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

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

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

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 the second. This means that we will check the second argument to see if it is a parser p applied to some type a, and then in a second phase analyse the first argument for type errors and in the appropiate case suggest other functions from the list as possible corrections.

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,

$\mathit{CustomErrors}$
$\quad [p_1 :\!\!\sim\!\!: p\ a :\!\!\Rightarrow\!\!: \mathit{ExpectedErrorMessage}$ `"(<$>)"` 2 `"a parser"` p_1
$\quad , [\mathit{Check}\ (\mathit{IsParser}\ p)]$
$\quad\quad , \ldots$
$\quad\quad] \Rightarrow p_1 \rightarrow \ldots$

At ... , we would like to know certainly that p is 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.

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

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

At this point, 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 first argument and prompt the user with useful hints about the possible *siblings* 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.

On the other hand, in case the type of the first argument turns out to be $p\ a$ then we should suggest the user that maybe he wanted to use either $(<¿)or(¡¡>)$.

As a last resource if the underlying types do not match we can still ask the user if his intension was to use (<⋆) or (⋆>).

It is important to remark that if the first type is not a parser but a *String* we will not 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 they 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 $p\ a$, we proceed to check the type of the first argument. In case its type is a plain function not wrapped in a parser then we should suggest <$> as *sibling* .

If 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 does not match a then we can suggest that maybe the intention of the user 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 *siblings* 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 an 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 there 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 easier, with less polymorphic functions, to make all the functions work for only this type. We choose our approach as it 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 does not 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. If the first argument is not a parser, we can still check whether 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

Other combinators present in the module that are not polymorphic in the parser type and can be easily customized.

```
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
```

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 appropriate 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

This module provides two infix combinators for parsers with underlying function types,

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

The second combinator, (<??>) is very similar to the ones we defined in 2.1 and up to some extent it can be considered a *sibling* of them. However, it is different in the sense that the plain parser type $p\ a$ occurs in the first argument. Unlike the other combinators, the error message of this type will not be biased towards any of its arguments and only in case the underlying function type is not as expected we will make a suggestion.

```
(<??>) ::
  CustomErrors
    [ [IsNotOfParserKind "(<??>)" 1 p2 p1 a1
      , IsNotOfParserKind "(<??>)" 1 p4 p3 f]
    , [IsNotAParser p1 :≁: False :⇛: ExpectedErrorMessage "(<??>)" 1 "a parser" p2
      , IsNotAParser p3 :≁: False :⇛: ExpectedErrorMessage "(<??>)" 1 "a parser" p4]
    , [p1 :≁: p3 :⇛: DifferentParsers "(<??>)" [ (p2, 1), (p4, 2)]]
    , [f :≁: (a2 → a2) :⇛?:
        ( [f :≁?: a1 :⇛!:
            VSep [ Text "In the application of '(<??>)', the underlying type of the #1 argument parser"
                 $$ Quote (ShowType p2) :◇: Comma
                 , Indent 2 (ShowType a1)
                 , Text "is not a function. But the underlying type of the parser in the #2 argument,"
                 $$ Quote (ShowType p4) :◇: Comma
                 , Indent 2 (ShowType a1)
                 , Text "matches." $$ Text "Maybe you intended to use  '(<|>)' or '(<<|>)'?"]]
        , FunctionTypeParserEq "(<??>)" 2 f 1)]
    , [a1 :≁: a2 :⇛:
        VSep [ Text "In the application of '(<??>)', the underlying type of the #1 argument parser"
             :⊕: Quote (ShowType p2) :◇: Comma
             , Indent 2 (ShowType a1)
             , Text "and the type of source and target of the function inside the #2 argument parser,"
             , Indent 2 (ShowType f)
             , Text "have to match."]]
    , [Check (IsParser p1)]
    ] ⇛ p2 → p3 f → p1 a1
(<??>) = (Derived. <??>)
```

