Pire - Pi-forall's Refactorer

Andreas Reuleaux

Table of Contents

Pi-forall 101 1
Pire: Source code and Installation 2
lib-pi-forall 3
building pi-forall as a library yourself 3
`PiForall` subdir/hierarchy 3
export more parsers 4
modified lib-pi-forall.cabal 5
testing lib-pi-forall 6
building Pire 6
testing your installation 7
unit tests 7
file tests 8
Pi-forall's Parsec parser 8
Backtracking in parsers and Pi-forall mimicry 14
Changes 19
concrete syntax and white space aware parsing, naming conventions 19
testing 19
t2s is handy when working on the command line 20
position decoration… 21
pretty printing 22
simple exprs 22

Pi-forall 101

Starting from just the bare minimum of language constructs:

```
a, A :=
  x             -- variables
  \x . a        -- lambda abstraction
  x b           -- function application
  (x: A) -> B   -- function type constructor (aka pi-types)
  x: Type       -- the type of types
```

Pi-forall is presented at various stages of increasing complexity in the course of Stephanie's discussion:

- More language constructs are added: type annotations, sigma types, propositional equality etc.
- The notion of (definitional) equality is refined from initially just alpha equivalence to beta-equivalence later.
- Constructors for new data types are introduced.

The language is a real dependently typed language:

- in that types are just expressions (by convention lower case letters are used for expressions, upper case letters for types)
- function type constructors (aka pi-types) are binders (the `x` in `(x: A) → B` can appear in `B`)
- any type has type `Type` (the "type in type"-axiom), a simple rule to make Pi-forall's type system consistent (but inconsistent as a logic, not taking into account different universe levels).

The language is somehow limited:

- in that there are nor checks for totality or termination,
- there is no means to execute Pi-forall programs currently
- there are no (proof) tactics .

At the same time some aspects of Pi-forall are relatively elaborated:

- the notion of beta-equivalence for definitional equality eg., as mentioned above.
- Variables can be marked as *erasable*, ie. needed only for type checking, not at runtime. (Those are different from *implicit arguments*: there aren't any implicit arguments allowed in Pi-forall).
- A simple module system is provided.

Pi-forall's type system is certainly its most interesting part and in the center of Stephanie's discussion: how the typing rules can be broken down into corresponding checking and inference rules, and are reflected in the implementation.

One aspect of Pi-forall's *implementation* should be mentioned as well however: heavy use is made of the *Unbound* library internally, also developed by Stephanie Weirich, together with Brent Yorgey, which handles many things automatically that otherwise would be laboursome to implement:

- binding of variables (by means of Church encodings)
- alpha equivalence
- finding free variables
- substitution, and more.

Pire: Source code and Installation

Installation instructions for Pire *as of December 2015 (tested w* ghc 7.8.4, and 7.10.3)/

You will need a patched version of Pi-forall, that provides *Pi-forall as a library*, before you can continue with *Pire*'s installation below.

There are two ways to obtain this patched version:

- get if from the pire website, as explained below: #lib_pi_forall, recommended, simple
- or patch it yourself, #build_pi_yourself

lib-pi-forall

- just visit the pire website at http://a-rx.info/pire, download the .tgz file there: http://a-rx.info/static/lib-pi-forall.tgz, and unpack it:

```
$ tar xvpzf lib-pi-forall.tgz
```

- continue either with #testing_lib_pi_forall (optional)
- or with *Pire*'s installation #building_pire

building pi-forall as a library yourself

Using the version from the pire website is the most straightforward approach, but patching Pi-forall yourself is not that difficult either:

Start by creating a copy of Pi-forall's full version (as found in its *full* directory):

```
$ git clone https://github.com/sweirich/pi-forall

as of December 2015, you will need a patched version, with changes for ghc 7.10.3:
$ git clone https://github.com/reuleaux/pi-forall

$ cd pi-forall

$ cp -rp full lib-pi-forall

$ cd lib-pi-forall
```

Now you will want to make these changes:

1. create a `PiForall` subdirectory/hierarchy, #subdir
2. make some more parser functions available for testing: #export_more_parsers
3. and in addition you will need a modified #lib-pi-forall.cabal file

`PiForall` subdir/hierarchy

The idea of creating a `PiForall` subdirectory/hierarchy is simple, really: all of Pi-forall's source files: `Syntax.hs`, `Parser.hs` etc. should live in `src/PiForall` (thus `src/PiForall/Syntax.hs` etc) rather than just in the `src` directory originally.

Thus move them there for starters (assuming that you are in the *lib* directory):

```
$ mkdir src/PiForall
$ mv src/*.hs src/PiForall
```

Now the import lists of these source files need to be adjusted, too: Instead of

```
import Syntax...
```

you will want

```
import PiForall.Syntax...
```

and likewise for the other imports, in each of Pi-forall's source files. This can be achieved easily in one go in the shell (assuming a Unix/Linux/Mac system here):

```
$ find ./src -name "*" | egrep "\.hs$" | xargs sed -i \
-e "s/ Syntax/ PiForall.Syntax/g" \
-e "s/ Parser/ PiForall.Parser/g" \
-e "s/ TypeCheck/ PiForall.TypeCheck/g" \
-e "s/ Main/ PiForall.Main/g" \
-e "s/ LayoutToken/ PiForall.LayoutToken/g" \
-e "s/ Equal/ PiForall.Equal/g" \
-e "s/ Environment/ PiForall.Environment/g" \
-e "s/ PrettyPrint/ PiForall.PrettyPrint/g" \
-e "s/ Modules/ PiForall.Modules/g" \
```

