Advanced Programming

2014 Exam

Maya Saietz
[redacted]

November 6, 2014

Contents

1 Parsing Fast

The code for this task can be found in appendix A.1, and tests can be run with the command `runhaskell TestFast.hs` from the directory `src/fast/`.

1.1 Picking a parser library

I implemented my parser using ReadP for the following reasons:

- I have a better mental model of ReadP/SimpleParse than of Parsec.
- ReadP has fancy searchable documentation – SimpleParse doesn't.

Of course, nothing made by others is ever perfect, and I did end up implementing my own versions of `munch` and `munch1` with different types than the standard ones. The new types are:

```
munch, munch1 :: ReadP a -> ReadP [a]
```

That is, instead of taking a predicate, they now take a parser which they greedily apply.

I also made an `option' :: ReadP a -> ReadP (Maybe a)`, which is useful when you want to branch depending on whether a thing was parsed or not.

1.2 Transformations to the grammar

The grammar contains quite a lot of left recursion in the productions for *Expr*. There are also some precedence rules that need to be expressed in the grammar.

I have only included the new version of *Expr* and the new nonterminals that it uses, as the rest of the grammar is unchanged.

$$\begin{aligned}
\mathit{Expr} &\to \texttt{'return'}\ \mathit{Expr}\\
\mathit{Expr} &\to \mathit{Expr0}
\end{aligned}$$

$$\begin{aligned}
\mathit{Expr0} &\to \texttt{'set'}\ \mathit{Name}\ \texttt{'='}\ \mathit{Expr}\\
\mathit{Expr0} &\to \texttt{'set'}\ \texttt{'self'}\ \texttt{'.'}\ \ \mathit{Name}\ \texttt{'='}\ \mathit{Expr}\\
\mathit{Expr0} &\to \mathit{Expr1}
\end{aligned}$$

$$\begin{aligned}
\mathit{Expr1} &\to \mathit{Expr1}\ \texttt{'+'}\ \mathit{Expr2}\\
\mathit{Expr1} &\to \mathit{Expr1}\ \texttt{'-'}\ \mathit{Expr2}\\
\mathit{Expr1} &\to \mathit{Expr2}
\end{aligned}$$

$$\begin{aligned}
\mathit{Expr2} &\to \mathit{Expr2}\ \texttt{'*'}\ \mathit{Expr3}\\
\mathit{Expr2} &\to \mathit{Expr2}\ \texttt{'/'}\ \mathit{Expr3}\\
\mathit{Expr2} &\to \mathit{Expr3}
\end{aligned}$$

$$\begin{aligned}
\mathit{Expr3} &\to \mathit{Expr4}\ \texttt{'.'}\ \ \mathit{CallChain}\\
\mathit{Expr3} &\to \mathit{Expr4}
\end{aligned}$$

$$\begin{aligned}
\mathit{CallChain} &\to \mathit{Name}\ \texttt{'('}\ \mathit{Args}\ \texttt{')'}\\
\mathit{CallChain} &\to \mathit{Name}\ \texttt{'('}\ \mathit{Args}\ \texttt{')'}\ \texttt{'.'}\ \ \mathit{CallChain}
\end{aligned}$$

$$\begin{aligned}
\mathit{Expr4} &\to \mathit{integer}\\
\mathit{Expr4} &\to \mathit{string}\\
\mathit{Expr4} &\to \mathit{Name}\ \texttt{'('}\ \mathit{Args}\ \texttt{')'}\\
\mathit{Expr4} &\to \texttt{'self'}\\
\mathit{Expr4} &\to \mathit{Name}\\
\mathit{Expr4} &\to \texttt{'self'}\ \texttt{'.'}\ \ \mathit{Name}\\
\mathit{Expr4} &\to \texttt{'new'}\ \mathit{Name}\ \texttt{'('}\ \mathit{Args}\ \texttt{')'}\\
\mathit{Expr4} &\to \texttt{'match'}\ \mathit{Expr}\ \texttt{'\{'}\ \mathit{Cases}\ \texttt{'\}'}\\
\mathit{Expr4} &\to \texttt{'send'}\ \texttt{'('}\ \mathit{Expr}\ \texttt{','}\ \mathit{Expr}\ \texttt{')'}\\
\mathit{Expr4} &\to \ \texttt{'('}\ \mathit{Expr}\ \texttt{')'}
\end{aligned}$$

The left recursion in the productions for *Expr*1 and *Expr*2 is handled by using the `chainl1` function from ReadP.