The other combinator <$$> is easier to customize, as we can check independently the first argument to be a function and the second to be a parser.

```
(<$$>) :: CustomErrors
  [ [IsNotOfParserKind "(<$$>)" 2 p2 p b
    , f :≁: (a → b1 → c) :⇛: FunctionTypeParser "(<$$>)" 1 f 2]
  , [IsNotAParser p :≁: False :⇛: ExpectedErrorMessage "(<$$>)" 2 "a parser" p2]
  , [b1 :≁: b :⇛:
    VSep [ Text "In the application of '(<$$>)', the underlying type of the #2 parser argument"
         :⊕: Quote (ShowType p2)
         , Indent 2 (ShowType b)
         , Text "and the type of the #2 argument of the function given as #1 argument,"
         , Indent 2 (ShowType f)
```

```
          , Text "have to match."]]
  , [Check (IsParser p)]
  ] ⇒ f → p b → p (a → c)
(<$$>) = (Derived. <$$>)
```

3.3 Function composition

The following two combinators show a similarity between their types:

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

In order to customize the error message we have to check that both arguments are parser like arguments, and they hold a function type inside. As a last step we should ensure the source/target of the arrow type matches as expected.

When the types do not match we can suggest the user that maybe they intended to use the other.

```
type CompositionError (name :: Symbol) (sug :: Symbol) p p1 p2 p3 f1 f2 a b1 b2 c tf1 tf2 =
CustomErrors
  [ [IsNotOfParserKind name 1 p1 p f1
    , IsNotOfParserKind name 2 p3 p2 f2]
  , [IsNotAParser p :≁: False :⇛: ExpectedErrorMessage name 1 "a parser" p
    , IsNotAParser p2 :≁: False :⇛: ExpectedErrorMessage name 2 "a parser" p2]
  , [p :≁: p2 :⇛: DifferentParsers "pEither" [(p1, 1), (p3, 2)]]
  , [f1 :≁: tf1  :⇛: FunctionTypeParser name 1 f1 1
    , f2 :≁: tf2 :⇛: FunctionTypeParser name 2 f2 1]
  , [b1 :≁: b2  :⇛?:
      ([c :≁?: a :⇛!:
        VCat [Text "The target type of the #2 function,"
             , Indent 2 (ShowType b2)
             , Text "and the source type of the #1 function,"
             , Indent 2 (ShowType b1)
             , Text "should match."
             , Text "Maybe you wanted to use" :⊕: Quote (Text sug) :⊕: Text "?"]]
       , VCat [Text "In the use of" :⊕: Quote (Text name) :⊕: Text "the target type of the #2 function,"
             , Indent 2 (ShowType b2)
             , Text "and the source type of the #1 function,"
             , Indent 2 (ShowType b1)
             , Text "should match."])]
  , [Check (IsParser p)]
  ]
(<.>) :: CompositionError "<.>" "<..>" p p1 p2 p3 f1 f2 a b1 b2 c (b1 → c) (a → b2)
  ⇒ p1 → p3 → p (a → c)
(<.>) = (Derived. <.>)
(<..>) :: CompositionError "<..>" "<.>" p p1 p2 p3 f1 f2 a b1 b2 c (a → b1) (b2 → c)
  ⇒ p1 → p3 → p (a → c)
(<..>) = (Derived. <..>)
```

Perhaps, the abstraction of the custom error to the *CompositionError* type synonym is slightly forced. This is because we must include as the last two arguments, the relation that must hold for each function type between its type variables $a\ b_1\ b_2\ c$.

3.4 List with separation parsers

The common type of this family of parsers is:

```
... :: IsParser p ⇒ p a1 → p a → p [a]
```

The error message for this kind of parsers is straightforward to customize. We should check that both arguments are parser like types and their underlying type matches.

```
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 more 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 also match the type of the second argument parser.

```
... :: IsParser p ⇒ p (c → c → c) → p c → p c
```

An option to customize the error message of this family of combinators would be to check the combinations of types for both arguments and analyse which one differs in type and give a precise error for it. Alternatively, we can tell the user that indeed we expect a function type of two arguments with type $c \to c \to c$. 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 name 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 customized error of this family of combinators, at the beginning we check that the second argument is a parser and 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 they swapped the arguments. The drawback of this approach is that we will make the suggestion even if the first argument 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 being an *Int*.

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]
```

Now, we simply need to write the type signatures using *Repeating* with the appropriate 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 following combinator:

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

If the second argument to the function is of type $f\ a$ maybe the user intended to use one of the functions defined in 3.6. However, there is no mechanism that allows us to ensure that indeed it is a parser and therefore we would be misleading the user with the type error. As a consequence, we choose only to provide basic type error messages in case the type of the arguments does not match.

```
pBetween ::
  CustomErrors
    [ [int1 :≁: Int :⇛: ExpectedErrorMessage "pBetween" 1
                        "the minimum number of elements 'Int' to be recognised" int1
      , int2 :≁: Int :⇛: ExpectedErrorMessage "pBetween" 2
                        "the minimum number of elements 'Int' to be recognised" int2
      , 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

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

The customization of the type error message for the combinator above makes explicit the reason why checking the type of several arguments to be parser does not scale well.

First, we check that all three arguments have parser like types and then that they are of the same parser type. However, we can only check this in blocks of two each time, thus we should include in the error message all given parsers. Moreover, the checking has to be done in different steps so as not to prompt the user with more than one error message.

```
pPacked :: CustomErrors
  [ [IsNotOfParserKind "pPacked" 1 p2 p1 b1
    , IsNotOfParserKind "pPacked" 2 p4 p3 b2
    , IsNotOfParserKind "pPacked" 3 p6 p5 a]
  , [IsNotAParser p1 :≁: False :⇛: ExpectedErrorMessage "pPacked" 1 "a parser" p1
    , IsNotAParser p3 :≁: False :⇛: ExpectedErrorMessage "pPacked" 2 "a parser" p3
    , IsNotAParser p5 :≁: False :⇛: ExpectedErrorMessage "pPacked" 3 "a parser" p5]
  ,  [p1 :≁: p3 :⇛: DifferentParsers "pPacked" [ (p1, 1), (p3, 2), (p5, 3)]]
  ,  [p3 :≁: p5 :⇛: DifferentParsers "pPacked" [ (p1, 1), (p3, 2), (p5, 3)]]
  ,  [Check (IsParser p1)]
  ] ⇒ p2 → p4 → p6 → p1 a
pPacked = Derived.pPacked
```

3.8 Fold parsers

In the combinators for folding a parser, we are going to customize the error message towards the second argument being a parser. Then we can check the first argument to analyse if it is a tuple with a function type.

The error message is longer due to the number of type variables involved to make a proper diagnostic of the error.

```
type PFoldrError (name :: Symbol) = ∀ p1 p2 a a1 a2 a3 b f1 v t1.
  CustomErrors
    [ [IsNotOfParserKind name 2 p2 p1 a]
    , [IsNotAParser p1 :≁: False :⇛: ExpectedErrorMessage name 2 "a parser" p1
      , t1 :≁: (f1, b) :⇛: ExpectedErrorMessage name 1 "a pair" t1]
    , [f1 :≁: (a1 → a2 → a3) :⇛: ExpectedErrorMessage name 1 "a function type as #1 component of the pair" f1]
    , [a1 :≁: a :⇛:
        VSep [Text "In the application of" :⊕: Quote (Text name) :◇: Comma
              :⊕: Text "the type of the #1 argument of the function inside the tuple,"
            , Indent 2 (Quote (ShowType f1))
            , Text "has to match the underlying type of the parser of the #2 argument,"
            , Indent 2 (Quote (ShowType a))]
      , a2 :≁: a3 :⇛:
        VSep [Text "In the application of" :⊕: Quote (Text name) :◇: Comma
              :⊕: Text "the #2 argument and return type of the function type inside the tuple,"
            , Indent 2 (Quote (ShowType f1))
            , Text "have to match."]]
    , [a2 :≁: b :⇛:
        VSep [Text "In the application of" :⊕: Quote (Text name) :◇: Comma
              :⊕: Text "the type of the #2 component of the tuple,"
            , Indent 2 (Quote (ShowType b))
            , Text "has to match the return type of the function of the #1 component,"
            , Indent 2 (Quote (ShowType f1))]]
    , [Check (IsParser p1)]
    ] ⇒ t1 → p2 → p1 a2
pFoldr :: PFoldrError "pFoldr"
pFoldr = Derived.pFoldr
pFoldr_ng :: PFoldrError "pFoldr_ng"
pFoldr_ng = Derived.pFoldr_ng
pFoldr1 :: PFoldrError "pFoldr1"
pFoldr1 = Derived.pFoldr1
pFoldr1_ng :: PFoldrError "pFoldr1_ng"
pFoldr1_ng = Derived.pFoldr1_ng
```

In an identical way we write customized error messages for folds with separation parsers.