This is a single command pipeline in the shell, really, even though it is spread out over several lines here: you can just cut/copy these lines with your mouse, and paste them into your shell. Convince yourself that this command has succeeded by looking at one of the source files (`Parser.hs` eg.) at least.

export more parsers

Now you will want to make some more parsers available for testing:

Again, the idea is simpler than the command to achieve this: While the parser originally exports only three functions:

```
module Parser
  (
   parseModuleFile,
   parseModuleImports,
   parseExpr
  )
  where
```

we will want a much longer export list, like so:

```
module PiForall.Parser
  (
   parseModuleFile,
   parseModuleImports,
   parseExpr
 , arg
 , dconapp
 , expr
 , impOrExpVar
 , bconst
 , varOrCon
 , ifExpr
 , trustme
 , refl
 , typen
 , lambda
 , letExpr

 , sigDef
 , valDef
 , dataDef
```

```
 , decl

 , match
 , caseExpr
 , natenc
 , substExpr
 , expProdOrAnnotOrParens

 , constructorDef
 , telescope

 , whiteSpace

  )
  where
```

You could just edit the `Parser.hs` file by hand, and create this long export list.

Again, this can be achieved with a shell command instead:

```
$ sed -i '/,/{N; s/\n.*parseExpr/&\
   , arg\
   , dconapp\
   , expr\
\
   , bconst\
   , varOrCon\
   , ifExpr\
   , trustme\
   , refl\
   , typen\
   , lambda\
   , letExpr\
\
   , sigDef\
   , valDef\
   , dataDef\
   , decl\
\
   , match\
   , caseExpr\
   , natenc\
   , substExpr\
   , expProdOrAnnotOrParens\
\
   , constructorDef\
   , telescope\
\
   , whiteSpace\
/g; $!P ; $!D }' ./src/PiForall/Parser.hs\
```

modified lib-pi-forall.cabal

Last, but not least, you will need a modified `lib-pi-forall.cabal` file (in your *lib-pi-forall* directory): get it from the pire website: http://a-rx.info/pire

or just follow this link: http://a-rx.info/static/lib-pi-forall.cabal

Ie. even though you are building Pi-forall as a lib yourself, this one file you will want to download from the version already prepared on the pire website (I had to put it somewhere).

testing lib-pi-forall

The following steps of creating a local Pi-forall sandbox (in your *lib-pi-forall* directory) are not strictly necessary:

```
$ cabal sandbox init
$ cabal update

$ cabal install highlight-versions
$ cabal install --dry-run | highlight-versions
$ cabal install
```

(assuming `.cabal-sandbox/bin` is in your path, and thus `highlight-version` is found).

But they allow you to do some experimentation on the *cabal repl* command prompt: You should be able eg. to load `PiForall.Parser`, `PiForall.Main` etc

```
$ cabal repl
```

```
>>> :l PiForall.Parser
>>> parseExpr   "   \\x  .  a  "

>>> :l PiForall.Main
>>> goFilename "./test/Nat.pi"


>>> :l PiForall.Parser
>>> runFreshM (evalStateT (runParserT (do { whiteSpace; v <- expr; eof; return v}) []
 "<interactive>" " Succ n") emptyConstructorNames)
```

whereas later, from within *Pire*, you will be able to use *PiForall* only as a module

```
>>> :m +PiForall.Parser
```

or

```
>>> import qualified PiForall.Parser as P
```

(but not load it with `:l` any more, and thus only the functions explicitly exported will be available then).

building Pire

Now to install *Pire* itself:

```
$ git clone https://github.com/reuleaux/pire

$ cd pire

$ cabal sandbox init
```

tell cabal to use the modified lib-pi-forall.cabal file (or more general: the lib-pi-forall sources, adjust the path to your needs):

```
$ cabal sandbox add-source .../lib-pi-forall
```

maybe convince yourself that this step did work:

```
$ cabal sandbox list-sources
```

and proceed with the installation:

```
$ cabal update
$ cabal install highlight-versions
$ cabal install --dry-run | highlight-versions
```

and if happy with this dry run, do the real installation:

```
$ cabal install
```

If everything went fine, you should be able to use Pire on the cabal repl command line (try to access Pi-Forall's parser eg.):

```
$ cabal repl
```

```
>>> :m +PiForall.Parser
>>> parseExpr   "   \\x  .  a  "
```

or

```
>>> import PiForall.Parser as P
```

Load some Pire modules and play with them (the `P` module is just a laundry list of further imports, for convenience, *P* for *Pire*):

```
>>> :l P
>>> lam "x" $ V "z"
Lam "x" (Scope (V (F (V "z"))))
```

testing your installation

Last, but not least, run some tests, there are

1. unit tests: #utests
2. file tests: #filetests
3. doctests
4. roundtrip test

unit tests

```
$ cabal configure --enable-tests
$ cabal repl utest
```

load them with either

```
>>> :l tests/Tests.hs
```

or

```
>>> :l Tests
```

and run them

```
>>> main' "*"
```

or just:

```
>>> main
```

file tests

likewise for the file tests, eg.:

```
>>> :l tests/FileTests.hs
>>> main' "*"
```

todo: update the (simpler now) testing steps !

Pi-forall's Parsec parser

While Pi-forall's parser is arguably its least interesting part (and barely mentioned in Stephanie's lectures) parsing is the starting point for any serious implementation (and hence refactoring) work.

Moreover, as the absy parser developed here is merely a rewrite of Pi-forall's *Parsec* parser in *Trifecta*, some examples, of how this Parsec parser can be used, may be worth looking at, both as a Parsec recap, and to get in first touch with some Pi-forall code.

