Bloggy Badger

Code and ideas in computer science. And maybe a short story or two.

Monday, September 07, 2015

Two kinds of backtracking

I sometimes write my own parser combinators. I sometimes make mistakes while implementing my own parser combinators. In this post, I describe a mistake I made a few times: using the wrong kind of backtracking effect.

2015-08-11 update: it turns out everything in this post is already well-known in the litterature, see the Reddit discussion for links.

Two ways to order effects

So, like I said, I sometimes write my own parser combinators. It's not that hard: a parser can either succeed and consume some characters from a String, or it can fail, causing the computation to backtrack. Those two effects are already implemented separately by the State and Maybe monads, so we can create our custom Parser monad as a combination of the two, using monad transformers.

The one aspect of monad transformers about which I always need to think twice is the order in which I need to place the layers in order to get the side-effects to interact in the way I expect. With State and Maybe, depending on the order in which the layers are placed, we can either get a permanent state which survives backtracking:

```
import Control.Applicative
import Control.Monad
import Control.Monad.Trans.Class
import Control.Monad.Trans.Maybe
import Control.Monad.Trans.State

type PermanentState a = MaybeT (State String) a
runPermanent :: PermanentState a -> String -> (Maybe a, String)
runPermanent = runState . runMaybeT

-- |
-- >>> runPermanent writeThenBacktrack "initial state"
-- (Just "secret state!","secret state")
writeThenBacktrack :: PermanentState String
writeThenBacktrack = writeSecret <|> appendBang
  where
    writeSecret :: PermanentState a
```


Popular Posts

Every single game from GitHub Game Off 2012

Rant against .gitkeep (and solution)

I understand comonads

Will it memoize?

Comonads are neighbourhoods, not objects

Subscribe via RSS

Posts

Comments

Blog Archive

▼ 2015 (14)
- ► December (9)
- ▼ September (1)
  - Two kinds of backtracking
- ► June (2)
- ► May (1)
- ► January (1)

► 2014 (11)
► 2013 (5)
► 2012 (13)
► 2011 (3)
► 2010 (2)
► 2009 (6)
► 2008 (16)
► 2007 (2)
► 2006 (3)

gelisam's shared items

a mathematician's lament
On Balanced Trees and Car Insurance

```
    writeSecret = do
        lift $ put "secret state"
        fail "backtracking"

    appendBang :: PermanentState String
    appendBang = do
        s <- lift get
        return $ s ++ "!"
```

Or we can get what we want for parsers, a path-specific state which gets reset along with the computation when we backtrack:

```
type Parser a = StateT String Maybe a

runParser :: Parser a -> String -> Maybe (a, String)
runParser = runStateT

-- |
-- >>> runParser writeThenBacktrack' "initial state"
-- Just ("initial state!","initial state")
writeThenBacktrack' :: Parser String
writeThenBacktrack' = writeSecret <|> appendBang
  where
    writeSecret :: Parser a
    writeSecret = do
        put "secret state"
        fail "backtracking"

    appendBang :: Parser String
    appendBang = do
        s <- get
        return $ s ++ "!"
```

This way, if we start consuming characters and we discover that we need to backtrack, we start the next alternative from the state we were at the beginning as if the characters were never consumed.

```
char :: Char -> Parser ()
char expected = do
  (c:cs) <- get
  guard (c == expected)
  put cs

-- |
-- >>> runParser twoAlternatives "aab"
-- Just (1,"")
--
-- >>> runParser twoAlternatives "ab"
-- Just (2,"")
twoAlternatives :: Parser Int
twoAlternatives = (pure 1 <* char 'a' <* char 'a' <* char 'b')
              <|> (pure 2 <* char 'a' <* char 'b')
```

Two kinds of backtracking

The kind of backtracking we have seen so far consists of abandoning the current alternative and trying the next one. This behavior can be summarized by the following equation:

