Combinator Libraries

Orphi the AweKid

18th June 2009

Abstract

Mainstream programming languages have a rich set of well-known and widely used idioms for solving particular problems. These need no further documentation here. The *Haskell* programming language is radically different to any mainstream language, and it has its own radically different way of approaching problems. However, since Haskell is an rare and obscure language, its idioms are not widely known in the mainstream community. This document gives an informal description of one very common Haskell idiom: the construction of *combinator libraries*. No knowledge of Haskell is assumed.

1 Introduction

The majority of mainstream programming languages today are very similar. Many of them are object-oriented, and those that are not are usually at least block-structured. The same language constructs appear in each of these languages, and learning a new one is typically a simple matter of learning some new syntax, and the particular mix of features the language provides. (E.g., reflection, dynamic loading, built-in data types, etc.)

As a result, all of these languages are typically used in broadly similar ways. A common set of techniques and idioms is well-known to most mainstream programmers, regardless of the particular language in question. For example, implementing an abstract base class featuring many concrete methods that call a small set of abstract ones is so common as to be almost "the definition" of how to use object-oriented languages.

Haskell, on the other hand, is a rather rare and obscure programming language, with a *radically* different design. Programs are defined as sets of transformation rules rather than ordered lists of commands, familiar constructs such as loops are absent from the language, the syntax does not remotely resemble anything mainstream, and it is possible to construct *programs* out of *type signatures!*

As a result of this radical design, the language is also *used* in a radically different way, resulting in a very different set of techniques and idioms. Put simply, Haskell programmers approach problems from a wholely different angle. While whole tomes could be filled with a discussion of this difference, this document focuses on a smaller task: presenting a simple, common Haskell idiom to a wider audience who may not be familiar with it.

Our subject for this document is an imformal exploration of *combinator libraries*. We will briefly touch on the much-feared concept of *monads*. The main body of this document is section 2, which contains a large example of a combinator library. Section 3 provides a brief overview of a

second example. No knowledge of Haskell is required, and all examples are presented as psuedo-code resembling a mainstream programming language.

2 Building parsers

2.1 Introduction

By way of an example, let us consider the problem of building a text parsing system.

At this point, a typical programmer using an OOP language is thinking about designing a "parser" object and how to store the current state of the parser, and what methods should be provided to update the parser's state, and maybe the parser operates as a finite state machine...

We however are interested in the Haskell approach. Parser libraries are an absolutely *classic* application for combinator technology in Haskell — and indeed, the Glasgow Haskell Compiler ships with such a library, named *Parsec*. Our description roughly follows the outline of this library (although many function names are changed to help readability).

2.2 What is a combinator?

Briefly, a *combinator library* is generally structured as follows:

- A (very) small selection of (very) simple "things" are provided.
- A selection of *combinators* enable the "things" to be *combined* in various ways to yield more complex "things".
- Often a number of "shortcuts" are also provided for items which are either commonly required or non-obvious to construct.

Despite the simplicity of this approach, it can be a very powerful way to work, as we will shortly see.

2.3 What is a parser?

Parsec treats parsers as first-class citizens. That is, a "parser" is a data item that can be passed around and operated on by functions.

Parsec provides a `Run()` function which takes a parser and some text and "runs" the parser over that text. When a parser is "run", it does the following:

- The parser consumes zero or more characters of input. (I.e., a given parser might consume nothing at all, or the entire input, or anything inbetween.)
- The parser may "succeed" or "fail".
  - If it fails, it gives a "reason" for failure.
  - If it succeeds, it "returns" a single data item (which may of course be a list object containing large amounts of data).

The caller of the `Run()` function is given a data structure indicating the success or failure of the parser, and the data returned or the failure reason (as appropriate).

So what's what a Parsec parser *is*, but what parsers does Parsec actually *provide*?

2.4 Primitive parsers

The Parsec library provides (at least) the following parsers:

- `P_Return(`x`)`: Always succeeds, consuming zero characters of input and returning x.
- `P_Fail(`x`)`: Always fails, consuming zero characters of input and returning the text string x as the failure reason.
- `P_Any()`: Consumes one character of input, and succeeds returning the read character as the result. (Fails if end-of-input.)
- `P_Char(`c`)`: If the next character of input is c then this parser consumes it and succeeds (returning c as the result). Otherwise it fails (with the reason "I was expecting c but I actually found x").

