Type error customization in uu-parsinglib*

Carlos Tomé Cortiñas

June 23, 2017

1 Introduction

In this document we present the implementation of a wrapper over the uu-parsinglib library for giving domain specific error messages to the user.

We have divided this document into several sections regarding the customization of each module included in the library. Moreover, in the appendices we include the modules that define some extra functionality for pretty printing (is not specific to this library) and also some common error combinators for the parsing library (domain specific).

2 Text.ParserCombinators.UU.Core

2.1 *Functor*, *Applicative*, *Alternative* and *ExtAlternative*

Many of the functionality provided by this parser library comes from the use of *Functor*, *Applicative* and *Alternative* type classes defined in Haskell standard library. Instead of redefining the classes to give custom error we define the same combinators they offer but enhanced with custom error messages.

The disadvantage of this approach is that the user has to be careful to use the ones defined here and not the provided ones by *Prelude*.

Moreover, uu-parsinglib defines the type class *ExtAlternative* which is meant to provide greedy versions of the parsers that can be built from the Haskell *Alternative* typeclass.

To customize the type errors of the methods of this class we will export the original class but customized versions of the combinators it offer.

Therefore, the combinators that we will customize here are,

$$
\begin{array}{l}
(<\$>) :: \mathit{Functor}\ f \Rightarrow (a \to b) \to f\ b \to f\ a \\
(<\star>) :: \mathit{Applicative}\ f \Rightarrow f\ (a \to b) \to f\ a \to f\ b \\
(<\star) :: \mathit{Applicative}\ f \Rightarrow f\ a \to f\ b \to f\ a \\
(\star>) :: \mathit{Applicative}\ f \Rightarrow f\ a \to f\ b \to f\ b \\
(<|>) :: \mathit{Alternative}\ f \Rightarrow f\ a \to f\ a \to f\ a \\
(\ll|>) :: \mathit{ExtAlternative}\ p \Rightarrow p\ a \to p\ a \to p\ a
\end{array}
$$

*https://hackage.haskell.org/package/uu-parsinglib

$(<?>) :: ExtAlternative\ p \Rightarrow p\ a\ \ \rightarrow String \rightarrow p\ a$

$must\_be\_non\_empty :: ExtAlternative\ p \Rightarrow String \rightarrow p\ a \rightarrow c \rightarrow c$
$must\_be\_non\_empties :: ExtAlternative\ p \Rightarrow String \rightarrow p\ a \rightarrow p\ b \rightarrow c \rightarrow c$

By looking at their type, we can understand that all of them share an almost identical type. Except for the last two function the identifiers for rest of them are very similar and can be mistakenly interchanged in their use.

Because the difference is most significant in the type of the first argument (except for $(<?>)$), we will customize the type errors to be biased towards identifying first the second argument being a parser p applied to some type a, and then regarding the possible type errors that derivate from the first argument suggest other functions in the list as possible solutions.

However, there is a drawback with this approach. Within the type error framework we are not able to know with certainty that at a given point in the error message a type p is indeed a parser. For example, in the following type signature,

```
CustomErrors
  [p1 :∝: p a :⇛: ExpectedErrorMessage "(<$>)" 2 "a parser" p1
    , [Check (IsParser p)]
    , ...
    ] ⇒ p1 → ...
```

At ... , we would like that we have certainty of p being a parser. However, the semantics of the combinators for customizing error messages do not ensure us that this is the case.

Because of such limitation, we can make use of type families that will help us to discard the cases where we are sure the argument cannot be a parser. If the type of p_1 cannot be decomposed into some p applied to a then it is not a parser, maybe is some type of kind $\star$ such as Int, String, etc.

But if the type can be decomposed, then we must make sure is not of some type that we know it is not an instance of *Parser*. For example, $p = ((\rightarrow)\ b)$ has the right kind but it is not a parser.

This are defined in **??** as *IsNotOfParserKind* and *IsNotAParser*. The former checks the condition of being of the appropriate shape and the later discards cases we know are not parsers.

Now that we have some kind of assurance (even not that much) that the second argument to the function looks like a parser, we can check the second argument and prompt the user with useful hints about the possible *sibling* he intended to use in case there is a type error.

For example, in a call to $(<\$>)$ that makes use of a first argument of type $p\ (a \rightarrow b)$ we can be pretty sure that the user intended to use $(<\star>)$ instead. However, this interpretation only holds if we already know that the second argument is of type $p\ a$ for a parser p. Therefore, we delay the check of the first argument to a second place to be able to make such suggestion.

If instead the type of the first argument turns out to be $p\ a$ then we should suggest that the user maybe wanted to use either $(<¿)or(¡¡>)$. As a last resource if the underlying types do not match we can still ask the user if he intended to use $(<\star)$ or $(\star>)$.