Say we are given a small Pi-forall input file Sample.pi with some function declarations:

```
-- -*- haskell -*-
-- Sample.pi

module Fac where

import Nat

-- data Nat : Type where
--   Zero
--   Succ of (Nat)

two : Nat
two = Succ (Succ Zero)

-- adapted from nat_eq in Nat
-- cf also http://mazzo.li/posts/AgdaSort.html

nat_leq : Nat -> Nat -> Bool
nat_leq = \ x y .
  case x of
     Zero -> True
     Succ m -> case y of
            Zero -> False
            Succ n -> nat_leq m n

fac: Nat -> Nat
fac = \n . if nat_leq n 0 then 1 else mult n (fac (minus n 1))
```

```
foo: Nat -> Nat
foo = \n. fac (mult 2 n)

bar: Nat -> Nat
bar = \n. let fac = ((\x . plus x 2) : Nat -> Nat) in plus (foo n) (fac (mult 3 n))
```

Never mind too much the details of those declarations at this point, just note a few points:

- The factorial function `fac` is defined the usual way, and requires the notion of natural numbers (Nat) together with some operations on them (`mult`, `minus`). Those we import from the Nat module, not shown here, but available with Pi-forall's source as well.
- We need so few things from Nat here, that we could have defined them ourselves, the commented out Nat data type definition gives a taste of what they'd look like, and indeed: doing so was useful when developing the Trifecta parser, as parsing a single file obviously is a easier than handling module imports.
- Pi-forall allows for Haskell-style comments.
- The remaining functions may serve as examples for some simple refactorings later.
- `nat_leq` (less or equal on naturals) and `two` exercise the Nat data type, and note how the `case` expression uses layout syntax (as in Haskell), ie. requires layout parsing.
- The very first `-- haskell --` line was just a means to get some (Haskell) syntax highlighting in Emacs for Pi-forall programs.

The usual way of interacting with Pi-forall is calling its `goFilename` function in Main.hs, which parses, typechecks, and pretty-prints the given Pi-forall source file, in the case of Sample.pi above:

```
>>> import PiForall.Main
>>> goFilename "samples/Sample.pi"
processing Sample.pi...
Parsing File "samples/Nat.pi"
Parsing File "samples/Sample.pi"
type checking...
Checking module "Nat"
Checking module "Fac"
module Fac where
import Nat
two : Nat
two = 2
nat_leq : Nat -> Nat -> Bool
nat_leq = \x y .
            case x of
              Zero -> True
              (Succ m) ->
                case y of
                  Zero -> False
                  (Succ n) -> nat_leq m n
fac : Nat -> Nat
fac = \n . if nat_leq n 0 then 1 else mult n (fac (minus n 1))
foo : Nat -> Nat
foo = \n . fac (mult 2 n)
bar : Nat -> Nat
bar = \n . let fac = \x . plus x 2 in plus (foo n) (fac (mult 3 n))
```

As can be seen:

- the output is quite readable and resembles the input, but all the comments are lost, as are the exact levels of indentation, newlines and parenthesis. The result is therefore of limited use for any refactoring efforts only, and this was the reason for developing another white space aware parser.
- Automatic conversions between different representations, like the one above for Nat from `Succ (Succ Zero)` to `2` in the function `two`, we don't want either for our refactoring purposes.
- The fact that our Sample.pi is also typechecked, cannot really be seen in the output above (except for the messages `type checking…` etc.), but is important for we don't want to concern ourselves with programs that are syntactically, but not type correct, and type checking gives us a means to ensure that.

While `goFilename` is a convenient way to parse, typecheck and pretty-print our Pi-forall code all in one go, sometimes we want more fine grained control, want to see intermediate results, the abstract syntax tree ie..

The following examples are meant as hints, how to "look under the hood" and discover more details, rather than as a complete description of the system.

> Tip
>
> The output of some of the following examples is very long, and has thus been shortened — not just by hand, but on the *ghci* (*cabal repl*) command line as well, to use them as doctests nevertheless. Having imported some modules initially:
>
> ```
> import PiForall.Modules
> import PiForall.PrettyPrint
> import Control.Monad.Except
> import Data.Either.Combinators
> ```
>
> we'd just write
>
> ```
> runExceptT (getModules ["samples"] "Sample.pi")
> ```
>
> eg., and see a list of modules being parsed, many lines of output ie., and we'd see the same list of modules, if we just return them after parsing:
>
> ```
> runExceptT (getModules ["samples"] "Sample.pi") >>= return
> ```
>
> We can get the result of the *Either* monad with *fromRight'* from the either package:
>
> ```
> runExceptT (getModules ["samples"] "Sample.pi") >>= return . fromRight'
> ```
>
> and we'd see the same output, if we *show* that list of modules and print the resulting *String*:
>
> ```
> runExceptT (getModules ["samples"] "Sample.pi") >>= putStrLn . show .
>  fromRight'
> ```
>
> even though, technically, we are looking at the *String* representation of the modules here. Taking this one step further, we can shorten the output to the first 400 characters:
>
> ```
> >>> runExceptT (getModules ["samples"] "Sample.pi") >>= putStrLn . ((+
> +"...") . take 400) . show . fromRight'
> Parsing File "samples/Nat.pi"
> Parsing File "samples/Sample.pi"
> ```

```
[Module {moduleName = "Nat", moduleImports = [], moduleEntries = [Data
 "Nat" Empty [ConstructorDef "samples/Nat.pi" (line 12, column 3) "Zero"
 Empty,ConstructorDef "samples/Nat.pi" (line 13, column 3) "Succ" (Cons
 Runtime _ (Pos "samples/Nat.pi" (line 13, column 12) (TCon "Nat" []))
 Empty)],Sig is_zero (Pos "samples/Nat.pi" (line 15, column 11) (Pi (<(_1,
{TCon "Nat" []})> TyBool))),Def is_zero (Po...
```

Adjust the following examples to your needs in this manner.

