:

[1] In fact, we provide Template Haskell code to do exactly that. The user should just call $(*deriveRec* '' *LineParser* '' *LineParserA*).

[2] We call standard types any type built using () , *Either* , (,) , [] and atomic types such as *Integer*, *String*, *Double*, ...

```haskell
instance (HasParser a, HasIso k (Result a)) ⇒ HasParser (k :<$> a) where
  type Result (k :<$> a) = k
  genParser _ = genParser (P a) >>= return ∘ og
```

Where the *HasIso a b* class defines two functions $go :: a \to b$ and $og :: b \to a$ to convert values from one type to another. In our CSV example, we have the following instance (which is nothing more than the initial algebra for the underlying functor):

```haskell
instance HasIso Line (Either String (String, Line)) where
  go (One s)        = Left s
  go (More s l)     = Right (s, l)
  og (Right (s, l)) = More s l
  og (Left s)       = Left s
```

The calculation below shows that *LineParser* indeed has the expected result type. Here we use $a{\sim}b$ to denote an isomorphism between a and b meaning that we parse a value of type b and inject it into something of type a through $og :: b \to a$.

```haskell
Result LineParser = Line∼Result (Iden :<|> Iden :<*> Sym "," :*> LineParser)
                  = Line∼Either (Result Iden) (Result (Iden :<*> Sym "," :*> LineParser))
                  = Line∼Either String (Result Iden, Result (Sym "," :*> LineParser))
                  = Line∼Either String (String, Result LineParser)
                  = Line∼Either String (String, Line)
```

And the result is precisely the *HasIso* instance that we have for the type *Line*. In fact, we provide Template Haskell code to generate these mechanical isomorphisms automatically, the user would just call $(*deriveIso* '' *Line*).

Even though we still have something of type *Line* on the right-hand-side of our isomorphism, that does not represent a problem as the instances will open it into another sum-of-products whenever needed. This is possible since the instance chosing is guided by induction in our grammar combinators, and a *LineParser* is always *tagged* by *Line*.

2.2 Lexing Remarks

In the example we gave in figure 3 we use a few atomic grammar combinators that were left unexplained, such as *Iden* and *Sym* ",". The reason for this is that the semantics of these combinators depends on the underlying lexer being used.

We provide a class *HasLexer lang* which ties this knot. It is fairly straight forward to use. The full, correct code, of the CSV example is shown below (where *a* :<!> *b* means parse *a* but only if it is *not* followed by *b*).

```haskell
data CSV
```

```
instance HasLexer CSV where
  identStart _  ','  = False
  identStart _  '\n' = False
  identStart _  _    = True
  identLetter p      = identStart p
  reservedList _     = []
type CSVParser = VMany LineParser
type LineParser'
  = Line :<$> Ident CSV :<!> Sym CSV ","
         :<|> Ident CSV :<*> Sym CSV "," :*> LineParser
newtype LineParser = LP LineParser'
data Line = One String | More String Line
  deriving (Eq, Show, Ord)
 $(deriveRec '' LineParser '' LineParser')
 $(deriveIso  '' Line)
```

The *HasLexer lang* class defines lexing primitives for a language, which is specified through an empty data constructor. The type signature for *identStart*, for instance, is:

```
identStart :: Proxy lang → Char → Bool
```

And it decides which characters start an identifier. For the readers familiar with `Text.Parsec`, our *HasLexer* is just a lifting from Parsec's lexing primitives to a typeclass. In fact, we use `Text.Parsec` and `Text.PrettyPrint` as our underlying libraries for parsing and printing.

2.3 Pretty Printing

Generating a pretty-printer is analogous to generating a parser, with the difference that a few combinators will have the same parsing behaviour but not the same pretty printing behaviour.

Our *HasPrinter* class also needs to access the *Result* type family declared in the *HasParser* class. We therefore assume that *HasParser a* ⇒ *HasPrinter a*.

```
class (HasParser a) ⇒ HasPrinter a where
  pp :: Proxy a → Result a → Doc
```

The *deriveRec* template haskell directive will also generate a *HasPrinter* instance for an encapsulated type.