The implementation of *CallChain* is less obvious (and practically the only interesting part of the parser), so I've included it here:

```
expr3 :: ReadP Expr
expr3 = (do
    e <- expr4
    _ <- symbol "."
    callChain e
    ) <++ expr4

callChain :: Expr -> ReadP Expr
callChain e = do
    (n, es) <- term
    let cm = CallMethod e n es
    option cm $ symbol "." >> callChain cm
```

callChain needs to handle some left-recursion, just like chainl1, and the implementation is actually a bit similar[1]. It uses the given expression as the left-hand-side of the result, and then passes that result to a recursive call to itself. If the recursive call does not succeed, it simply returns the previously computed result.

1.3 Testing the parser

The main part of the tests for the parser is done using QuickCheck. For this purpose I have implemented a generator for Fast syntax trees (FastGenerator.hs) and a pretty-printer (PrettyFast.hs). The pretty-printer tries to write as few parentheses as possible, and also tries to make some kind of indentation. "Pretty" might be an overstatement, though.

Given these two modules, it is now possible to test for the following property:

```
Right prog == parseString (ppProg prog)
```

Of course, this approach has several limitations. First of all, it only does positive testing – all generated program texts are expected to parse. Second, the pretty-printer always sets the whitespace and parentheses the same way, so it doesn't test for whitespace-handling or for what happens when there are more parentheses than is necessary. Finally, there could be bugs in the generator or pretty-printer – I actually uncovered several in the pretty-printer when I first wrote the tests.

Despite these shortcomings, such a test still says something about the robustness of a parser – the randomly generated programs contain a lot of things that are legal according to the grammar, but that sane programmers would rarely do, and this ensures that the parser can handle those corner cases.

The QuickCheck testing is supplemented by several handwritten, negative tests, and additionally all of the tests for the interpreter depend on the parser, so this adds some handwritten positive tests.

The negative tests are not very comprehensive – there are simply too many ways of making syntax errors. They cover some common typos, as well as things I judged might be easy to implement wrongly in the parser.

Tests can be run with the command `runhaskell TestFast.hs` from the directory `src/fast/`, and can also be seen in appendix B.1

1.4 Conclusion

The parser passes all the tests I've been able to come up with, so I'm going to assume that it's correct.

As far as code readability goes, I think it's pretty good. The code is split up in several sections – external API, custom parser combinators and so on – and the actual parsers are arranged in a top-down manner, with the parser of whole programs first, and the expression parsers last. Non-obvious function and parsers have descriptive comments, but most of it should be easy enough to make sense of.

2 Interpreting Fast

The code for this task can be found in appendix A.2, and tests can be run with the command `runhaskell TestFast.hs` from the directory `src/fast/`.

2.1 Data structures

My interpreter uses two data types that are both monads, `FastM` and `FastMethodM`, as suggested by the skeleton file. Both of these can be understood in terms of `Reader`, `Writer` and `State` from Haskell's standard libraries (and had this not been an exam, I probably would have implemented them using the corresponding monad transformers, but that would not demonstrate that I know how to write a monad instance).

`FastM` needs to read from the abstract syntax tree of the program, it needs to write some output, and it needs to keep track of a global state. `FastMethodM`, which is used for actually executing a method, needs to know (or read) what object it's executing in, and it needs to keep track of the method's state (that is, the variable bindings).

All bindings, whether it be class names to class declarations, names to fields or names to values, are stored using `Data.Map.Map`.

2.2 Handling `return`

One of the more challenging parts of interpreting Fast is that `return x` is an expression, rather than a statement, and thus can be found just about anywhere. If `return` is used inside some other expression (example: `2 + return 3`), evaluation should stop, and the current method should return. For example, after evaluating the exprssion `(set self.x = 2) + (return 4) + (set self.x = 3)`, the value of `self.x` will be `2`, and the method containing the expression will return `4`. None of the `+` expressions are ever fully evaluated.

This was handled by having `evalExpr` return both the result of evaluation, and a boolean indicating whether we're in the middle of returning or not. If we evaluate a sub-expression, and it reports that it's in the middle of returning, we immediately return its value.

Something similar happens when evaluating a list of expressions, such as the body of a method or a match-case. If we run into a return, or an expression that is in the middle of returning, we stop evaluation and return.

I needed a special function for evaluating a list of arguments. `evalArgs :: Exprs -> FastMethodM (Either Value Values)` returns `Right argValues` if none of the expressions caused a return, or `Left returnValue` if one of them did.

2.3 Handling limited-scope variables (method calls and match)

When calling a method, we introduce bindings that should only exist within that method. This happens automatically, because `evalMethodBody` always creates a new, empty `MethodState`.

The more complicated case is when bindings are introduced in a match case, when matching on a `TermPattern` or `AnyValue` pattern. In this case, we use the function `runWithMatchBindings`, which takes care of deleting the new bindings after we're done with the match case, and also restores any bindings that were overwritten.