It is important to remark that if the first type is not a parser but a *String* then we won't suggest (<?>) as a *sibling*. This is because we cannot encode nested conditions within the type error DSL.

```
(<$>) ::
  CustomErrors
    [ [IsNotOfParserKind "(<$>)" 2 p2 p a1]
    , [IsNotAParser p :≁: False :⇛: ExpectedErrorMessage "(<$>)" 2 "a parser" p2]
    , [f1 :≁: (a → b) :⇛?:
      ( [f1 :≁?: p (a1 → b) :⇛!:
            VSep [Text "The #1 argument to '(<$>)' is a function wrapped in a parser:"
                  , Indent 2 (ShowType f1)
                  , Text "Maybe you pretended to use '(<*>)'?"]
        , f1 :≁?: p a1 :⇛!:
            VSep [Text "The #1 argument to '(<$>)' is a parser not a plain function:"
                  , Indent 2 (ShowType f1)
                  , Text "It matches the parser type of the #2 argument." $$
                    Text "Maybe you pretended to use '(<|>)' or '(<<|>)'?"]
        , f1 :≁?: p c :⇛!:
            VSep [Text "The #1 argument to '(<$>)' is a parser not a plain function:"
                  , Indent 2 (ShowType f1)
                  , Text "Maybe you pretended to use '(<*)' or '(*>)'?"]]
        , ExpectedErrorMessage "(<$>)" 1 "a function of at least 1 argument" f1)]
    , [a :≁: a1 :⇛:
      VSep [Text "In the application of '(<$>)', the source type of the function in the #1 argument"
            $$Quote (ShowType f1) :◇: Comma
          , Indent 2 (ShowType a)
          , Text "and the underlying type of the parser in the #2 argument,"
            $$Quote (ShowType p2) :◇: Comma
          , Indent 2 (ShowType a1)
          , Text "should match."]]
    , [Check (IsParser p)]
    ] ⇒ f1 → p a1 → p b
(<$>) = (Applicative. <$>)
(<⋆>) ::
  CustomErrors
    [ [IsNotOfParserKind "(<*>)" 2 p2 p1 a1]
    , [IsNotAParser p1 :≁: False :⇛: ExpectedErrorMessage "(<*>)" 2 "a parser" p1]
    , [IsNotOfParserKind "(<*>)" 1 p4 p3 f1]
    , [IsNotAParser p3 :≁: False :⇛?:
      ( [p4 :≁?: (a1 → b) :⇛!:
            VSep [Text "The #1 argument to '(<*>)' is a plain function,"
                  , Indent 2 (ShowType p4)
                  , Text "but it should be wrapped on a parser as,"
                  , Indent 2 (ShowType (p1 p4))
                  , Text "Maybe you pretended to use '(<$>)'?"]]
        , ExpectedErrorMessage "(<*>)" 1 "a parser with a function type of at least 1 argument" p3)]
    , [p1 :≁: p3 :⇛: DifferentParsers "(<*>)" [ (p1, 2), (p3, 1)]]
    , [f1 :≁: (a → b) :⇛?:
      ( [f1 :≁?: a1 :⇛!:
            VSep [Text "The #1 argument to '(<$>)' is a parser not a plain function:"
                  , Indent 2 (ShowType f1)
                  , Text "It matches the parser type of the #2 argument." $$
                    Text "Maybe you pretended to use '(<|>)' or '(<<|>)'?"]]
        , VSep [Text "The #1 argument to '(<$>)' is a parser without an underlying function:"
                  , Indent 2 (ShowType p4)
```

```
                    , Text "Maybe you pretended to use '(<*)' or '(*>)'?"])]
      , [a :≁: a1 :⇛:
        VSep [Text "In the application of '(<*>)', the source type of the function in the #1 argument"
              $$ Quote (ShowType f1) :◇: Comma
             , Indent 2 (ShowType a1)
             , Text "and the underlying type of the parser of the #2 argument,"
               :⊕: Quote (ShowType p1)
             , Indent 2 (ShowType a)
             , Text "should match."]]
      , [Check (IsParser p1)]
      ] ⇒ p4 → p2 → p1 b
(<⋆>) = (Applicative.<⋆>)
```

