MP4 - CPS Transform

Logistics

- revision: 1.1
- due: July 13, 2016

Changelog

1.1

- Changed due date from July 6 to July 13.
- Added test-set.
- Fix `var` parser to allow for numbers in variable names.
- Changed `cpsDecl` to not modify the name of the function so that we don't have to worry about whether the applied function in an `AppExp` is a bound by the arguments in the declaration or comes from the global scope.

Objectives

One of the main objectives of this course is to make you comfortable translating code representations from one form to another. In MP2, you wrote an interpreter to evaluate expressions. In this MP you will translate a very small subset of functional-style code from direct style into continuation passing style (CPS). Functional-language compilers can use CPS to express control-flow so that functional language can be compiled to an imperative target.

This will ensure that you understand CPS, and also help you be able to take a mathematical description of a program transformation and implement it in code. You will also practice using a method of propagating state by using a technique known as plumbing.

Goals

- Be able to manually translate a chunk of simple functional code into CPS.
- Write some functions which can automatically translate simple functional code into CPS.

Getting Started

Relevant Files

In the directory `app` you'll find `Main.hs` with all the relevant code. In this file you will find all of the data definitions, a simple parser, the REPL, and stubbed-out functions for you to finish.

Running Code

As usual, you have to `stack init` (you only need to do this once).

To run your code, start GHCi with `stack ghci` (make sure to load the `Main` module if `stack ghci` doesn't automatically). From here, you can test individual functions, or you can run the REPL by calling `main`. Note that the initial `$` and `>` are prompts.

```
$ stack ghci
 ... More Output ...
Prelude> :l Main
Ok, modules loaded: Main.
*Main> main
```

To run the REPL directly, build the executable with `stack build` and run it with `stack exec main`.

Testing Your Code

As in MP2, you will be able to run the test-suite with `stack test`:

```
$ stack test
```

It will tell you which test-suites you pass, fail, and have exceptions on. To see an individual test-suite (so you can run the tests yourself by hand to see where the failure happens), look in the file `test/Spec.hs`.

It will also tell you whether you have passed enough of the tests to receive credit for the ML. If you do not pass 60% of the tests on the MP, you will not receive credit for the associated ML.

You can run individual test-sets by running `stack ghci` and loading the `Spec` module with `:l Spec`. Then you can run the tests (specified in `test/Tests.hs`) just by using the name of the test:

Look in the file `test/Tests.hs` to see which tests were run.

Given Code

The Types

We have two types for this program. The first is called `Stmt`, and it corresponds roughly to a Haskell function declaration. It has one constructor `Decl :: String -> [String] -> Exp -> Stmt`. The first argument is the name of the function, the second is a list of parameter names, and the last is the function body (which is an `Exp`).

We have also provided a `Show` instance for the `Stmt` type, which means that the function `show :: Read a => a -> String` is defined for our type. This enables pretty-printing so that we can get useful output.

```haskell
data Stmt = Decl String [String] Exp
            deriving (Eq)

instance Show Stmt where
    show (Decl f params exp) = f ++ " " ++ intercalate " " params ++ " = " ++ (show exp)
```

The second type we have is called `Exp`, which represents expressions. There are six constructors that are encapsulated within `Exp`. We have `IfExp`, with the usual three arguments; `AppExp` for function application (note: functions are only applied to one of their arguments at a time); `IntExp` and `VarExp` for integers and variables; an `OpExp` that takes an operator (as a string) with two operands; and finally a `LamExp` to represent anonymous functions.

We've also provided a `Show` instance for the `Exp` type so that it can be pretty-printed as well.

```haskell
data Exp = IfExp Exp Exp Exp
         | OpExp String Exp Exp
         | AppExp Exp Exp
         | IntExp Integer
         | VarExp String
         | LamExp String Exp
```

```
        deriving (Eq)

instance Show Exp where
    show (VarExp s)       = s
    show (IntExp i)       = show i
    show (AppExp f e)     = show f ++ " " ++ show e
    show (OpExp op e1 e2) = "(" ++ show e1 ++ " " ++ op ++ " " ++ show e2 ++ ")"
    show (IfExp e1 e2 e3) = "(if " ++ show e1 ++ " then " ++ show e2 ++ " else " ++ show e3 ++ ")"
    show (LamExp x e)     = "(\\" ++ x ++ " -> " ++ (show e) ++ ")"
```

The Parser

We've provided you with a parser again this time. It will live in `app/Main.hs` with the rest of the code.