```
type PFoldrSepError (name :: Symbol) = ∀ p1 p2 p3 p4 a a1 a2 a3 b f1 v t1.
  CustomErrors
    [ [IsNotOfParserKind name 2 p2 p1 a
      , IsNotOfParserKind name 3 p4 p3 v]
    , [IsNotAParser p1 :≁: False :⇛: ExpectedErrorMessage name 2 "a parser" p1
      , IsNotAParser p3 :≁: False :⇛: ExpectedErrorMessage name 3 "a parser" p3]
    , [p1  :≁: p3 :⇛: DifferentParsers name [(p1, 2), (p3, 3)]
      , t1 :≁: (f1, b) :⇛: ExpectedErrorMessage name 1 "a pair" t1]
    , [f1 :≁: (a1 → a2 → a3) :⇛: ExpectedErrorMessage name 1 "a function type as #1 component of the pair" t1]
    , [a1 :≁: v :⇛: VSep [Text "In the application of" :⊕: Quote (Text name) :◇: Comma
```

```
                  :⊕: Text "the type of the #1 argument of the function inside the tuple,"
                  , Indent 2 (Quote (ShowType f1))
                  , Text "has to match the underlying type of the parser of the #3 argument,"
                  , Indent 2 (Quote (ShowType v))]
    , a2 :≁: a3 :⇛: VSep [Text "In the application of" :⊕: Quote (Text name) :◇: Comma
      :⊕: Text "the second argument and return type of the function type inside the tuple,"
                  , Indent 2 (Quote (ShowType f1))
                  , Text "have to match."]]
  , [a2 :≁: b :⇛: VSep [Text "In the application of" :⊕: Quote (Text name) :◇: Comma
                  :⊕: Text "the #2 component of the tuple,"
                  , Indent 2 (Quote (ShowType b))
                  , Text "has to match the return type of the function of the #1 component."
                  , Indent 2 (Quote (ShowType f1))]]
  , [Check (IsParser p1)]
  ] ⇒ t1 → p2 → p4 → p1 b
pFoldrSep :: PFoldrSepError "pFoldrSep"
pFoldrSep = Derived.pFoldrSep

pFoldrSep_ng :: PFoldrSepError "pFoldrSep_ng"
pFoldrSep_ng = Derived.pFoldrSep_ng

pFoldr1Sep :: PFoldrSepError "pFoldr1Sep"
pFoldr1Sep = Derived.pFoldr1Sep

pFoldr1Sep_ng :: PFoldrSepError "pFoldr1Sep_ng"
pFoldr1Sep_ng = Derived.pFoldr1Sep_ng
```

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 ensure that at a specific point in the error message a type p is an instance of the class *IsParser*. Therefore, we must take additional measures to rule out cases we know do not belong to the class. Furthermore, this additional measurements poison the type of the combinator as no longer can be reduced to a type similar to the original one.

- The type signatures for customized error messages are insanely big because of all the cases that have to be accounted for.

- 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 (when this is wrong can lead to misleading).

- The type *ErrorMessage* provided by GHC does not work as smoothly as it should be. For example, printing a complicated type with *ShowType* 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 allows much better formatting of 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 its use. Moreover, due to the limited expressivity of the type error DSL an educated choice has to be made in order to give preference to some of the arguments to a function over the others.

- Sometimes debugging customized type error messages with kind errors can be cumbersome as the outputted error messages by GHC are just unreadable. 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 (name :: Symbol) (arg :: ℕ) (f :: ⋆) (n :: ℕ) =
  VCat [ Text "In the application of" :⊕: Quote (Text name) :⊕:
         Text ", is expected as #" :◇: ShowType arg :⊕:
         Text "argument a function type of" :⊕: ShowType n :⊕:
         Text "arguments but got" :◇: Colon
       , Indent 4 (ShowType f) :◇: Dot]
type FunctionTypeParser (name :: Symbol) (arg :: ℕ) (f :: ⋆) (n :: ℕ) =
  VCat [ Text "In the application of" :⊕: Quote (Text name) :⊕:
         Text ", is 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 (name :: Symbol) (arg :: ℕ) (f :: ⋆) (n :: ℕ) =
  VCat [ Text "In the application of" :⊕: Quote (Text name) :⊕:
           Text ", is 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
```

$$Map\ f\ [] = []$$
$$Map\ f\ (x : xs) = Apply\ f\ x : (Map\ f\ xs)$$