Assuming the module imports above, ie. initially just:

```
import PiForall.Modules
import PiForall.PrettyPrint
import Control.Monad.Except
import Data.Either.Combinators
```

you will need more imports in the course of the following examples (import them now, or later when needed):

```
import PiForall.Syntax
import PiForall.Parser
-- import Unbound.LocallyNameless
import Unbound.Generics.LocallyNameless
import Control.Monad.State.Lazy
import Text.Parsec
import qualified Data.Set as S
```

Given that our Sample.pi file imports the Nat module, we cannot use the parsing functions from Parser.hs directly, but only those from Module.hs (namely `getModules`). Cheating at the source code of `goFilename` we can get the abstract syntax tree of our Sample.pi file by running `getModules` in the *Control.Monad.Except* monad, like so:

```
>>> runExceptT (getModules ["samples"] "Sample.pi") >>= putStrLn . ((++"...") . take
 400) . show . fromRight'
Parsing File "samples/Nat.pi"
Parsing File "samples/Sample.pi"
[Module {moduleName = "Nat", moduleImports = [], moduleEntries = [Data "Nat" Empty
 [ConstructorDef "samples/Nat.pi" (line 12, column 3) "Zero" Empty,ConstructorDef
 "samples/Nat.pi" (line 13, column 3) "Succ" (Cons Runtime _ (Pos "samples/Nat.pi" (line
 13, column 12) (TCon "Nat" [])) Empty)],Sig is_zero (Pos "samples/Nat.pi" (line 15,
 column 11) (Pi (<(_1,{TCon "Nat" []})> TyBool))),Def is_zero (Po...
```

The result is a list of modules, really: Nat.pi and Sample.pi, in the Either monad. Normally we'd be interested only in the *last* module parsed (Sample.pi ie. here, and again, only the beginning of the output is shown):

```
>>> runExceptT (getModules ["samples"] "Sample.pi") >>= putStrLn . ((++"...") . take
 400) . show . last . fromRight'
Parsing File "samples/Nat.pi"
Parsing File "samples/Sample.pi"
Module {moduleName = "Fac", moduleImports = [ModuleImport "Nat"], moduleEntries =
 [Sig two (Pos "samples/Sample.pi" (line 13, column 7) (TCon "Nat" [])),Def two (Pos
 "samples/Sample.pi" (line 14, column 7) (DCon "Succ" [Arg Runtime (Paren (Pos "samples/
Sample.pi" (line 14, column 13) (DCon "Succ" [Arg Runtime (DCon "Zero" [] (Annot
 Nothing))] (Annot Nothing)))))] (Annot Nothing))),Sig nat_leq (Pos ...
```

Now we can use *disp* for pretty printing the result:

```
>>> runExceptT (getModules ["samples"] "Sample.pi") >>= return . disp . last . fromRight'
```

```
Parsing File "samples/Nat.pi"
Parsing File "samples/Sample.pi"
module Fac where
import Nat
two : Nat
two = 2
nat_leq : Nat -> Nat -> Bool
nat_leq = \x y .
            case x of
              Zero -> True
              (Succ m) ->
                case y of
                  Zero -> False
                  (Succ n) -> nat_leq m n
fac : Nat -> Nat
fac = \n . if nat_leq n 0 then 1 else mult n ((fac ((minus n 1))))
foo : Nat -> Nat
foo = \n . fac ((mult 2 n))
bar : Nat -> Nat
bar = \n .
        let fac = ((\x . plus x 2) : Nat -> Nat) in
        plus ((foo n)) ((fac ((mult 3 n))))
```

If we were looking at the contents of just a single self-contained file, without any imports of other modules ie., say at a file Simple.pi:

```
-- Simple.pi


-- fromSuccess  `liftM` parseFromFileEx (runInnerParser $ evalStateT (whiteSpace *> many
 decl) piInit) "samples/Simple.pi"
-- resp
-- parseFromFileEx (runInnerParser $ evalStateT (whiteSpace *> many decl) piInit)
 "samples/Simple.pi" >>= return . fromSuccess
-- parseFromFileEx (runInnerParser $ evalStateT (whiteSpace *> many decl) piInit)
 "samples/Simple.pi" >>= return . nopos . fromSuccess



module Fac where


-- caseExpr w/ prelude
-- case x of
--    Zero -> True
--    Succ m -> False


-- caseExpr w/ prelude
-- case y of
--     Zero -> False
--     Succ n -> nat_leq m n



-- caseExpr w/ prelude
-- case x of
--    Zero -> True
--    Succ m -> case y of
--           Zero -> False
--           Succ n -> nat_leq m n
```

```
-- lambdaPAs w/ prelude
-- \ x y .
--   case x of
--      Zero -> True
--      Succ m -> case y of
--             Zero -> False
--             Succ n -> nat_leq m n



data Nat : Type where
  Zero
  Succ of (Nat)


two : Nat
two = Succ (Succ Zero)


nat_leq : Nat -> Nat -> Bool
nat_leq = \ x y .
  case x of
     Zero -> True
     Succ m -> case y of
            Zero -> False
            Succ n -> nat_leq m n


fac: Nat -> Nat
fac = \n . if nat_leq n 0 then 1 else mult n (fac (minus n 1))
```

then we could use `parseModuleFile` from Parser.hs directly. (We need to provide `emptyConstructorNames` from Syntax.hs, so that the parser can collect information about data types and constructors found (Nat, Zero, Succ,…) running in the State monad internally):

```
>>> runExceptT (parseModuleFile  emptyConstructorNames  "samples/Simple.pi") >>= return .
 disp . fromRight'
Parsing File "samples/Simple.pi"
module Fac where
data Nat : Type where
  Zero
  Succ of (_ : Nat)
two : Nat
two = 2
nat_leq : Nat -> Nat -> Bool
nat_leq = \x y .
            case x of
              Zero -> True
              (Succ m) ->
                case y of
                  Zero -> False
                  (Succ n) -> nat_leq m n
fac : Nat -> Nat
fac = \n . if nat_leq n 0 then 1 else mult n ((fac ((minus n 1))))
```

