pcl: A Robust Monadic Parser Combinator Library for OCaml

Chris Casinghino

ccasin@seas.upenn.edu

OCaml Summer Project Meeting

Outline

Outline

Benefits of Parser Combinators

- Parsers are specified directly in target language.
  - No need to learn seperate grammar specification language.
  - Parser may be modified on the fly, no seperate building process.
  - Parsers are typechecked.
- Easy to extend set of available combinators.
- Lexer and parser may be written with the same tool.

Parsec: Parser Combinators for Haskell

- **Parsec** is a popular monadic parser combinator library for Haskell.
  - Parsec has been around for years, and is widely used.
  - Large combinator library, including parsers for lexing, expressions, etc.
  - Expressive error reporting, efficient.
- pcl is, in large part, a port of Parsec to OCaml.
  - This presents challenges - Parsec makes use of monads, laziness.
  - Also added some new features.

Outline

The Basic Combinator Building Blocks

- Think of `'a parser` as an abstract type - an `'a parser` can be run on input, yielding an `'a`, or it can be combined into other parsers
- **val** `return : 'a -> 'a parser`
  - `return v` is a parser which consumes no input, and succeeds with the result `v`.
- **val** `(>>=) : 'a parser -> ('a -> 'b parser) -> 'b parser`
  - `p >>= f` is a parser which first parses a `p`, evaluates `f` on the result, and then applies the created parser.
  - `>>=` is called “bind”.

Bind and Return, An Example

- As an example, suppose we have two parsers, one that returns `char` and one that returns `int`.
  - **val** `p : char parser`
  - **val** `q : int parser`
- From these, we can build a new parser which parsers a p, then a q, and returns both:
  - **val** `pq : char*int parser`
  - **let** `pq =`

```
        p
>>= fun r1 -> q
>>= fun r2 -> return (r1,r2)
```

Two More Primitives

- **val** `mzero : 'a parser`
  - `mzero` is a parser which fails, consuming no input.
- **val** `any : char parser`
  - any is a parser which returns the first token of input (unless you are at EOF).
- Example:
  - **val** `satisfy : (char -> bool) -> char parser`
  - **let** `satisfy pred =`

```
        any
>>= fun c -> if pred c then return c
                       else mzero
```

Using Satisfy

- `satisfy` can be used to create many useful parsers.
- **val** `lower, upper, digit : char parser`
  - **let** `lower = satisfy (fun c -> c >= 'a' && c <= 'z')`
  - **let** `upper = satisfy (fun c -> c >= 'A' && c <= 'Z')`
  - **let** `digit = satisfy (fun c -> c >= '0' && c <= '9')`

The Choice Combinator

- Suppose we want to parse either one of two distinct things.
- **val** `(<|>) : 'a parser -> 'a parser -> 'a parser`
  - `p <|> q` tries the parser p, if it succeeds, that result is returned, if it fails (without consuming input), the parser q is used
- Examples: `alpha, alphaNum : char parser`
  - **let** `alpha = lower <|> upper`
  - **let** `alphaNum = alpha <|> digit`

The Many Combinator

- One common need is to parse several of a particular item in sequence.
- `many p` parses zero or more occurances of the parser `p`
- **val** `many : 'a parser -> 'a list parser`
- **let rec** `many p =`

```
    (     p
     >>= fun r -> many p
     >>= fun rs -> return (r::rs))
<|> (return [])
```

Putting it Together: The Ident Parser

- Many programming languages define identifiers to be strings of alphanumerical characters, beginning with a letter.
- Suppose we have a function
  val `implode : char list -> string`
- We can write a parser for identifiers:
  - **val** `ident : string parser`
  - **let** `ident =`

```
      alpha
>>= fun c -> many alphaNum
>>= fun cs -> return (implode (c::cs))
```

Outline

Laziness

- Parsec makes use of Haskell's laziness at several key points
- One is input, don't have to read the whole input stream into memory at once.
  - This presents a problem - OCaml has lazy input streams, but they are destructive, which is a problem for the backtracking nature of parsing
  - To solve this, I implemented a simple non-destructive LazyList structure on top of OCaml streams

Laziness continued

- Another problem is <|> - we need partial evaluation to discard unused branches
- Parsec uses a type constructor which indicates whether data is consumed, and counts on Haskell's laziness to delay the computation of the actual result.
- In pcl, we use a similar datatype, but with thunks and OCaml's `lazy` keyword in the primitive token recognizers

String Parsers

- Parsers which return a `String` are easy in Haskell
  - Since `String = [Char]`, the `many` parser and others are already returning strings
- In OCaml, converting a `char list` to a `string` is linear time.
  - Parsing directly into a `string` or `array` isn't cheap when we don't know how long the result will be, since we must resize.
  - Imploding at the end costs about the same as the resize overhead.
  - Luckily, for most common cases, this expense isn't overwhelming, but it does slow things down a little.

New Features in pcl

- In addition to adding a number of new combinators, pcl generalizes Parsec's expression module.
- Parsec supports prefix, postfix, and infix operators in expression parsers.
- But sometimes this isn't enough, consider OCaml's `match <exp> with <cases>` construction.
- pcl adds support for expressions surrounded by arbitrary other parsers.

References

Graham Hutton and Erik Meijer
Monadic Parser Combinators
Department of Computer Science, University of Nottingham, NOTTCS-TR-96-4, 1996.

Daan Leijen and Erik Meijer
Parsec: Direct Style Monadic Parser Combinators for the Real World
Department of Computer Science, Universiteit Utrecht, UU-CS-2001-27, 2001.