Lessons from Laundry
Life in a Lazy Universe
A Story About Odd and Even Numbers
A defining moment (trying to define "science", "research", and "math")
Games Without Frontiers: How Videogames Blind Us With Science
The Princess Rescuing Application
What Colour are your bits?
Earn or Learn? The math.
Elliot Aronson's war on AIDS
Secrets of motivation
Don't know what you don't know

Links

Lost Garden
sigfpe blog
Lambda the Ultimate
namebinding blog
Agda
try Haskell! (in your browser)
try ruby! (in your browser)
Chicken Scheme
Computer Science at McGill
Lost Garden

```
(f1 <|> f2) <*> x = if succeeds f1
                   then f1 <*> x
                   else f2 <*> x
```

I'll call that kind "if-then-else backtracking". There is another, more general kind of backtracking which Maybe does not support, which I'll call "distributive backtracking" after the following equation:

```
(f1 <|> f2) <*> x = (f1 <*> x)
                <|> (f2 <*> x)
```

The difference between the two behaviors is that if `x` fails in `f1 <*> x`, if-then-else backtracking will give up on the entire `(f1 <|> f2) <*> x` expression, whereas distributive backtracking will also try `f2 <*> x` before giving up. If the fact that `x` failed in `f1 <*> x` is sufficient to determine that it will also fail in `f2 <*> x`, then failing early is a good performance improvement. Otherwise, it makes the parser fail more often than it should.

Examples

For a concrete case in which this makes a difference, consider this alternate implementation of `twoAlternatives`:

```
-- |
-- >>> runParser twoAlternatives' "aab"
-- Just (1,"")
--
-- >>> runParser twoAlternatives' "ab"
-- Nothing
twoAlternatives' :: Parser Int
twoAlternatives' = optionalPrefix <* char 'a' <* char 'b'
  where
    optionalPrefix :: Parser Int
    optionalPrefix = (pure 1 <* char 'a')
                 <|> (pure 2)
```

Instead of repeating the `char 'a' <* char 'b'` suffix twice, I have refactored the code so that only the optional `'a'` prefix is included in the disjunction. By doing so, I have accidentally changed the meaning of the parser: instead of accepting `"aab"` and `"ab"`, it now only accepts `"aab"`. That's because by the time the parser encounters the mismatched `'b'`, it has already made its decision to use the extra `'a'` in the prefix, so it's too late to go back and use an empty prefix instead.

In this simple contrieved example, it would be straightforward to reorganize the definition of `twoAlternatives'` to both avoid the repetition and retain the original meaning. In the more complicated real-world circumstances in which I encountered this limitation, however, refactoring was not a solution.

I was trying to debug a parser in the same way I debug some programs: by commenting out portions of it in order to isolate the problem to a smaller amount of code. Unfortunately, the different portions of the parser were tightly coupled: the commented-out portion was supposed to parse a few characters, the next portion was supposed to parse the next few characters, and commenting

out the first parser caused the second parser to fail, as it was presented with the characters which were intended for the first parser. So after commenting out the first parser, I replaced it with a dummy parser which accepts any string of characters whatsoever.

Unfortunately, this dummy parser consumed the entire input, and then the second parser was again presented with the wrong part of the input, namely the end-of-file marker. I expected the dummy parser to backtrack and to try consuming slightly fewer characters, but it couldn't, because the parsing framework I was using was based on if-then-else backtracking. After the dummy parser consumed everything, no backtracking was allowed, just like what happened after we parsed the extra `'a'` prefix above.

In order to make my dummy parser stop at the correct spot, I'd have to know what character the second parser was expecting to start at, so I could tell my dummy parser to accept everything but that character. Except that the first parser might have been supposed to parse some copies of that character as well, in which case that strategy would cause us to stop too early. So I'd have to reimplement much of the logic of the parser I was commenting out, which defeated the purpose of commenting it out. I was stuck.