The situation is less complicated (does not require any file I/O) if we just want to parse a string as an expression:

```
>>> parseExpr   "   \\x  .  a  "
Right (Pos "<interactive>" (line 1, column 4) (Lam (<(x,{Annot Nothing})> Pos
 "<interactive>" (line 1, column 11) (Var a))))
>>> fromRight' $ parseExpr   "   \\x  .  a  "
```

```
Pos "<interactive>" (line 1, column 4) (Lam (<(x,{Annot Nothing})> Pos
 "<interactive>" (line 1, column 11) (Var a)))
```

Again, pretty printing is convenient for reading the result:

```
>>> disp $ fromRight' $ parseExpr   "   \\x  .  a  "
\x . a
```

Parser.hs exports only `parseModuleFile` and `parseExpr`, but we can exercise the individual building blocks of the parser of course, by loading Parser.hs directly (the import as above, assumed already): ie. we can parse expressions, declarations, signature definitions, variables etc. (shown is parsing an expression):

```
>>> runFreshM (evalStateT (runParserT (do { whiteSpace; v <- expr; eof; return v}) []
 "<interactive>" " Succ n") emptyConstructorNames)
Right (Pos "<interactive>" (line 1, column 2) (App (Var Succ) (Var n)))
```

This time we have to "unpeel the onion" of our monad transformer stack ourselves: the aforementioned *StateT* keeps track of constructor names, and *FreshM* from unbound is responsible for names and variable bindings.

Note how, without any further knowledge of *Succ*, *Succ n* is just parsed as a function application. This may not be what we want: An admittedly ad-hoc `prelude` helps recognising *Succ* as a data constructor rather than as a regular function:

```
>>> let prelude = ConstructorNames (S.fromList ["Nat"]) (S.fromList ["Zero", "Succ"])
>>> runFreshM (evalStateT (runParserT (do { whiteSpace; v <- expr; eof; return v}) []
 "<interactive>" " Succ n") prelude)
Right (Pos "<interactive>" (line 1, column 2) (DCon "Succ" [Arg Runtime (Var n)] (Annot
 Nothing)))
```

Backtracking in parsers and Pi-forall mimicry

> This chapter describes some implementation details: how Pire's Trifacta parser behaved differently from Pi-forall's Parsec parser originally, and how it mimics Pi-forall/Parsec's exact behaviour now.
>
> These details may not be terribly important for understanding Pire's approach to refactorings, but maybe they are interesting as a bug hunting experience (if one may call Pire/Trifecta's original behaviour a bug), and they should give some insight, what's happening under the hood.

I have solved the problem of parsing `A→BA→B` (and similar more complicated cases, cf. the examples below) in exactly the same manner as Pi-forall does.

To make a long story short: All of Pi-forall's test files parse fine now (comparing the result of `untie $ parse …` with Pi-forall's parser), as covered by the tests.

tl;dr

Strictly speaking: yes, maybe it's not that important, if the fresh variables used read as

> `_, _1, _2, _3, …` - as always in Pire's Trifecta parser's case (so far)

or as in Pi-forall/Parsec's case:

```
sometimes as   _, _1, _2, _3, …
at other times as _2, _3, _4, …
or _4, _5, _6, …
but also as _16, _17, _18, …
etc.
```

depending on the level of parens / number of arrows etc.

Nevertheless, I feel much more comfortable now, that I have found a way to make my Trifecta parser behave exactly in Pi-forall's manner (*Pi-forall mimicry*), and understand what's going on.

I can thus continue to be very strict in my comparisons of results: *parse* & *untie* vs. Pi-forall's original parser, and would hope that this way, other possible subtle differences / future issues won't go unnoticed.

