Lecture 27

Lectured and scribed by Sunil Kothari

December 12, 2008

1 Overview

Our goal today is to have parsers which transform types in one language to another language. Specifically, we want "$(a \rightarrow (b \rightarrow c))$" to be converted into "$(Arrow\ (TyVar\ a)(Arrow\ (TyVar\ b)(TyVar\ c)))$".

We plan to do the following:

1. We will start with a simple type language which requires parens for unique parse trees.
2. We will look at an efficient version of the above parser.
3. Next, we look at how to parse without parens.
4. Finally, we make the parens optional in our language.

2 Review

Recall that parser type is defined as:

$$newType\ Parser\ a = MkP(String \rightarrow [(a, String)])$$

In the previous lecture, we looked at the different ways of defining a type. Look at the notes or in the book, if you want more information.
For all the parsers that we define in this lecture, most of them can be made from only two operators. return is defined as:

```
return :: a -> Parser a
return v = MkP( i -> [(v,i)])

Parser1> apply (return 1) "abc"
[(1,"abc")] :: [(Integer,String)]
```

The other operator is (>>=), called "then" in Hutton's book.

```
p >>= f MkP (\i -> case (apply p i) of
                      [] -> []
                      [(v,out)] -> apply (f v) out
```

In general, parsers are built in the following fashion

$p1 >>= \lambda v1 \rightarrow$
$p2 >>= \lambda v2 \rightarrow$
$\vdots$
$pn >>= \lambda vn \rightarrow$
$return\ (f\ v1\ v2\ \ldots vn)$

This notation is normally not used. Instead Haskell provides the do notation

do $v1 \leftarrow p1$
   $v2 \leftarrow p2$
   $\vdots$
   $v3 \leftarrow pn$
return $f\ v1\ v2\ \ldots\ vn$

Let's briefly review some of the parsers that Prof. Caldwell introduced last time.

item, which parses a string character by character

```
item:: Parser Char
item = MkP(ı -> case i of
                    [] -> []
                    (x:xs) -> [(x,xs)])
```

Next, we define a parser which reads the first three characters of a string and returns a pair of the first and the third character, and also returns the remaining string.

```
p :: Parser (Char,Char)

p =  do x <- item
        item
        y <- item
        return (x,y)

Parser> apply p "xyzzy"
[(('x','z'),"zy")] :: [((Char,Char),String)]
```

The two parsers can also be combined using the +++ (choice) operator.

```
 p `plus` q = Mkp f
     where f x = apply p s ++ apply q s
```

Hutton's choice operator

```
p +++ q  = MkP (\i -> case apply p i of
                           [] -> apply q i
                           m -> m )
```

A standard thing is to tokenize *i.e.* eat as many space as possible.

```
space :: Parser ()
space = do many (char ' ')
           return ()

token :: Parser a -> Parser a
token p = do space
             v <- p
             space
             return v
```

3 Parsing types

This is what was covered in last lecture. Today, we will build on that and convert types in one language to another language. In compilers, we often do that. We write programs in one language (normally called the source language) and convert it into another language (often called the target language). For example, transforming C++ programs to Java programs.
Each of these languages has a grammar, which dictates what terms (programs) are syntactically valid in a given language. In our case, the source grammar is given as:

$type ::= string$
$\quad | \text{"("}\ type\ \text{"}->\text{"}\ type\ \text{")"}$
$\quad | \text{"("}\ type\ \text{"}\times\text{"}\ type\ \text{")"}$

We need parenthesis in this language, since some strings can be parsed into more than one parse trees. For example, the string "$a \to b \to c$" can be parsed as $(a \to b) \to c$ or as $a \to (b \to c)$.
The target language is given as:

$type ::= TyVar\ string$
$\quad | \ Arrow\ type\ type$
$\quad | \ Prod\ type\ type$

Let's write a parser now:

```
typ::Parser Type
typ = tyvar +++ arrow  +++ prod
```

The *typ* parser is a "choice" in the sense that we can either have a type variable or an arrow type or a product type. The *tyvar*, *arrow*, and *prod* parsers are defined as:

```
tyvar:: Parser Type
tyvar = do n <- identifier
           return (TyVar n)

arrow::Parser Type
arrow = do symbol "("
           a <- typ
           symbol "-"
           symbol ">"
           b <- typ
           symbol ")"
           return (Arrow a b)

prod:: Parser Type
prod = do symbol "("
          a <- typ
          symbol "x"
          b <- typ
          symbol ")"
          return (Prod a b)
```