For the functions $(<\star)$, $(\star>)$ it is not possible to propose $(<\star>)$ as a *sibling*because if the first parser has a function type inside is still well typed. However, for the case of the first argument being a plain function we can suggest the user that maybe he/she wanted to write (<$>) instead.

```
type ErrorApplicative (name :: Symbol) p1 p2 p3 p4 a b c = CustomErrors
  [ [IsNotOfParserKind name 2 p2 p1 a]
    , [IsNotAParser p1 :≁: False :⇛: ExpectedErrorMessage name 2 "a parser" p1]
    , [IsNotOfParserKind "(<*)" 1 p4 p3 b]
    , [IsNotAParser p3 :≁: False :⇛?:
       ( [p4 :≁?: (a → c) :⇛!:
         VSep [Text "The #1 argument to" :⊕: Quote (Text name)
                 :⊕: Text "is a plain function,"
                , Indent 2 (ShowType p4)
                , Text "but it should be a parser type." $$
                 Text "Maybe you pretended to use '(<$>)'?"]]
         , ExpectedErrorMessage name 1 "a parser" p4)]
    , [p1 :≁: p3 :⇛: DifferentParsers name [ (p1, 2), (p3, 1)]]
    , [Check (IsParser p1)]
    ]
(<⋆) :: ErrorApplicative "(<*)" p1 p2 p3 p4 a b c ⇒ p4 → p2 → p1 b
(<⋆) = (Applicative.<⋆)
(⋆>) :: ErrorApplicative "(*>)" p1 p2 p3 p4 a b c ⇒ p4 → p2 → p1 a
(⋆>) = (Applicative.⋆>)
```

For the combinators from *Alternative* and *ExtAlternative* , once we can assure with some certainty that the second argument is of type $p\ a$ for some parser p and an argument a, we can proceed to check the type of the first argument. If its type is a plain function not wrapped in a parser then we should suggest <$> as *sibling*.

In the case it is a parser but the underlying type is a function such that the source type matches a then we should suggest $(<\star>)$ as a *sibling*. In the remaining case if the underlying type is not a function and doesn't match a then we can suggest that maybe the intention was to use either $(<\star)$ or $(\star>)$.

```
type ErrorAlternative (name :: Symbol) p1 p2 p3 p4 a b c = CustomErrors
  [ [IsNotOfParserKind name 2 p2 p1 a]
    , [IsNotAParser p1 :≁: False :⇛: ExpectedErrorMessage name 2 "a parser" p1]
    , [IsNotOfParserKind "(<|>)" 1 p4 p3 b]
    , [IsNotAParser p3 :≁: False :⇛?:
       ( [p4 :≁?: (a → c) :⇛!:
```

```
      VSep [Text "The #1 argument to" :⊕: Quote (Text name)
              :⊕: Text "is a plain function,"
              , Indent 2 (ShowType p4)
              , Text "but it should be a parser type." $$
              Text "Maybe you pretended to use '(<$>)'?"]]
      , ExpectedErrorMessage name 1 "a parser" p4)]
  , [p1 :≁: p3 :⇛: DifferentParsers "(<|>)" [ (p1, 2), (p3, 1)]]
  , [a :≁: b :⇛?:
    ( [b :≁?: (a → c) :⇛!:
      VSep [Text "The #1 argument to" :⊕: Quote (Text name)
              :⊕: Text "is a parser with an underlying function type,"
              , Indent 2 (ShowType p4)
              , Text "but it should match the #2 argument parser type." $$
              Text "Maybe you pretended to use '(<*>)'?"]]
    , Text "Don")]
  , [Check (IsParser p1)]
  ]
(<|>) :: ErrorAlternative "(<|>)" p1 p2 p3 p4 a b c ⇒ p4 → p2 → p1 a
(<|>) = (Applicative. <|>)
(≪|>) :: ErrorAlternative "(<<|>)" p1 p2 p3 p4 a b c ⇒ p4 → p2 → p1 a
(≪|>) = (Core. ≪|>)
```

For the combinator (<?>) we can check in paralel the second argument to be of type *String* and the first one to be a parser like. In order to be able to suggest corrections in case the second argument is not a *String* we are going to bias the type error message towards the first argument being a parser.

In this way, if the second is also a parser we can suggest one of the above combinators. However, we cannot conditionally check in case the second argument is not a *String* if the first one is a parser with an underlying function type to suggest (<⋆>).

```
(<?>) :: CustomErrors
  [ [IsNotOfParserKind "(<?>)" 1 p2 p1 a]
  , [IsNotAParser p1 :≁: False :⇛: ExpectedErrorMessage "(<|>)" 2 "a parser" p1]
  , [str :≁: String :⇛?:
    ( [str :≁?: p1 a :⇛!:
        VCat [Text "The #2 argument to (<?>) is a parser and not a String."
              , Text "Maybe you wanted to use '(<<|>)' or '(<|>)'?"]
      , str :≁?: p1 b :⇛!:
        VCat [Text "The #2 argument to (<?>) is a parser and not a String."
              , Text "Maybe you wanted to use '(<*)' or '(*>)'?"]]
    , ExpectedErrorMessage "<?>" 2 "a String" str)]
  , [Check (IsParser p1)]
  ] ⇛ p2 → str → p1 a
(<?>) = (Core. <?>)
```