A few examples may be worth looking at (and yes, there is a system to this, it's just not obvious, how it works `:-)`):

```
> import PiForall.Parser as P
> import Pire.NoPos
> import Data.Either.Combinators (fromRight')
> nopos $ fromRight' $ P.parseExpr "A->B"
Pi (<(_,{Var A})> Var B)
> nopos $ fromRight' $ P.parseExpr "A->B->C"
Pi (<(_,{Var A})> Pi (<(_1,{Var B})> Var C))
> nopos $ fromRight' $ P.parseExpr "A->B->C->D"
Pi (<(_,{Var A})> Pi (<(_1,{Var B})> Pi (<(_2,{Var C})> Var D)))
> nopos $ fromRight' $ P.parseExpr "A->B->C->D->E"
Pi (<(_,{Var A})> Pi (<(_1,{Var B})> Pi (<(_2,{Var C})> Pi (<(_3,{Var D})> Var E))))
> nopos $ fromRight' $ P.parseExpr "(A->B)"
Paren (Pi (<(_,{Var A})> Var B))
> nopos $ fromRight' $ P.parseExpr "(A->B->C)"
Paren (Pi (<(_,{Var A})> Pi (<(_1,{Var B})> Var C)))
> nopos $ fromRight' $ P.parseExpr "(A->B->C->D)"
Paren (Pi (<(_,{Var A})> Pi (<(_1,{Var B})> Pi (<(_2,{Var C})> Var D))))
> nopos $ fromRight' $ P.parseExpr "(A->B->C->D->E)"
Paren (Pi (<(_,{Var A})> Pi (<(_1,{Var B})> Pi (<(_2,{Var C})> Pi
(<(_3,{Var D})> Var E)))))
> nopos $ fromRight' $ P.parseExpr "((A->B))"
Paren (Paren (Pi (<(_2,{Var A})> Var B)))
> nopos $ fromRight' $ P.parseExpr "((A->B->C))"
Paren (Paren (Pi (<(_4,{Var A})> Pi (<(_5,{Var B})> Var C))))
> nopos $ fromRight' $ P.parseExpr "((A->B->C->D))"
Paren (Paren (Pi (<(_6,{Var A})> Pi (<(_7,{Var B})> Pi (<(_8,{Var C})> Var D)))))
> nopos $ fromRight' $ P.parseExpr "((A->B->C->D->E))"
Paren (Paren (Pi (<(_8,{Var A})> Pi (<(_9,{Var B})> Pi (<(_10,{Var
C})> Pi (<(_11,{Var D})> Var E))))))
> nopos $ fromRight' $ P.parseExpr "(((A->B)))"
Paren (Paren (Paren (Pi (<(_8,{Var A})> Var B))))
> nopos $ fromRight' $ P.parseExpr "(((A->B->C)))"
Paren (Paren (Paren (Pi (<(_16,{Var A})> Pi (<(_17,{Var B})> Var C)))))
> nopos $ fromRight' $ P.parseExpr "(((A->B->C->D)))"
Paren (Paren (Paren (Pi (<(_24,{Var A})> Pi (<(_25,{Var B})> Pi
(<(_26,{Var C})> Var D))))))
> nopos $ fromRight' $ P.parseExpr "(((A->B->C->D->E)))"
Paren (Paren (Paren (Pi (<(_32,{Var A})> Pi (<(_33,{Var B})> Pi
(<(_34,{Var C})> Pi (<(_35,{Var D})> Var E)))))))
> nopos $ fromRight' $ P.parseExpr "((((A->B))))"
Paren (Paren (Paren (Paren (Pi (<(_26,{Var A})> Var B)))))
> nopos $ fromRight' $ P.parseExpr "((((A->B->C))))"
Paren (Paren (Paren (Paren (Pi (<(_52,{Var A})> Pi (<(_53,{Var B})> Var C))))))
```

```
> nopos $ fromRight' $ P.parseExpr "((((A->B->C->D))))"
Paren (Paren (Paren (Paren (Pi (<(_78,{Var A})> Pi (<(_79,{Var B})> Pi
(<(_80,{Var C})> Var D)))))))
>
```

etc.

I started out reducing Pire's parser and later Pi-forall's parser to the bare minimum of functions that would still show the above behaviour (or lack thereof in Pire's case initially), ie. got rid of *decls*, and lots more stuff.

Then I added some writing capabilities to Pire's parser: ran it in the *RWS (Reader-Writer-State)* monad, rather than just in the *State* monad, which allowed me sprinkle some *tell me*'s in the code, and also I added some increment function calls here and there (that would increment the state's counter for fresh variables).

I expirmented with commenting out some of the code, eg. choices of the grammar as well.

That way I found that one particular *choice* function within the local function def of *beforeBinder* in *expProdOrAnnotOrParens* was at the culprit, originally:

```
beforeBinder = parens $
  choice [do e1 <- try (term >>= (\e1 -> colon >> return e1))
             e2 <- expr
             return $ Colon' e1 e2
         , do e1 <- try (term >>= (\e1 -> comma >> return e1))
              e2 <- expr
              return $ Comma' e1 e2
         , Nope <$> expr]
```

I could change Pire's parsing behaviour by adding some *increments* and *tell me*'s (it just wouldn't behave like Pi-forall's parser still), like so:

```
choice [
  do
    tell "in colon branch\n"
    e1 <- try (term >>= (\e1 -> colon >> return e1))
    e2 <- expr
    return $ Colon' e1 e2
, do
   tell "in comma branch\n"
   e1 <- try (term >>= (\e1 -> comma >> return e1))
   e2 <- expr
   return $ Comma' e1 e2
, do
    tell "in comma branch\n"
    increment
   , Nope <$> expr

]
```

Likewise commenting out eg. the middle comma choice above (and in Pi-forall's parser) had an impact in terms of fresh variables' choice.

I was still far from real Pi-forall mimicry, no matter what I tried: where I put my increment function calls etc.: I couldn't make Pire's parser behave in Pi-forall's way. But at least that got me on the right track for understanding what's going on:

Pi-forall's parser is defined as:

```
type LParser a = ParsecT
                   String                      -- The input is a sequence of Char
                   [Column] (                  -- The internal state for Layout tabs
                      StateT ConstructorNames
                      FreshM)                  -- The internal state for generating
 fresh names,
                   a                           -- the type of the object being parsed
```

Never mind the details, just notice that that the monad transformer stack is:

ParsecT(…(StateT…(FreshM))

with *FreshM* within *ParsecT*. It is possible to use Parsec the other way around (*Parsec* inside *StateT*)

StateT… Parsec

it's just not done that way in Pi-forall, and the two behave differently:

the *StateT … Parsec* does backtracking for the state in case a choice fails,

the *ParsecT(…(StateT…(FreshM))* does not do (cannot do) any backtracking, cf. eg.Roman Cheplyaka' blog (author of tasty/tasty-hunit etc, hey!): https://ro-che.info/articles/2012-01-02-composing-monads

The trifecta-parser is written in this later style:

StateT …(inner trifecta parser)

and does backtracking. I was happy with the parser so far, (this is the style of the *Idris*' parser), and if I wouldn't want to mimic Pi-forall's behaviour, my choice of fresh vars seemed more natural.