We can load this and check.

```
Parser1> apply typ "(a -> (b -> c))"
[(Arrow (TyVar "a") (Arrow (TyVar "b") (TyVar "c")),[])] :: [(Type,String)]
Parser1> apply typ "(a -> b -> c))"
[] :: [(Type,String)]
Parser1> apply typ "(a -> b -> c)"
[] :: [(Type,String)]
Parser1> apply typ "(a -> b) -> c"
[(Arrow (TyVar "a") (TyVar "b"),"-> c")] :: [(Type,String)]
```

Note that we the parser takes care of extra spaces as shown below:

```
Parser1> apply typ "((a ->      b     ) -> c)"
[(Arrow (Arrow (TyVar "a") (TyVar "b")) (TyVar "c"),[])] :: [(Type,String)]
Parser1>
```

The parser so created is not an efficient parser. The inefficiency comes from the following code:

```
typ::Parser Type
typ = tyvar +++ arrow  +++ prod
```

To have an efficient parser, we change the code slightly. The new parser is:

```
typ::Parser Type
typ = tyvar +++ comptype

comptype ::Parser Type
comptype = do symbol "("
              a <- typ
              constructor <- op
              c <- typ
              symbol ")"
              return (constructor a c)

op:: Parser (Type -> Type -> Type)
op = arrow +++ prod



arrow ::Parser (Type -> Type -> Type)
arrow = do symbol "->"
           return Arrow


prod::Parser (Type -> Type -> Type)
prod = do symbol "x"
          return Prod
```

This doesn't change the final output. For example,

```
Parser2> apply typ "((a ->      b     ) -> c)"
[(Arrow (Arrow (TyVar "a") (TyVar "b")) (TyVar "c"),[])] :: [(Type,String)]
Parser2>
```

Now we switch to a type language, where the compound types always associate to the right. So we can do away with the parens in our source language. The new grammar is given as:

type ::= *string*
| *type* " → " *type*
| *type* " × " *type*

The parser undergoes a slight modification.

```
typ::Parser Type
  typ = comptype

  comptype ::Parser Type
  comptype = do a <- tyvar
                do constructor <- op
                   c <- comptype
                   return (constructor a c)
                 +++ return a

  op:: Parser (Type -> Type -> Type)
  op = arrow +++ prod
```

We can load this in the interpreter and the results are as expected.

```
Parser3> apply typ "a -> b -> c"
[(Arrow (TyVar "a") (Arrow (TyVar "b") (TyVar "c")),[])] :: [(Type,String)]
Parser3> apply typ "a -> b -> c-> d -> e"
[(Arrow (TyVar "a") (Arrow (TyVar "b") (Arrow (TyVar "c") (Arrow (TyVar "d") (TyVar "e")
Parser3> apply typ "a -> b x c"
[(Arrow (TyVar "a") (Prod (TyVar "b") (TyVar "c")),[])] :: [(Type,String)]
Parser3>
```

Finally, we make the parenthesis optional in the sense that if they are not specified then the compound types will associate to the right. Again, the modified code is given below:

```
typ::Parser Type
    typ = comptype +++ parencomptype

    parencomptype :: Parser Type
    parencomptype = do symbol "("
                       a <- typ
                       symbol ")"
                       return a


    comptype ::Parser Type
    comptype = do a <- tyvar +++ parencomptype
                  do constructor <- op
                     c <- typ
                     return (constructor a c)
                   +++ return a
```

Again, the results are as we expect.

```
Parser4> apply typ "a -> (b -> c) -> d -> e"
```

```
[(Arrow (TyVar "a") (Arrow (Arrow (TyVar "b") (TyVar "c")) (Arrow (TyVar "d") (TyVar "e"]
Parser4> apply typ "a -> (b -> c -> d) -> e"
[(Arrow (TyVar "a") (Arrow (Arrow (TyVar "b") (Arrow (TyVar "c") (TyVar "d"))) (TyVar "e
Parser4>
```