The last two combinators are not really *sibling*of the above, but they are between them. We can see that their type is similar except that one takes an extra argument.

```
must_be_non_empty :: CustomErrors
  [ [IsNotOfParserKind "must_be_non_empty" 2 p2 p1 a
  , str :≁: String :⇛: ExpectedErrorMessage "must_be_non_empty" 1 "a String for the error message" str]
  , [IsNotAParser p1 :≁: False :⇛: ExpectedErrorMessage "must_be_non_empty" 2 "a parser" p1]
```

```
  , [cf :∝: (c → c) :⇛?:
       ( [cf :∝?: (p1 b → c → c) :⇛!:
         VCat [Text "One argument extra given to must_be_non_empty,"
           , Text "Maybe you wanted to use 'must_be_non_empties'?"]]
         , ExpectedErrorMessage "must_be_non_empty" 3 "an argument" pbc)]
  , [Check (IsParser p1)]
  ] ⇒ str → p2 → cf
must_be_non_empty = Core.must_be_non_empty

must_be_non_empties :: CustomErrors
  [ [IsNotOfParserKind "must_be_non_empties" 2 p2 p1 a
    ,  str :∝: String :⇛: ExpectedErrorMessage "must_be_non_empties" 1 "a String for the error message" str]
    , [IsNotAParser p1 :∝: False :⇛: ExpectedErrorMessage "must_be_non_empties" 2 "a parser" p1]
    , [pbc :∝: (p3 b → c → c) :⇛?:
       ( [pbc :∝?: (c → c) :⇛!: VCat [Text "Missing argument for must_be_non_empties,"
         , Text "Maybe you wanted to use 'must_be_non_empty'?"]]
         , ExpectedErrorMessage "must_be_non_empties" 3 "a parser" pbc)]
    , [p1 :∝: p3 :⇛: DifferentParsers "(<|>)" [ (p1, 2), (p3, 3)]]
  , [Check (IsParser p1)]
  ] ⇒ str → p2 → pbc
must_be_non_empties = Core.must_be_non_empties
```

As a final note, the *IsParser* typeclass is defined in the module *Core* as a means to be a unifying class for the ones supported by the parsers. In our solution we keep the type of all combinators polymorphic in the parser type p as long as is an instance of this class. However, the only ever instance of the class declared in the library is $P\ st\ a$. Maybe it would have been much more easier, with less polymorphic functions type errors are much more easy to spell, to make all the functions work for only this type. We choose our approach because is more extensible in the sense that if another parser of *IsParser* is declared our custom type error messages do not need to be changed.

2.2 Evaluation functions

```
parse :: (Eof t) ⇒ P t a → t → a
parse_h :: (Eof t) ⇒ P t a → t → a
```

In order to give a custom error message, we should note that the error for this functions has to be biased towards the type $P\ t\ a$, because if that argument is not a parser then it doesn't make sense to check whether the second argument's type t matches the parser state.

Moreover, it is a common source of errors to use the evaluator function of a DSL, in this case parser combinators, and supply the arguments in the wrong order. In case the first argument is not a parser, we can still check if the second argument is a parser and the first argument matches the type for the state of the parser. In this situation we should suggest to the user that is very likely the arguments are swapped.

```
type ParserError (name :: Symbol) = ∀ p t t1 a.
  CustomErrors
    [ [p :∝: P t1 a :⇛?:
      ( [t :∝?: P p a :⇛!:
```

```
      VCat [ Text "It seems that the #2 argument given to" :⊕: Text name
           , Text "is a parser" :⊕: Quote (ShowType t)
           , Text "and the #1 argument's type matches the state for such parser."
           , Text "Maybe, are the arguments swapped?"]]
    , ExpectedErrorMessage name 1 "a parser" p)]
    , [t1 :≁: t    :⇛: ExpectedErrorMessage name 2 "the state for the parser" t]
    , [Check (Eof t)]
    ] ⇒ p → t → a
parse :: ParserError "parse"
parse = Core.parse
parse_h :: ParserError "parse_h"
parse_h = Core.parse_h
```

2.3 Various combinators

Some other combinators present in the module are,
addLength :: $Int \to P\ st\ a \to P\ st\ a$ *micro* :: $P\ state\ a \to Int \to P\ state\ a$

```
addLength ::
  CustomErrors
    [ [int :≁: Int :⇛: ExpectedErrorMessage "addLength" 1 "the number of elements to add" int
      , p :≁: P st a :⇛: ExpectedErrorMessage "addLength" 2 "a parser" p]
    ] ⇒ int → p → P st a
addLength = Core.addLength
micro ::
  CustomErrors
    [ [int :≁: Int :⇛: ExpectedErrorMessage "micro" 2 "the cost to add" int
      , p :≁: P state a :⇛: ExpectedErrorMessage "micro" 1 "a parser" p]
    ] ⇒ p → int → P state a
micro = Core.micro
pSymExt :: (∀ a.(token → state → Steps a) → state → Steps a) → Core.ℕ → Maybe token → P state token
pSymExt = Core.pSymExt
pSwitch :: (st1 → (st2, st2 → st1)) → P st2 a → P st1 a
pSwitch = Core.pSwitch
```