First, we have a type synonym to make reading the types of parsers easier.

```
-- Pretty parser type
type Parser = ParsecT String () Identity
```

Lexicals

Lexical parsers are just small parsers which look for very simple structure on the input stream. You'll notice that the return types of these parsers are all simple types (like `Parser String`, or `Parser Integer`). This means that they don't have any meaning for our programming language yet, we'll be using them as building blocks for more complex parsers.

```
symbol :: String -> Parser String
symbol s = do string s
              spaces
              return s

int :: Parser Integer
int = do digits <- many1 digit <?> "an integer"
         spaces
         return (read digits :: Integer)

var :: Parser String
var = let keywords = ["if", "then", "else"]
      in  try $ do v1 <- letter                  <?> "an identifier"
                   vs <- many (letter <|> digit) <?> "an identifier"
                   spaces
                   let v = v1:vs
                   if (any (== v) keywords)
                    then fail "keyword"
                    else return v

oper :: Parser String
oper = do op <- many1 (oneOf "+-*/<>=") <?> "an operator"
          spaces
          return op

parens :: Parser a -> Parser a
parens p = do symbol "("
              pp <- p
```

```
              symbol ")"
              return pp
```

Expressions

Now we can write some parsers that are actually relevant to our programming language. You're not expected to fully understand these parsers yet, or to be able to reproduce them. For your next MP though, you will have to write a simple parser, so it might be worth it to read through and see if it makes sense.

The majority of the complexity in our grammar is with expressions, which is why the parsers for them are more complicated than for lexicals or statements. This is apparent when we look at how many data-constructors we have for `Exp`, we need to make sure our parser can handle each of them.

```
intExp :: Parser Exp
intExp = do i <- int
            return $ IntExp i

varExp :: Parser Exp
varExp = do v <- var
            return $ VarExp v

opExp :: String -> Parser (Exp -> Exp -> Exp)
opExp str = do symbol str
               return (OpExp str)

mulOp :: Parser (Exp -> Exp -> Exp)
mulOp = opExp "*" <|> opExp "/"

addOp :: Parser (Exp -> Exp -> Exp)
addOp = opExp "+" <|> opExp "-"

compOp :: Parser (Exp -> Exp -> Exp)
compOp =     opExp "<"  <|> opExp ">"
         <|> opExp "<=" <|> opExp ">="
         <|> opExp "/=" <|> opExp "=="

ifExp :: Parser Exp
ifExp = do try $ symbol "if"
           e1 <- expr
           symbol "then"
           e2 <- expr
           symbol "else"
           e3 <- expr
           return $ IfExp e1 e2 e3

lamExp :: Parser Exp
lamExp = do try $ symbol "\\"
            param <- var
            symbol "->"
            body <- expr
            return $ LamExp param body

atom :: Parser Exp
atom =     intExp
```

```
        <|> ifExp
        <|> lamExp
        <|> varExp
        <|> parens expr

expr :: Parser Exp
expr = let arith  = term `chainl1` addOp
           term   = factor `chainl1` mulOp
           factor = app
           app    = do f <- many1 atom
                       return $ foldl1 AppExp f
       in  arith `chainl1` compOp
```

We've also provided the function `parseExp :: String -> Either ParseError Exp` which can be used to see what the parser returns for a specific input string. This is useful for understanding how the data-structures we use to store programs written in our language (the abstract syntax tree) are related to the grammar/syntax of our language.

```
parseExp :: String -> Either ParseError Exp
parseExp str = parse expr "stdin" str
```

```
*Main Lib> parseExp "x + 1"
Right (OpExp "+" (VarExp "x") (IntExp 1))
*Main Lib> parseExp "asdfa*"
Left "stdin" (line 1, column 7):
unexpected end of input
expecting white space, an integer, "if", "\\", an identifier or "("
```