So these are the primitive parsers. It should be obvious that each parser takes only a few lines to implement, their functionallity being extremely simple. What is *not* yet obvious is how anything remotely *useful* can be achieved with just this raw material.

Indeed, it is *not* actually possible to do anything really useful until we also have some combinators to play with...

2.5 The `P_Chain()` combinator

The first combinator we will consider is the `P_Chain()` combinator. This takes two parsers and combines them into a single, bigger parser. For example,

```
parserX = P_Chain(parser1, parser2);
```

Informally, when `parserX` is run, it will run `parser1` followed by `parser2`. More precisely,

- First, `parser1` is run.
- If `parser1` fails, then `parserX` fails (with the same reason data).
- If `parser1` succeeds, it's "result" is thrown away, and `parser2` is run (starting from the input position where `parser1` left off — note that the `Run()` function can't do this). The success/failure result of `parser2` is then the overall result for `parserX`.

It should be clear that repeated application of `P_Chain()` allows an arbitrary number of parsers to be "chained together" into a single large parser. It should also be clear that

```
parserX = P_Chain(parser1, P_Chain(parser2, parser3));
parserY = P_Chain(P_Chain(parser1, parser2), parser3));
```

both implement equivilent parsers. That is, the `P_Chain()` parser is *associative*, and therefore the order of composition is unimportant. Note however that it is definitely *not commutative* — that is, exchanging the order of the parsers themselves, rather than the order of composing, *does* change the resulting parser.

Consider a small example:

```
parserX =
  P_Chain(P_Char('B'), P_Chain(P_Char('I'), P_Char('G')));
```

This parser will succeed for any string beginning with `BIG` (and return `G` as the result) and fail for any other input string. For example, the string `BAD` would cause a failure with reason

Line 1, column 2: Unpexected `A`; expecting `I`.

The "reason" is a data structure that can be programmatically queried, rather than a plain text message. (E.g., you could query the line number and hilight that line on-screen or similar.)

Parsers that parse a specific, fixed string are very common, so Parsec provides a shortcut for this common case:

```
functionP_String(str)
{
  parser = P_Char(str[0]);

  for (p=1; p<str.length; p++)
  parser = P_Chain(parser, P_Char(str[p]);

  return parser;
}
```

Now we can simply write `P_String("BIG")` to build the same parser. This is much shorter to write and easier to read.

Notice, however, that our "big parser" still returns `G` as its result. We can change this with the following dodge:

```
parserY = P_Chain(P_String("BIG"), P_Return("BIG"));
```

Recall taht `P_Chain()` will only run the second parser if the first one succeeds. So if the first parser (`P_String()`) succeeds, the second one (`P_Return()`) will be run, instantly returning the string just parsed.

We could of course have `P_Return()` return something more useful. But the above example illustrates why this primitive is useful; it enables us to control what value a given parser returns on success.

Note that the "real" implementation of `P_String()` actually returns the entire string as the parser result, rather than just the final character. (So the listing given above is actually incomplete. It is incomplete in other ways too, as we will see later.)

2.6 The `P_Fork()` combinator

Thus far, we can't do much of any interest. We can write parsers that succeed if one specific input is present, and fail otherwise. Let us now consider a second combinator, `P_Fork()`. Like `P_Chain()`, it takes two parsers and returns a parser. Let us consider how the two combinators differ.

```
paserX = P_Chain(parser1, parser2);
paserY = P_Fork(parser1, parser2);
```

As we know, `P_Chain()` will run one parser, and if that succeeds it will then run the other. However, `P_Fork()` will run one parser, and *if that fails* it will run the other parser!

Put simply, `P_Chain()` runs one parser *and then* the other, while `P_Fork()` runs one parser *or else* the other. More techincally:

- First, `parser1` is run.
- If `parser1` succeeds, `parserY` succeeds (with the same return value).
- If `parser1` fails, `parser2` is run (starting from where `parser1` *started*, not where it *finished*).
- If `parser2` succeeds, `parserY` succeeds (with the same return value). In other words, the failure reason from `parser1` is "ignored".