3 Text.ParserCombinators.UU.Derived

3.1 *pEither*

The first interesting combinator that can be customized of this module is

```
pEither :: IsParser p ⇒ p a → p b → p (Either a b)
```

In this case, we have to check that both argument are parsers or at least look like parsers and if they are different give the appropiate error message.

```
pEither :: CustomErrors
  [ [IsNotOfParserKind "pEither" 1 p1 p a
    , IsNotOfParserKind "pEither" 2 p3 p2 b]
  , [IsNotAParser p :≁: False :⇛: ExpectedErrorMessage "pExact" 1 "a parser" p
    , IsNotAParser p2 :≁: False :⇛: ExpectedErrorMessage "pExact" 2 "a parser" p2]
  , [p :≁: p2 :⇛: DifferentParsers "pEither" [ (p1, 1), (p3, 2)]]
```

```
    , [Check (IsParser p)]
    ]  ⇒ p1 → p3 → p (Either a b)
pEither = Derived.pEither
```

3.2 Infix combinators

More Another interesting case for error customization is when an arrow type is expected as an argument and some other argument's type has a relation with it. In the combinator with type:

```
(<$$>) :: IsParser p ⇒ (a → b → c) → p b → p (a → c)
(<??>) :: IsParser p ⇒ p a → p (a → a) → p a
```

First, we can see that the first argument is expected to be a function with at least two arguments. Moreover, the second argument's type must coincide with the underlying type of the parser p. The later error is dependent of the former because if the first argument is not a function then there is nothing else to check. Therefore, we will encode this with the following type signature.

```
(<$$>) :: CustomErrors
  [ [f :≁: (a → b1 → c) :⇛:
      VCat [Text "Expected as 1st argument a function type of 2 arguments but got:"
           , Indent 4 (ShowType f)]
    , IsNotOfParserKind "(<$$>)" 2 p1 p b]
    , [IsNotAParser p :≁: False :⇛: ExpectedErrorMessage "(<$$>" 2 "a parser" p1]
    , [b1 :≁: b :⇛:
      VCat [Text "The underlying type of the parser" :⊕: Quote (ShowType p1)
           , Indent 4 (ShowType b)
           , Text "and the type of the 2nd argument of the function:"
           , Indent 4 (ShowType f)
           , Text "have to agree."]]
    , [Check (IsParser p)]
    ] ⇒ f → p b → p (a → c)
(<$$>) = (Derived. <$$>)
```

It is interesting to note that through the customization of errors we will find ourselves many times with having multiple arguments that require the same parser. Therefore, by using some type level machinery we abstracted over this case providing the type level function *DifferentParsers* (f::*Symbol*) (p::$[(k, \mathbb{N})]$). This function defined in expectes the name of the function we are customizing along a type level list of parsers numbered with the argument were they appear.

With this tools, we can encode a custom error for the following combinator:

As:

```
(<??>) :: CustomErrors
  [ [(p1 :≁: p) :⇛: DifferentParsers "<??>" [ (p1, 1), (p, 2)]]
    , [f :≁: (a → a) :⇛:
      VCat [Text "Expected the underlying type of the 2nd parser to be a function type"
           :⊕: Text "of 1 argument, but got:"
           , Indent 4 (ShowType f)]]
    , [a :≁: a1       :⇛:
```

```
        VCat [Text "The underlying type of the 1st parser" :⊕: Quote (ShowType p)
             , Indent 4 (ShowType a1)
             , Text "and the type of source and target of function inside the 2nd parser:"
             , Indent 4 (ShowType f)
             , Text "have to agree."]]
    , [Check (IsParser p)]
    ] ⇒ p1 a1 → p f → p a1
(<??>) = (Derived. <??>)
```

We should also note that the case where a parser is wrapping an arrow type. Therefore, we can also abstract over this fact with the type level function $FunctionTypeParser\ (arg :: \mathbb{N})\ (f :: Symbol)\ (n :: \mathbb{N}) :: ErrorMessage$ defined in . add reference