Trifecta parsers cannot be written in a monad transformer (*TrifectaT*) style (not that I am aware of: it is not in the docs, I have never seen a Trifecta parser as a monad transformer wrapping a state, it's always the state wrapping the trifecta parser).

A reduced to the bare minimum Pi-forall grammar shows what's going on:

```
expr ->
        -- pi-type, right-associativity with the help of buildExpressionParser table
        term "->" term

        | term


term ->
        -- App
        factor @: factor

        | factor


factor -> var

          -- annotation Ann
          | "(" term ":" expr ")"

          -- Prod
          | "(" term "," expr ")"

          -- Paren
```

```
          | "(" expr ")"
```

Now see, what happens when we parse `A→BA→B` as an *expr*: well, we must find a *term*, and in turn a *factor* then:

Oh, it starts with `(`, cannot just be a *var* then, but must be one of the other cases, so try them (in this order):

```
| "(" term ":" expr ")"
| "(" term "," expr ")"
| "(" expr ")"
```

so have a look what's inside the outer parens: `(A→B)`, and try to parse that as a term: well that's possible: `(A→B)` is a term (therby using some fresh vars for the pi-type found), it's just not followed by a `:`, so put back

> in the trifecta case: backtracking clears the state as if we had never used any fresh vars for the Pi-type found
>
> in Pi-forall's case: the fresh vars for the term / Pi-type `(A→B)` are used forever.

try next possibility, similar: parsing `(A→B)` as a term succeeds (und renders fresh vars used in the case of Pi-forall), just not followed by `,`

so finally go for the last case: `(` *expr* `)`.

This also explains, why longer arrow chains `A→B→C→DA→B→C→D` make a difference: use `_, _1, _2` in the first attempt of parsing the inner `(A→B→C→D)`, `_3, _4, _5` in the second attempt, and finally return `_6, _7, _8`

So how could I mimic Pi-forall's behaviour?

Well the solution was to rely on *lookAhead*:

```
choice [do { e1 <- try (term >>= (\e1 -> colon >> return e1)) ;
             e2 <- expr ;
             return $ Colon' e1 e2
           }
        , do { lookAhead term ;
               e1 <- try (term >>= (\e1 -> comma >> return e1)) ;
               e2 <- expr ;
               return $ Comma' e1 e2
             }
        , do { lookAhead term ;
               lookAhead term ;
               Nope <$> expr
             }
        ]
```

If we are in the second (comma) case, we must already have tried the first case (and succeeded with term): *lookAhead* doesn't consume the input, put triggers the state's counter increment (as if no backtracking had happened)

Likewise: if we are in the third case: we must have looked at the other two case already (and found a term each time): thus call *lookAhead* twice.
OK, I am very glad, I got this sorted out, and will continue with the finishing touches for a new Pire release, soon, and turn my attention to the refactorings.

Changes

complete rewrite as of December 2017, summary of major changes and some design decisions (ordered by topic), recent and older ones:

- Pire is based on Edward Kmett's Bound library instead of Unbound, used in Pi-forall
- While an older version of Pire included Pi-forall as a library, the current one does not.

  It was used for testing Pire's parser against Pi-forall's original parser at the time, and will possibly be reintroduced again.

  Thus functions to translate between the various syntax representations, like `untie` have disappeared as well for now.

concrete syntax and white space aware parsing, naming conventions

- For every abstract syntax expression (`If`, say), a white space aware version (`If_`) was given in the past, covering not only the abstract syntax essentials, but also any white in between. Necessarily these expressions (declarations, modules etc.) were complicated:

  While `If` was defined as

  ```
  If (Expr t a) (Expr t a) (Expr t a) (Annot t a)
  ```

  eg., its white space aware counterpart `If_` was

  ```
  If_ (Token 'IfTokTy t) (Expr t a) (Token 'ThenTokTy t) (Expr t a) (Token 'ElseTokTy t)
   (Expr t a) (Annot t a)
  ```

  each of the token and subexpressions allowing for trailing white space.

  `expr_` was the parser of these white space aware expressions (and still is), `expr` being the ordinary abstract syntax tree parser.

  This naming convention applies to module names as well: `Modules_` eg. is the white space aware version of `Modules`, with some exceptions: `WsAwareParser` vs. `Parser` notably.

  These white space aware expression (like `If_` above) turned out

- likewise nopos (no position) operates on $\text{Absy}_{\text{Bound}}$ (but on $\text{ConcreteSyntax}_{\text{Bound}}$ and $\text{Absy}_{\text{Unbound}}$ as well with the help of type classes): get rid of position information to make the parsers output more human readable

testing

- tasty/hunit test available in Pire/Tests.hs and Pire/FileTests.hs There are currently 340 tests in Pire/Tests.hs and 46 tests in FileTests.hs to provide compatibility with Pi-forall
- doctests in most of the source files, run them with

  ```
  $ doctest -isrc src/Pire/Parser/Basic.hs
  ```

```
$ doctest -isrc Pire.Parser.Basic
$ doctest -isrc src/Pire/Parser/*
```

eg.