- If `parser2` fails, the failure reasons for both parsers are combined together and `parserY` fails with that reason data.

As an example, consider this:

```
parserYN = P_Fork(P_Char('Y'), P_Char('N'));
```

This will parser the character `Y` (and return same), *or* it will parse the character `N` (and return same). If the input is neither of these, it will fail with a message similar to

> Line 1, column 1: Unpexpected `J`, expecting `Y` or `N`.

Note how the two failures have been "merged" into a single message.

By using `P_Fork()`, you can chain together an arbitrary number of possible alternative parsers. (`P_Fork()` is associative but not commutative, just like `P_Chain()`.) The general method is along the lines of "parse this as an integer, or else parse it as a variable name, or else parse it as a function call, or else..."

It should now be clear that when a parser "fails", this is not a catastrophic condition, but merely a normal part of how a complex parser works internally. If the top-level parser fails, this indicates a user error in the input provided. But otherwise, it's just business as usual.

2.7 Optional parsing

Consider the following parser:

```
parserX = P_Fork(P_Char('X'), P_Succeed('?'));
```

This parser either consumes the letter `X` (returning `X`), or consumes nothing and returns `?`. In other words, it *optionally* parses an `X`, returning a default value if one isn't found.

This is a common need, and so again Parsec provides a shortcut:

```
function P_Optional(parser1, default_value)
{
  return P_Fork(parser1, P_Succeed(default_value));
}
```

2.8 Multiple parsing

Now suppose we want to run the same parser multiple times over, repeating it as many times as possible until it fails. Parsec provides the following shortcut for this:

```
function P_Many1(parser)
{
  return P_Chain(parser, P_Optional(P_Many1(parser)));
}
```

Here we parse one copy of the thing, "optionally" followed by "many" more copies.

Notice that this parses *one* or more copies of the given parses; Parsec provides another variant that does *zero* or more copies. Incautious use of this facility can result in infinite loops though! (Parsec also provides shortcuts for running a parser exactly n times, and for parsing multiple items that have seperators or terminators.)

2.9 The P_SuperChain() combinator

The parsers given so far are all very well, but all they do is succeed or fail depending on what the input is. None of them *return* anything of much interest. There is a specific reason for that.

Suppose, for example, that we wanted to run two parsers and return the results of each. We could try

```
parserX = P_Chain(paser1, parser2);
```

This will run `parser1` followed by `parser2`. But the `P_Chain()` combinator *throws away the result* from `parser1` before `parser2` is even run. So how are we to make a parser that returns both results?

To do that, we need a new, more complicated combinator, which I will call `P_SuperChain()`. Instead of taking two parsers, it takes a parser and a function that returns a parser, like so:

```
parserX = P_SuperChain(parser1, function1);
```

What it does is this:

- First, `parser1` is run.
- If that succeeds, `function1()` is called, and the return value from `parser1` is passed as an argument.
- `function1()` must return a new parser, which `P_SuperChain()` then runs (in the style of `P_Chain()`).

Note that if `parser1` fails, the rest of the sequence is aborted, similar to the way `P_Chain()` operates.

To see how we can achieve our stated goal, consider the following code:

```
parserX = P_SuperChain(parser1, foo);

function foo(x)
{
  function bar(y) {return P_Success([x, y]);}

  return P_SuperChain(parser2, bar);
}
```

Let us pick our way through this tangle of code. When `parserX` is run, the following occurs:

- `parser1` is run.
- `P_SuperChain()` calls `foo()` with the result from `parser1` as its argument. In other words, `x` is now whatever the result from `parser1` was.
- As you can see, the `foo()` function immediately returns a new parser, which also uses `P_SuperChain()`.
- `parser2` is run.
- The `bar()` function is called, with the result from `parser` as its arugment (i.e., `y`).
- Notice that `bar()` is declared inside `foo()`, so it can access `x` (which is a parameter to `foo()`). When run, the `bar`() function just returns a `P_Succeed()` parser which yields the value we want when run.