3.3 Function composition

The follwowing two combinators, show a similarity between their types. We can think that besides both parsers being the same parsers and the underlying types being functions of one argument, if the combination of source/target type of the functions does not match we can suggest the user that maybe he/she pretended to use the other one.

```
(<.>) :: IsParser p ⇒ p (b → c) → p (a → b) → p (a → c)
(<..>) :: IsParser p ⇒ p (a → b) → p (b → c) → p (a → c)
```

```
(<.>) :: CustomErrors
   [ [p :≁: p2 :⇛: DifferentParsers "<.>" [ (p, 1), (p2, 2)]
     , f1 :≁: (b1 → c) :⇛: FunctionTypeParser 1 f1 1
     , f2 :≁: (a → b2) :⇛: FunctionTypeParser 2 f2 1]
     , [b1 :≁: b2      :⇛:
        VCat [Text "The target type of the 2nd function:"
             , Indent 4 (ShowType b2)
             , Text "and the source type of the first one:"
             , Indent 4 (ShowType b1)
             , Text "should match."
             , Text "Maybe you wanted to use (<.>) instead?"]]
     , [Check (IsParser p)]
     ] ⇒ p f1 → p2 f2 → p (a → c)
(<.>) = (Derived. <.>)
(<..>) :: CustomErrors
   [ [p :≁: p2 :⇛: DifferentParsers "<..>" [ (p, 1), (p2, 2)]
     , f2 :≁: (b1 → c) :⇛: FunctionTypeParser 1 f1 1
     , f1 :≁: (a → b2) :⇛: FunctionTypeParser 2 f2 1]
     , [b1 :≁: b2      :⇛:
        VCat [Text "The target type of the 2nd function:"
             , Indent 4 (ShowType b2)
             , Text "and the source type of the first one:"
             , Indent 4 (ShowType b1)
             , Text "should match."
             , Text "Maybe you wanted to use (<..>) instead?"]]
     , [Check (IsParser p)]
     ] ⇒ p f1 → p2 f2 → p (a → c)
(<..>) = (Derived. <..>)
```

3.4 List with separation parsers

For the familiy of separation parsers the error is quite straightforward to customize. We should make sure that both arguments are parser like arguments and finally the underlying typ has to match.

```
type PListSep (name :: Symbol) = ∀ p p1 p2 p3 a b.
  CustomErrors
    [ [IsNotOfParserKind name 1 p1 p a
      , IsNotOfParserKind name 2 p3 p2 b]
    , [IsNotAParser p :≁: False :⇛: ExpectedErrorMessage name 1 "a parser" p
      , IsNotAParser p2 :≁: False :⇛: ExpectedErrorMessage name 2 "a parser" p2]
    , [p  :≁: p2 :⇛: DifferentParsers name [ (p, 1), (p2, 2)]]
    , [a :≁: b    :⇛: VSep [Text "The underlying type of both parsers,"
                    , Indent 2 (Quote (ShowType p1) :⊕: Text "and" :⊕: Quote (ShowType p3))
                    , Text "does not match."]]
    , [Check (IsParser p)]
    ] ⇒ p1 → p3 → p [a]

pListSep    :: PListSep "pListSep"
pListSep    = Derived.pListSep

pListSep_ng :: PListSep "pListSep_ng"
pListSep_ng = Derived.pListSep_ng

pList1Sep   :: PListSep "pList1Sep"
pList1Sep   = Derived.pList1Sep

pList1Sep_ng :: PListSep "pList1Sep_ng"
pList1Sep_ng = Derived.pList1Sep_ng
```

3.5 Chain parsers

In the combinators for chaining parsers the customized type error is a bit involved. It must first check that the provided arguments are parser like types. Then the underlying type of the first parser must be the function used to chain, and it should be of exactly two arguments that moreover match the type of the second argument parser.

An option to customize the error to this family of combinators would be to check the combinations of types for both arguments to check which one differs and then give a precise error for it. Another option, that we choose to follow is to tell the user that indeed we expect a function type of two arguments with the type $c \to c \to c$. Therefore, in a subsequent step we check if the c matches the underlying type of the second argument parser.

```
type PChain (name :: Symbol) = ∀ p p1 p2 p3 fc c1 c.
  CustomErrors
    [ [IsNotOfParserKind name 1 p1 p fc
      , IsNotOfParserKind name 2 p3 p2 c]
    , [IsNotAParser p :≁: False :⇛: ExpectedErrorMessage name 1 "a parser" p
      , IsNotAParser p2 :≁: False :⇛: ExpectedErrorMessage name 2 "a parser" p2]
    , [p :≁: p2 :⇛: DifferentParsers name [ (p, 1), (p2, 2)]]
    , [fc :≁: (c1 → c1 → c1) :⇛: FunctionTypeParserEq 1 p1 2]
    , [c1 :≁: c :⇛: VSep [Text "The underlying type of the #2 argument parser,"
      , Indent 2 (Quote (ShowType p3))
      , Text "has to match the type of arguments and target of the function in the #1 argument,"
```

```
      , Indent 2 (Quote (ShowType p1))
      ]]
    , [Check (IsParser p)]
    ] ⇒ p1 → p3 → p c

pChainr :: PChain "pChainr"
pChainr = Derived.pChainr

pChainr_ng :: PChain "pChainr_ng"
pChainr_ng = Derived.pChainr_ng

pChainl :: PChain "pChainl"
pChainl = Derived.pChainl

pChainl_ng :: PChain "pChainl_ng"
pChainl_ng = Derived.pChainl_ng
```

3.6 Repeating parsers

There are some combinators that share a common pattern for repeatedly applying a given parser a fixed number of times. These are,

```
pExact   :: (IsParser f) ⇒ Int → f a → f [a]
pAtLeast :: (IsParser f) ⇒ Int → f a → f [a]
pAtMost  :: (IsParser f) ⇒ Int → f a → f [a]
```

For the cutomized error of this family of combinators, we are going to first check that the second argument is a parser and then that the first one is an *Int*. In case we find the first one is not a parser but an *Int* we can suggest the user that maybe he swapped the arguments. The drawback of this approach is that we will make the suggestion even if the first one is already an *Int* and not a parser. However, there is no way to encode in the framework this double dependency of the first being a parser and the second one being an *Int*.