2.4 Testing the interpreter

The interpreter is tested on a variety of Fast-programs. For each program, there is a corresponding `.out`-file containing the expected output. Some tests expect errors, some expect a successful evaluation with a specific result. It is worth noting that these tests depend on the parser as well, and so serve as some extra positive tests for task 1.

Tests can be run with the command `runhaskell TestFast.hs` from the directory `src/fast/`, and can also be seen in appendix B.1

3 Spreadsheet

The code for this task can be found in appendix A.3, and tests in appendix B.2.

3.1 The sheet server

The sheet-server is fairly simple. It keeps track of a list of cells, and accepts exactly one message, namely the one that tells it to create or look up a cell – that is, the message that `cell/2` sends. `cell/2` expects a reply, and is therefore blocking.

3.2 Cell servers

A cell server is much more complicated that a sheet server. It keeps track of the following things:

1. The name of the cell (`Name`)
2. The sheet server it belongs to (`Sheet`)
3. The value of the cell (`Value`)
4. The list of viewers associated with the cell (`Viewers`)
5. A `State` variable, containing the following information:
   - Whether the cell is updating or ready
   - Which dependencies it's waiting for, if it's updating
   - The current dependencies of the cell, and the last values reported by them

When a new cell is created, it has the value `undefined`, and the state {`ready`, `[]`} – it's ready, and has no dependencies.

3.2.1 Keeping track of viewers

Removing a viewer is done using `lists:delete/2`, which deletes the *first* occurence of an element from a list. Because of this, we need the invariant that a viewer can only appear once in a list of viewers. This is enforced by only adding viewers if they are not already in the list.

Both `add_viewer/2` and `remove_viewer/2` are non-blocking, as no one expects any reply, and it can't go wrong in any particularly terrible ways.

3.2.2 Formulas

Formulas are handled like so:

1. When a cell receives a formula, it tells all its viewers that it's updating, and sets its own state to be waiting for all dependencies. It also adds itself as a viewer to all of them.
2. As soon as each dependency gets a new viewer, it should send back an update message with its current value.
3. Once the cell has received update messages from all cells, it spawns a process that runs the formula function.
4. At some point, the cell will get a reply from the calculating process, and will tell its viewers that it has a new value.
5. Whenever a dependency updates, step 3 and 4 are repeated.

6. Whenever a dependency informs the cell that it's updating, the cell goes into an updating state, waiting for that dependency, and we're back at step 3 again.

3.3 Robustness and error handling

There should be no possibility of deadlocks anywhere in the code, because most of the communication between processes is asyncronous. The only blocking functions are `cell/2` and `get_viewers/1`. In `cell/2` the only communication is between the sheet server and the calling process, and in `get_viewers/1` the only communication is between the cell server and the calling process.

Other than this, I have done practically no error handling. Circular dependencies are not handled, and if a cell crashes, the rest of the sheet will probably just go on using the last reported value from that cell.

Long-running formulas *should* be working – all of the calculations stuff happens in seperate processes, so it's possible to interact with a cell while it's calculating. However, if a cell is updated with a new formula while the old one is still calculating, and if the calculation of the old value finishes *after* the calculation of the new one, the new value will be overwritten by the old one. This could easily be fixed by giving each calculation some sort of reference, and then keep track on which calculation we're waiting for.

Circular dependencies could be handled through `add_viewer/2` – whenever a cell gets a new viewer, it could tell all of its dependencies that this new viewer indirectly depends on them, and they could report onwards to *their* dependencies and so on, and if any cell is ever told that it now indirectly depends on itself, well, then we have a circular dependency.

The simplest way to at least have crashing cells *noticed* would be to link them all to the sheet server, so it crashes when they crash. Better, though, would be to monitor them and restart them when they crash.

Another kind of error handling which would be practical is detecting pattern-matching errors in formulas. For example, this formula should result in some sort of well-handled error:

```
{formula, fun ([X]) -> X end, [a, b]}
```

But in the current implementation, it just causes everything to crash.

3.4 Testing the sheet server

Tests for the sheet server consist of a file, `src/sheet/tests.erl` which contains some functions that use the sheet server. Some of these print the expected output, while some of them fail. `tests:runall/0` runs all tests, including the ones that fail or crash, while `tests:runsucceeding/0` runs all the tests that don't crash.

Testing ought to be far more thorough, but I'm out of time.

The test output can be a bit hard to read, as it appears in a somewhat random order. For a better testing experience, run the test functions one by one.

3.5 Conclusion

- The sheet server works as long as nothing goes wrong!
- Error handling is very lacking, but I have some thoughts on how to do it.
- I'm fairly sure that what I've written is not idiomatic Erlang code. I do think it's fairly readable, though.

References

[1] http://hackage.haskell.org/package/base-4.7.0.1/docs/src/Text-ParserCombinators-ReadP.html#chainl1, 2014-11-01