This might seem like an awful lot of work just to achieve such a tiny task! However, the above is significantly easier to implement in Haskell. (Functions can be nameless, and can be declared in-line in the middle of an expression.) In fact, Haskell actually provides a special form of syntax suguar especially for doing things like the above. It looks something like this:

```
parserX =
BEGIN
  parser1 --> x;
  parser2 --> y;
  P_Success([x, y]);
END;
```

This is automatically converted into the previously shown code. However, it's both easier to read and to write. Without going into the precise process, suffice it to say that

```
...
parserA;
parserB;
parserC;
...
```

is converted into calls to `P_Chain()`, while

```
...
parserA --> a;
parserB --> b;
parserC --> c;
...
```

is converted into calls to `P_SuperChain()` with extra auxhilery functions being auto-generated behind the scenes. The precise algorithm isn't important; trust me, it *works*.

Using this technology, almost all of Parsec's built-in shortcuts (e.g., `P_Many()`) actually return the thing parsed, rather than just running the parser and discarding its output.

2.10 A complete example

Now that we've seen most of the Parsec library, let's implement something with it:

```
expression = P_Fork(operation, P_Fork(number, variable));

operation =
BEGIN
  P_Char('(');
  expression --> x
  operator   --> o
  expression --> y
  P_Char(')');
  P_Success([o, x, y]);
END;

operator = P_Fork(P_Char('+'), P_Char('-'));

number   = P_Many1(P_Digit());
variable = P_Many1(P_Letter());
```

Here we have a parser which will accept all of the following strings:

```
5
x
(2+2)
(2+(2+2))
((2+x)-y)
(((x-y)+xyz)-25)
```

It will also *return* a primitive tree structure representing the thing it has parsed. And it will correctly report things like mismatched brackets. (The parsers `P_Digit()` and `P_Letter()` are pre-defined shortcuts provided by Parsec itself.)

2.11 A cautionary tale

Notice that all compound expressions *must* be bracketted. This simplifies parsing, but it's annoying for the user. Let's see if we can't remove that restriction. We might try to do that by writing

```
expression = P_Fork(operation, ...);

operation =
BEGIN
  expression --> x
  operator   --> o
  expression --> y
END;
```

This will dive into an infinite loop when run. To understand why, consider what Parsec will do:

> OK, I need to parse an expression. How do I do that? OK, well it says an expression might be an operation, so how do I parse an operation? OK, it says to start by parsing an expression and putting it in `x`. OK, so how do I parse an expression?...

You see the circularity in the logic? This is technically known as a *left-recursive grammer*, and Parsec cannot handle such a thing.

Fortunately, in this case it is easily possible to *factor* the grammer to avoid the left-recursion. The solution is of the form

```
expression =
BEGIN
  P_Many1SepBy(term1, P_Char("-")) --> term_list;
  P_Succeed(fold("-", term_list));
END;

term1 =
BEGIN
 P_Many1SepBy(term2, P_Char("+")) --> term_list;
 P_Succeed(fold("+", tern_list));
END;

term2 = P_Fork(number, variable);
```

In other words, we parse a list of "terms" seperated by operators, and then use the `fold` function to insert the appropriate operator between the terms. (This function is pre-defined in Haskell, but is not hard to define in any language.) In this way, the left-recursion is avoided.

Notice the way in which a `term1` is a sequence of `term2`. This can be extended, and used to implement operator precedences (e.g., `+` binds tighter than `-` in our example above).

Again, Parsec provides a (rather more complicated) shortcut function for multiple complicated expression parsers. It requires a structure describing the rules of the grammer and parsers for the individual parts, and assembles them for you in a way yielding the requested operator priorities and avoiding left-recursion.

2.12 Error messages

As already mentioned, the `P_Char()` function tells you what it was expecting to find when it failed, and `P_Fork()` merges such messages. But we can do better still. There is a combinator called `P_Name()` which takes a parser and a string and "names" the parser with that string. When the parser fails, it is mentioned by name. For example,

> unexpected `+`; expecting number, variable or `(`.

if you had parsers named "number" and "variable".

2.13 Backtracking

In the interests of efficiency, the `P_Fork()` combinator doesn't actually work in *exactly* the way described earlier. We said that if the first parser fails, the second parser is run. This isn't entirely true. If the first parser fails *before consuming any input*, then the second parser is run. If the first parser consumes some input and *then* fails, the second parser is not even tried, and the whole operation fails.