Moreover, this only occurs in the first check to see if the parser argument we expect has the right kind, this is a parser p applied to some type a, $p\ a$. Once we know it has this shape, we still have to rule out the cases where we know the type is not a parser.

In order to encode all the three cases together we will make use of some type level machinery.

```
type Repeating (name :: Symbol) = ∀ int p p1 a.
  CustomErrors
    [ [p1 :≁: p a :⇛?:
      ( [int ∼ p1 :⇛!: Text "The #2 argument is an 'Int', Maybe the arguments are swapped?"]
        , ExpectedErrorMessage name 2 "a parser" p1)]
      , [IsNotAParser p :≁: False :⇛: ExpectedErrorMessage name 2 "a parser" p1
        , int :≁: Int :⇛: ExpectedErrorMessage name 1 "a 'Int'" int]
        , [Check (IsParser p)]
        ] ⇒ int → p1 → p [a]
```

And now we simply need to write the type signatures using *Repeating* with the appropiate type level *String* for the name of the function. Maybe this could be done more automatically by means of Template Haskell.

```
pExact   :: Repeating "pExact"
pExact   = Derived.pExact
```

```
pAtLeast :: Repeating "pAtLeast"
pAtLeast = Derived.pAtLeast
pAtMost :: Repeating "pAtMost"
pAtMost = Derived.pAtMost
```

For the follwing combinator,

```
pBetween :: (IsParser f) ⇒ Int → Int → f a → f [a]
```

the ideal type error message would be to point out in case the second argument to the function is of type $f\ a$ that maybe one of the combinators above was the intended one to use. However, there is no mechanism that allows us to be sure that indeed is a parser in case the argument is not of *Int* type and therefore we would be misleading the user with the type error. Because of this we choose to only provide basic type error messages in case the arguments type do not match what was expected.

```
pBetween ::
  CustomErrors
    [ [int1 :⊸: Int  :⇛: ExpectedErrorMessage "pBetween" 1 "the minimum number of elements 'Int' to be recognised
      , int2 :⊸: Int :⇛: ExpectedErrorMessage "pBetween" 2 "the minimum number of elements 'Int' to be recognised
        , IsNotOfParserKind name 3 p1 p a]
      , [IsNotAParser p :⊸: False :⇛: ExpectedErrorMessage name 1 "a parser" p1]
      , [Check (IsParser p)]
      ] ⇒ int1 → int2 → p1 → p [a]
pBetween = Derived.pBetween
```

3.7 Other combinators

An interesting case of this pattern occurs when the numbers of parsers involved in the type of a combinator is greater than two. For example,

```
pPacked :: IsParser p ⇒ p b1 → p b2 → p a → p a
```

```
pPacked :: CustomErrors
  [ [p :⊸: p1   :⇛: DifferentParsers "pPacked" [ (p, 1), (p1, 2), (p2, 3)]
    , p1 :⊸: p2 :⇛: DifferentParsers "pPacked" [ (p, 1), (p1, 2), (p2, 3)]]
    , [Check (IsParser p)]
    ] ⇒ p b1 → p1 b2 → p2 a → p a
pPacked = Derived.pPacked
```

4 General remarks and conclusions

- The addition of new siblings to a customized error message is not composable. It involves hardcoding in the correct place of the type the conditions that must be met in order to hint the user with a proper suggestion that when applied will make the expression well typed.

  An example of this can be seen in the encoding of combinators for the *Functor*, *Applicative*, *Alternative* and *ExtAlternative* as explained in 2.

- Encoding requirements over type classes in the error messages is weak. We cannot be sure that at a certain point in the error message for example a type p is indeed an instance of the class *IsParser*. Therefore, we must take additional measures to ensure that we rule out cases we know do not belong to the class. Moreover, this additional measurements poison the type of the combinator that no longer can be reduced to a type similar to the original one.

- The impossibility to include some form of reified expression where the type error is generated forbids the DSL writer to specify precisely in the type error message the source of the error. For now, the only way to refer to it is hardcode the name of the function involved and number the arguments (which indeed leaves room open for a lot of misleading in the errors in case it is wrong).

- The type *ErrorMessage* provided by GHC does not work as smoothly as it should be. For example, printing a type with *ShowType* with a complicated type at the end of a sentence makes it unreadable. As an improvement to this mechanism I would like to have access to all the pretty printing machinery implemented in GHC through a type level API that can allow for much better formatting of the error messages.