Declarations

Once we can parse expressions, declarations are comparatively simple.

```
decl :: Parser Stmt
decl = do f <- var
          params <- many1 var
          symbol "="
          body <- expr
          return $ Decl f params body
```

Once again, we have provided the function `parseDecl :: String -> Either ParseError Stmt` which can be used to investigate the connection between the grammar of our language and abstract syntax tree of a particular program.

```
parseDecl :: String -> Either ParseError Stmt
parseDecl str = parse decl "stdin" str
```

```
*Main Lib> parseDecl "f x = x + 1"
Right (Decl "f" ["x"] (OpExp "+" (VarExp "x") (IntExp 1)))
```

The REPL

The REPL can be used to enter CPS declarations and see the resulting translations. It will prompt you for string, pass it through the `parseDecl` parser, check for success, and if so pass it through `cpsDecl`. Then the result will be pretty-printed using the `show :: Show a => a -> String` function.

```
prompt :: String -> IO ()
prompt str = hPutStr stdout str >> hFlush stdout
```

```haskell
printLn :: String -> IO ()
printLn str = hPutStrLn stdout str >> hFlush stdout

repl :: IO ()
repl = do input <- prompt "> " >> getLine
          case input of
            "quit" -> return ()
            _      -> do case parseDecl input of
                            Left err    -> do printLn "Parse error!"
                                              printLn $ show err
                            Right decl  -> printLn . show $ cpsDecl decl
                         repl


main :: IO ()
main = do putStrLn "Welcome to the CPS Transformer!"
          repl
          putStrLn "GoodBye!"
```

Problems

Manual Translation

First, you'll manually translate a few Haskell functions into CPS. This can help you gain an intuition about how the automatic translator will work.

`factk :: Integer -> (Integer -> t) -> t`

Write the factorial function in continuation passing style. The first argument is the number that it should compute the factorial of. The second argument is the continuation to apply to the result.

Note that you *will not receive credit* if you calculate the factorial separately using a normal factorial function then put it through the continuation. You should instead recursively cal `factk` with an updated continuation which contains the computation to make at this step in the recursion as well as the original continuation.

```
*Main Lib> factk 10 id
3628800
```

`evenoddk :: [Integer] -> (Integer -> t) -> (Integer -> t) -> t`

Write the function `evenoddk` in continuation passing style.

The `evenoddk` function takes a list and two continuations. The first continuation, if run, will receive the sum of all the even elements of the list. The second continuation, if run, will receive the sum of all the odd elements of the list.

No additions should be performed until a continuation is called. This means that you need to build up the addition to make in each continuation, *not do the addition as you process the list.*

The function decides what to do when it gets to the last element of the list. If it is even, it calls the even continuation, otherwise it calls the odd continuation.

You can assume the input list will have at least one entry in it.

```
*Main Lib> evenoddk [2,4,6,1] id id
1
*Main Lib> evenoddk [2,4,6,1,4] id id
16
*Main Lib> evenoddk [2,4,6,1,9] id id
10
```

Automated Translation

Here, you'll define two functions do all the translation. We will call `cpsDecl` on a declaration (`Stmt`). This function will return a new declaration with an added argument `k`, and a body which has been CPS transformed to use that as its continuation.

We'll also have a function `cpsExp` which is used to translate an `Exp` into CPS. It takes an extra argument, which is an integer, and we return an integer in addition to the transformed expression. These integers are how we are going to get fresh variables each time we want to generate a new continuation. They act as a sort of accumulator, receiving the number for the next fresh variable and returning a potentially updated number after the transformation is performed.