Implementing distributive backtracking using the list monad

The easiest way to switch from if-then-else backtracking to distributive backtracking is to implement our backtracking using a list instead of a Maybe.

```
type Parser a = StateT String [] a


runParser :: Parser a -> String -> [(a, String)]
runParser = runStateT
```

With this simple change (I did not have to change any other definition nor type signature), `twoAlternatives'` now succeeds at parsing its input.

```
-- |
-- >>> runParser twoAlternatives' "aab"
-- [(1,"")]
--
-- >>> runParser twoAlternatives' "ab"
-- [(2,"")]
twoAlternatives' :: Parser Int
twoAlternatives' = optionalPrefix <* char 'a' <* char 'b'
  where
    optionalPrefix :: Parser Int
    optionalPrefix = (pure 1 <* char 'a')
                 <|> (pure 2)
```

The reason it works is because `optionalPrefix` is no longer returning the first result which works, instead it returns a lazy list containing both results. This way, once the mismatched `'b'` is encountered, there is enough information to go back and try another element from the list.

Implementing distributive backtracking in terms of if-then-else backtracking

If, like me, you're stuck with a parser based on if-then-else backtracking and you can't change the implementation, do not despair! As the title of this section indicates, it is possible to implement distributive backtracking on top of it.

Notice that the difference between the two kinds of backtracking is only apparent when a disjunction is followed by sequential composition, i.e., the case `(f1 <|> f2) <*> x`. If the parser was already written in the canonical form `(f1 <*> x) <|> (f2 <*> x)`, where `x` can contain more disjunctions but `f1` and `f2` cannot, the kind of backtracking involved would make no difference. So, can we somehow put expressions into canonical form before executing our parser?

To turn `(f1 <|> f2) <*> x` into `(f1 <*> x) <|> (f2 <*> x)`, we must apply the continuation `(<*> x)` to both sides of the `(<|>)`. So I guessed that my wrapper's representation ought to receive a continuation, and the rest of the implementation followed naturally from following the types:

```
newtype DistributiveParser a = DistributiveParser
  { unDistributiveParser :: forall r. (a -> Parser r) -> Parser r
  } deriving Functor

runDistributiveParser :: DistributiveParser a
                      -> String -> Maybe (a, String)
runDistributiveParser p = runParser (unDistributiveParser p return)

instance Alternative DistributiveParser where
    empty = DistributiveParser $ \cc -> fail "backtracking"
    f1 <|> f2 = DistributiveParser $ \cc -> unDistributiveParser f1 cc
                                        <|> unDistributiveParser f2 cc

instance Applicative DistributiveParser where
    pure x = DistributiveParser $ \cc -> cc x
    df <*> dx = DistributiveParser $ \cc ->
                unDistributiveParser df $ \f ->
                unDistributiveParser dx $ \x ->
                cc (f x)

distributiveParser :: Parser a -> DistributiveParser a
distributiveParser px = DistributiveParser $ \cc -> do
    x <- px
    cc x

char' :: Char -> DistributiveParser ()
char' = distributiveParser . char
```

By reimplementing `twoAlternatives'` using our new wrapper, we obtain a version which succeeds at parsing its input, even though the underlying backtracking implementation is still if-then-else backtracking.

```
-- |
-- >>> runDistributiveParser twoAlternatives'' "aab"
-- Just (1,"")
--
-- >>> runDistributiveParser twoAlternatives'' "ab"
```

```
-- Just (2,"")
twoAlternatives'' :: DistributiveParser Int
twoAlternatives'' = optionalPrefix <* char' 'a' <* char' 'b'
  where
    optionalPrefix :: DistributiveParser Int
    optionalPrefix = (pure 1 <* char' 'a')
                <|> (pure 2)
```

Conclusion

I guess the conclusion is: make sure to use the appropriate implementation of backtracking if you can, and also when you can't :)

Posted by gelisam on Monday, September 07, 2015

No comments:

Post a Comment

Links to this post

Create a Link

Newer Post Home Older Post

Subscribe to: Post Comments (Atom)