The reason for this is simple; if the first parser is allowed to consume some input before failing, we must "rewind" the input stream to the same place before starting the second parser. To be able to do this, at every fork we must keep hold of that data in case the current branch fails and we need to go to the other branch.

Actually, in many cases you can tell from the first character what the next item is. For example, if you're expecting either a series of digits *or* a series of letters, you just need to use whichever parser gets past the first character without failing. Indeed, if you read three letters and then a digit, it is actually *pointess* to try the other parsers because we know they will all fail.

So, in the name of efficiency, Parsec *defaults* to not backtracking. This means that as soon as a character of input has been consumed, the garbage collector can reclaim the memory that character uses (in principle, at least). If, on the other hand, you *want* to be able to backtrack, Parsec provides the `P_Try()` combinator.

`P_Try()` takes a parser and makes it so that if that parser fails, it "looks like" it failed without consuming any input. In other words, `P_Try()` is responsible for being able to rewind the input stream, and hence the inevitable overhead in memory.

In summary: By default Parsec takes the no-overhead choice of not allowing backtracking. If you need backtracking, you can manually turn it on in just the places where you need it, so overhead is minimised. You only pay for what you need.

2.14 The real Parsec

That concludes our little discussion of Parsec. I would like to make a few assorted notes:

- Writing long sequences of `P_Chain( P_Chain( P_Chain(` ... becomes tedious rapidly. In the real Haskell implementation, this combinator is *actually* called '`>>`'. This means that you can write things such as '`paraser1 >> parser2 >> parser3`'.
- Similarly, `P_Fork()` is actually called '`<|>`'.
- Parsec's actual parsers and combinators do not have '`P_`' in their names. I added that for clarity. (So `P_Return()` is actually just '`return`' in Haskell. This seems to confuse newcommers greatly, since they expect `return` to be a language keyword for returning data from a function, not a function for constructing parsers.)
- The `P_SuperChain()` function is actually called '`>>=`'. For no specific reason. (!)
- Together, `return` and `>>=` (i.e., `P_Return()` and `P_SuperChain()`) form Haskell's much-feared *monad* concept. If you understood how `P_SueprChain()` works, you now know about monads!
- In the general case, `return` turns an ordinary-thing into a monad-thing (in this case, a *parser*), and `>>=` runs a monad-thing, takes the data it produces and feeds it to a function (which is obliged to return a new monad-thing), and then runs the monad-thing just returned.
- Since Parsec parsers are a monad, Parsec is a *monadic combinator library*. A surprising number of things besides parsers can be represented as monads, so *monadic* combinator libraries are very common.
- Haskell's "special syntax" for using `P_SuperChain()` actually applies to *any* monad, not just to Parsec parsers. In fact, basic input/output operations are represented as a monad in Haskell, which explains the need for syntax sugar. (I/O is a *rather common* programming task, after all.)
- Parsec has a number of additional facilities that I haven't mentioned here. For example, Parsec isn't limited to processing lists of *characters*; you could use it to process lists of *tokens* instead, if you were using a seperate tokeniser. (Doing so tends to be slightly more efficient.) It also has features for opening and parsing files from disk from inside a parser (e.g., for `#include` functionallity) and so forth.

3 Financial contracts

Not wishing to give the impression that combinator libraries are *only* of use for constructing parsers, we will now briefly skim a description of a library for working with financial contracts.

The basic idea is that a "contract" has a financial "value" at any given point in time. This is our starting point, from which everything else follows.

The simplest contracts have a fixed value that never changes for all eternity. (Unlikely in reality, but useful for modelling.) Then we have various ways of combining contracts. For example, we can form a new

contract who's value at any given time is the sum of the values of two subcontracts at that time.

Another possibility is a contract that initially has the same value as contract X, until at a certain time when its value changes to be the same value as contract Y.

More tricky still is a contract that allows you to *choose* two possible "options". Once the choice is made, the contract's value is the same as the value of the chosen subcontract.

It should be evident that by using these functions for combining simple contracts together, new contracts of great complexity and sophistication can be constructed. Further, the contract construction functions provide a "language" for humans discussing contracts. It should also be quite evident that figuring out what value a given contract has at any point in time (given any particular set of choices) is a fairly easy task for a computer.