In particular, we have given you a utility function called `gensym` (for "generate symbol") that generates fresh variables each time it is called using that particular integer. Whenever we need a fresh variable, we call `gensym` with our old integer.

```
gensym :: Integer -> (String, Integer)
gensym i = ("v" ++ show i, i + 1)
```

```
*Main Lib> gensym 20
("v20",21)
```

Notice that the return type for both `gensym` and `cpsExp` is a tuple of an expected return type and an integer; this is how we remember what numbers have already been used (and what comes next). This is a common pattern in languages that do not have mutation (or even languages that do but we chose not to use it), and is often called "plumbing".

The CPS transform $[\![D]\!]$ represents a program transform taking declaration D from direct style to continuation passing style. The CPS transform $[\![E]\!]_k$ represents a program transform taking expression E from direct style to continuation passing style with k as its continuation. Remember that the continuation is a function to apply to the result of the current expression. In this example, we will call the continuation k on the value that expression E evaluates to.

We will also distinguish between *simple* expressions (no available function calls) and normal expressions. (We will go into more detail later about how we will distinguish simple from normal expressions.) Operators will be left in direct style, and we will not be converting λ-expressions in this MP (though your code will certainly emit them!).[1]

To indicate that an expression is simple, we will put an overbar on it, like this: $\overline{e}$. Some expressions, like variables and integers, are obviously simple so we will omit the overbar in such a case. As an example, $\overline{e_1} + e_2$ indicates that e_1 is simple, but that e_2 is not. Thus, e_2 would need to be transformed.

Define `isSimple`

The `isSimple` function takes an expression and determines if it is simple or not. A simple expression, in our context, is one that does not have an available function call. A function call is available if it's possible for

[1]Most automatic CPS transforms do not make this distinction: there are, in fact, many different CPS transforms, and researchers have spent a lot of time coming up with transforms that have different properties. This means that if you find a paper describing a CPS transform, odds are excellent it will not look exactly like this one. You should still find it recognizable though!

the current expression to execute it. In the language subset you have been given, a function call is always available unless it is within the body of an unapplied λ-expression. We aren't transforming λ-expressions in this MP, so you can ignore implementing `isSimple` over `LamExp`s.

```
*Main Lib> isSimple (AppExp (VarExp "f") (IntExp 10))
False
*Main Lib> isSimple (OpExp "+" (IntExp 10) (VarExp "v"))
True
*Main Lib> isSimple (OpExp "+" (IntExp 10) (AppExp (VarExp "f") (VarExp "v")))
False
```

```
cpsExp :: Exp -> Exp -> Integer -> (Exp, Integer)
```

You'll need to define `cpsExp` over all of the data-constructors that make up the `Exp` type. The first argument to `cpsExp` is the expression you are transforming, and the second is the current continuation that has been built up.

Remember that we have provided you with a parser and the function `parseExp`, which you can use to type in expressions for testing. Instead of having to write the entire Haskell AST of an expression, you can use `parseExp`.

Define `cpsExp` for Integer and Variable Expressions

$$
\begin{aligned}
[\![i]\!]_k &= k\ i \\
[\![v]\!]_k &= k\ v
\end{aligned}
$$

In Haskell, this would (almost) be implemented as:

```haskell
almostCpsExp :: Exp -> Exp -> Exp
almostCpsExp (IntExp i) k = AppExp k (IntExp i)
almostCpsExp (VarExp v) k = AppExp k (VarExp v)
```

But, remember that we have to add in the extra "plumbing" which keeps track of how many fresh variables we've generated, which accounts for the extra `Integer` in both the arguments and return of `cpsExp`.

Define `cpsExp` for If Expressions

$$
\begin{aligned}
[\![\texttt{if } \overline{e_1} \texttt{ then } e_2 \texttt{ else } e_3]\!]_k &= \texttt{if } e_1 \texttt{ then } [\![e_2]\!]_k \texttt{ else } [\![e_3]\!]_k \\
[\![\texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3]\!]_k &= [\![e_1]\!]_{(\lambda v.\texttt{if } v \texttt{ then } [\![e_2]\!]_k \texttt{ else } [\![e_3]\!]_k)}, \text{where } v \text{ is fresh.}
\end{aligned}
$$

For an if expression with a not-simple guard, we need to transform the guard and give it a continuation that selects the proper (transformed) branch for us. If the guard is simple, we can leave it alone and transform the branches using the original continuation.