If we require our pretty printer and parser to be an isomorphism, we run into a few problems. Namelly, this isomorphism has to be modulo formating and comments. These problems are left as future work.

2.4 Summary

On this section we presented our *grammar combinators* library with a simple use case. We showed how to encode a language's grammar on Haskell type-level, which allows us to generate both a parser and a pretty printer for it. We proposed a solution for handling mutually recursive and user defined types.

The advantages of this approach are many. On one side, it is easy to do generic programming on the elements resulting from our parsers, once the target types are all built using products, coproducts, unit and lists (minus isomorphism for user defined types). On the other hand, it is extensible! For a user can always define new domain-specific combinators and immediatly integrate them with the rest of the library.

In the following sections we give the intuition behind what captures differences in a type a. As expected, this definition follows by induction on the structure of a. Later, we show how we proved that this definition of a *diff* is correct, in Agda.

3 Diffing the results

- *Give some context: Tree-edit distance;*

To Research!

- *Check out the antidiagonal with more attention: `http://blog.sigfpe.com/2007/09/type-of-distinct-pairs.html`*
- *The LCS problem is closely related to diffing. We want to preserve the LCS of two structures! How does our diffing relate? Does this imply maximum sharing?*

Having a regular, yet extensible, way to parse different files gives us a stepping stone to start discussing how to track differences in the results of those parsers. There has been plenty of research on this topic ([CITE STUFF!]), however, most of the time data is converted to an untyped intermediate representation. We would like to stay type-safe. [WHY?] Our research shows that the type $D\ a$ that expresses the differences between two elements of type a can be determined by induction on a's structure.

- *introduce* edit-script, diffing *and* patching *or* apply

For example, let us see the differences between the original CSV and Alice's edits:

```
items      ,qty ,unit  | items      ,qty ,unit
wheat-flour,2   ,cp    | wheat-flour,1   ,cp
eggs       ,2   ,units | eggs       ,2   ,units
```

Alice | Original

Fig. 4. Alice's edits

As we can see, Alice edited the second line of the file. A line-based edit script, which is something that transform a file into another, would look like the one presented in figure 5. That edit script has a few problems: (A) it is deleting and inserting almost identical lines and (B) it is unaware of the CSV file semantics, making it harder to identify actual conflicts.

```
Cpy "items,qty,unit"
  (Del "weat-flour,1,cp"
    (Ins "wheat-flour,2,cp"
      (Cpy "eggs,2,unis"
        End)))
```

Fig. 5. Line-based edit-script for figure 4

- *Explain the patching problem.*
- *We want a type-safe approach.*
- *Argue that the types resulting from our parser are in a sub-language of what we treated next.*

4 Context Free Datatypes

- *Explain the universe we're using.*
- *Explain the intuition behing our D datatype.*
- *Mention that it is correct.*

```
data U : ℕ → Set where
  u1    : {n : ℕ} → U n
  _⊕_ : {n : ℕ} → U n → U n → U n
  _⊗_ : {n : ℕ} → U n → U n → U n
  β     : {n : ℕ} → U (suc n) → U n → U n
  μ     : {n : ℕ} → U (suc n) → U n
  vl    : {n : ℕ} → U (suc n)
  wk    : {n : ℕ} → U n → U (suc n)
```

5 Related Work

- People have done similar things... or not.

6 Conclusion

- This is what we take out of it.

References

1. Alp Mestanogullari, Sönke Hahn, Julian K. Arni, and Andres Löh. Type-level web apis with servant: An exercise in domain-specific generic programming. In *Proceedings of the 11th ACM SIGPLAN Workshop on Generic Programming*, WGP 2015, pages 1–12, New York, NY, USA, 2015. ACM.

2. Tillmann Rendel and Klaus Ostermann. Invertible syntax descriptions: Unifying parsing and pretty printing. *SIGPLAN Not.*, 45(11):1–12, 2010.
3. Jeremy Shaw. boomerang. `http://hackage.haskell.org/package/boomerang`. Acessed: November 2015.