- Providing the user with meaningful error messages according to the selected domain is not an easy task. Especially with polymorphic combinators and a lot of siblings present, the author of the library will have to take into account all the possible corner cases of type errors that could arise from the use of the library. Moreover, due to the somewhat limited expressivity of the type error DSL an educated choice has to be made to give preference to some of the arguments to a function over others.

- Sometimes debugging customized type error messages with kind errors can be a bit cumbersome as the outputted error messages by GHC are just unreadable given a somewhat involved customized error. Maybe it could be nice to provide customized kind error messages to the type error DSL itself.

A GHC.TypeErrors.Utils

This module defines domain specific combinators for type error messages for the library.

```
type FunctionType (arg :: ℕ) (f :: ⋆) (n :: ℕ) =
  VCat [ Text "Expected as #" :◇: ShowType arg :⊕:
         Text "argument a function type of" :⊕: ShowType n :⊕:
         Text "arguments but got" :◇: Colon
         , Indent 4 (ShowType f) :◇: Dot]
type FunctionTypeParser (arg :: ℕ) (f :: ⋆) (n :: ℕ) =
  VCat [ Text "Expected as #" :◇: ShowType arg :⊕:
```

```
        Text "argument a parser with an underlying function type of" :⊕: ShowType n :⊕:
        Text "arguments but got" :◇: Colon
        , Indent 4 (ShowType f) :◇: Dot]

type FunctionTypeParserEq (arg :: ℕ) (f :: ⋆) (n :: ℕ) =
  VCat [ Text "Expected as #" :◇: ShowType arg :⊕:
        Text "argument a parser with an underlying function type of" :⊕: ShowType n :⊕:
        Text "arguments, with all arguments and target of the same type but got" :◇: Colon
        , Indent 2 (ShowType f) :◇: Dot]

type family DifferentParsers (f :: Symbol) (p :: [(k, ℕ)]) where
  DifferentParsers f p =
    Text "The parsers of the arguments for" :⊕: Text f :⊕: Text "do not coincide:" $$
      Indent 4 (VCat (Map MakeParserArgSym p))

type family MakeParserArg p where
  MakeParserArg (p, n) = Text "The parser of the #" :◇: ShowType n :⊕:
    Text "argument is" :⊕: Quote (ShowType p) :◇: Dot

data MakeParserArgSym :: ((k, ℕ) ⇝ ErrorMessage) → ⋆

type instance Apply MakeParserArgSym x = MakeParserArg x

type ExpectedErrorMessage (name :: Symbol) (argn :: ℕ) (descr :: Symbol) t =
  VCat [ Text "The #" :◇: ShowType argn :⊕: Text "argument to" :⊕: Quote (Text name)
          :⊕: Text "is expected to be" :⊕: Text descr :◇: Text ", but its type is" :◇: Colon
       , Empty
       , Indent 2 (ShowType t)]

type IsNotOfParserKind (name :: Symbol) (argn :: ℕ) p₁ p a =
  p₁ :≁: p a :⇛: ExpectedErrorMessage name argn "a parser" p₁

type family IsNotAParser (p :: ⋆ → ⋆) where
  IsNotAParser ((→) b) = True
  IsNotAParser [] = True
  IsNotAParser _ = False
```

B GHC.TypeErrors.PP

In this module a basic set of combinators for type level pretty printing of error messages are defined. This module is library independent and maybe it can be completed and made into its own library.

As an aside, the optimal option would be that GHC supports all this combinators by default as they are a type level reflection of the custom Pretty printing that GHC uses internally for displaying error messages to the user.

```
type Empty = Text ""
type Space = Text " "
type Colon = Text ":"
type Dot = Text "."
type Comma = Text ","
type Quote n = Text "’" :◇: n :◇: Text "‘"

type family (:⊕:) (a :: ErrorMessage) (b :: ErrorMessage) where
  a :⊕: b = a :◇: Space :◇: b

infixl 6 :⊕:

type family VCat a where
  VCat [] = Empty
  VCat (x : xs) = x $$ VCat xs

type family VSep a where
```

```
  VSep [] = Empty
  VSep (x : xs) = x $$Empty $$ VSep xs
type family HCat a where
  HCat [] = Empty
  HCat (x : xs) = x :◇: HCat xs
type family HSep a where
  HSep [] = Empty
  HSep (x : xs) = x :⊕: HSep xs
type family Indent (n :: ℕ) (e :: ErrorMessage) where
  Indent 0 x = x
  Indent n x = Empty :⊕: Indent (n − 1) x
data (⇝) :: ⋆ → ⋆ → ⋆
type family Apply (f :: (k1 ⇝ k2) → ⋆) (x :: k1) :: k2
type family Map (f :: (k1 ⇝ k2) → ⋆) (xs :: [k1]) :: [k2] where
  Map f [] = []
  Map f (x : xs) = Apply f x : (Map f xs)
```