Notice how the order of evaluation becomes explicit when we make this transform; the condition of the if expression is evaluated first, and only then is one of the branches picked for evaluation. This is part of the power of CPS, you can choose exactly how expressions in your language are simplified down.

Define `cpsExp` for Operator Expressions

These are the most complex, since there are four possible cases.

For $\otimes$ a binary operator:

$$
\begin{aligned}
[\![\overline{e_1} \otimes \overline{e_2}]\!]_k &= k\ (e_1 \otimes e_2) \\
[\![e_1 \otimes \overline{e_2}]\!]_k &= [\![e_1]\!]_{(\lambda v.k(v \otimes e_2))}, \text{where } v \text{ is fresh.} \\
[\![\overline{e_1} \otimes e_2]\!]_k &= [\![e_2]\!]_{(\lambda v.k(e_1 \otimes v))}, \text{where } v \text{ is fresh.} \\
[\![e_1 \otimes e_2]\!]_k &= [\![e_1]\!]_{(\lambda v_1.[\![e_2]\!]_{(\lambda v2.k(v_1 \otimes v_2))})}, \text{where } v_1 \text{ and } v_2 \text{ are fresh.}
\end{aligned}
$$

Notice that we are careful to preserve the order of the arguments to the operator - the order we convert expressions *to* is very important (that is, e_1 and e_2 cannot be swapped on the right-hand side of any of the above), because this preserves the order of evaluation. In real life it doesn't matter the order we convert them *in* (that is, whether we choose to convert e_1 or e_2 first), as long as their results are passed in the correct order, but to keep from annoying the graders[2] please evaluate things from left to right.

Define `cpsExp` for Application Expressions

$$
\begin{aligned}
[\![f\ \overline{e}]\!]_k &= f\ e\ k \\
[\![f\ e]\!]_k &= [\![e]\!]_{(\lambda v.f\ v\ k)}, \text{where } v \text{ is fresh.}
\end{aligned}
$$

Note there are two rules: one for when the argument is simple, and one for when the argument needs a conversion. We are going to assume that the argument is run first, the result of which is passed into the function.

Notice the "where v is fresh" part. Variables don't grow moldy, but if you give every continuation function a parameter named v, it won't work. We could have nested continuations which lead to unwanted α-capture. Therefore, we need to generate new names whenever we make a new λ-expression.

This should be done using the `gensym :: Integer -> (String, Integer)` function explained above. Remember that `cpsExp` needs to return the most up-to-date fresh variable number so that further calls to `cpsExp` don't use the same variables again.

Define `cpsDecl`

Declarations are much simpler than expressions because we don't need to maintain the state of a fresh-variable counter when translating declarations. This is because each declaration has its own local variable scope anyway.

$$
[\![f\ x_1\ \cdots\ x_n\ =\ e]\!] = (f\ x_1\ \cdots\ x_n\ k\ =\ [\![e]\!]_k)
$$

This corresponds to the `cpsDecl` function. It adds an extra parameter `k` to the parameter list, and then translates the body of the function. (The parentheses on the right-hand side are added for clarity - the equality on the left tranforms to the whole equality on the right.)

Monads (optional)

This part is highly optional.

Perhaps you are thinking "couldn't we use a monad to get rid of this plumbing?" If so, you are correct. There is a monad called the `State` monad that can do this for us. (In fact, there is a `GenSym` monad as well!)

You are not required to use monads for this MP. But if you would like to use them, you may, as long as you **do not change the type of `cpsDecl` or `cpsExp`**. Define your own function `cpsExpM`, the monadic version of `cpsExp`, and you can define `cpsExp` in terms of `cpsExpM` instead. The testing interface to **`cpsExp` must remain unchanged**.

[2] Annoying a grader is bad luck.