- The unit tests have been moved from the src directory to their own tests directory, and are treated as a (cabal) test-suite utests, use them eg. with:

```
$ cabal configure --enable-tests
$ cabal repl utests
```

```
>>> :l tests/Tests.hs
>>> main' "*"
>>> :l tests/FileTests.hs
>>> main' "*"
```

- testing has been improved: the doctests, while in place for a while already, usage like this:

```
$ find ./src -iname "*.hs" | egrep -v -i "keepme|whatever" | xargs doctest -isrc
```

can now be called from a separate `doctests.hs` program in the tests subdir, and are part of the the cabal test-suite, they can be called togehter with:

```
$ cabal configure --enable-tests
$ cabal build
$ cabal test
```

Three tests currently: unit tests (utests), file tests (filetests), and doctests. OK, more docs to come, how to use them, but these should go in the docs section, not the changes section.

- a kind of literate programming for the docs introduced (with special `// doctest` … `// /doctest` comments, more details in the docs/README [https://github.com/reuleaux/pire/blob/master/docs/README.adoc#doctest_snippets], ie. the other way around as extracting code snippets, as recently introduced), the idea is to have readable adoc files, with examples doctested nevertheless, cf. eg. the Parsec chapter [parsec.xml#parsec_parsers]

- The individual tests in Tests.hs pointing to precise features/requirements each, were easier to handle in case of failures, than the relatively complex file tests in FileTests.hs.

  Thus more individual tests were added: there are 468 of them now (plus the 57 file tests).

t2s is handy when working on the command line

- Working on the command line it seems most natural to work with Strings, like so

```
>>> V "a" :@ V "b"
...
>>> lam "x" $ V "a" :@ V "b"
...
>>> Ws_ (V "b") $ Ws "{-foo-}"
...
```

These are examples of String expressions

```
>>> :t Ws_ (V "b") $ Ws "{-foo-}"
Ws_ (V "b") $ Ws "{-foo-}" :: Expr [Char] [Char]
```

but the parser yields Text expressions, eg.

```
>>> nopos $ parse lambda " \\ x . a"
Lam "x" (Scope (V (F (V "a"))))
>>> :t nopos $ parse lambda " \\ x . a"
nopos $ parse lambda " \\ x . a" :: Pire.Syntax.Ex.Ex
```

```
>>> t2s $ nopos $ parse lambda " \\ x . a"
Lam "x" (Scope (V (F (V "a"))))
>>> :t t2s $ nopos $ parse lambda " \\ x . a"
t2s $ nopos $ parse lambda " \\ x . a" :: Expr String String
```

Yes, one *can* use `:set -XOverloadedStrings` on the ghci (cabal repl) command prompt, but this only so good: often enough, it's not clear what is meant by `"foo"` then: the String, the Text?

- This t2s conversion one wants to do with the least effort of course, and cheating at the *ermine* source code (as for many things *Bound* related as well), *bitraverse* (*Bitraversable* etc) comes in handy: once expressed, how to bitraverse the expression tree in general, the String to Text conversion of Expr t a becomes just a bitraversal with T.unpack for both: its t and a parameters.

position decoration…

not currently implemented…

- can use bitraversal to decorate the syntax tree with (very fine grained) position information: ie. for every identifier, token, white space etc, all this after parsing. (independent of the (Pos …) info wrapped around some exprs by the parser: didn't want to completely change the parser again, and found this position info used too sparingly)

  had to change tokens for this to work: every token keeps track of the string (text) parsed, in order to participate in this decoration with position info

  likewise had to reintroduce white space aware constructors for Type_, TyBool_ etc, had removed them just recently (oh well!), because I thought I could save them with Ws_ (Type t) Ws eg., but now we need not only the white space after the Type, but also the string (text) "Type" itself, thus Type_ t (Ws t), ie. the t in the middle.

  anyway, happy with the position decoration, watch out for bound names, though (some examples in Decorate.hs). This will hopefully not be a problem as we need to open binders (instantiate lambdas etc), as we navigate the tree.

  beginning navigation in the concrete syntax tree (parsed w/ white space), more to come, likewise more explanations, examples.

- *uwrap* introduced in addition to *wrap*. *unwrap* can be simpler (rely on hoistScope ie.), recall: wrap allows bound vars to be wrapped w/ their textual representation, for they can contribute to the calculation of the position info)

- further groundwork for navigation in the white space aware syntax tree, ie. functions/classes/general approuch in `Decorate.hs` and `Refactor.hs` refined, to be continued.

- position info, ranges, and navigation in the ws aware syntax tree, and refactoring split into pieces and moved to Refactor/*.hs

- Text2String supports modules now
- can use bifoldMap/foldMap on the decorated syntax tree (with position info ie.) to calculate ranges (similar in spirit to bitraversal), range calculation thus simplified
- first steps of walking the (ws aware) tree by line column number
- managed to even traverse ConstructorNames (use them as an applicative), by means of a poor man's traversal on sets. decorateM thus returns a module now (the old tuple version is still available as decorateM'), decorated modules can thus be treated as any other pieces of the zipper
- *Decorated* typeclass and instances introduced, and constraints for /Range/s simplified that way.

pretty printing

- made pretty printing work on T.Text only, to simplify matters, but provided printing of String Exprs/Decls/Modules etc by means of Text2String.

simple exprs

- white space aware versions of the SimpleExpr parsers introduced as well: `simple_expr`, `simple_term` etc
- the Expr constructor Pos renamed to Position, so Pos could be used for the constructor of position info as calculated / decorated (formerly LineColumn), just this one constructor now (ie. From, To, AnyPos merged to Pos)