MP 3 / ML 2: Continuation-Passing Style

CS 421
Revision 1.0

Assigned Friday, March 4, 2016
Due Monday, March 14, 2016 - 11:59pm

1 Objectives and Background

One of the main objectives of this course is to have you become 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 `Haskell` code from direct style into continuation passing style (CPS).

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.

2 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`, the first argument is the name of the function, the second is a list of parameter names, and finally, an expression body.

```
data Stmt = Decl String [String] Exp
    deriving Show

class ToStr a where
  toStr :: a -> String

instance ToStr Stmt where
  toStr (Decl f params exp) = f ++ " " ++ intercalate " " params ++ " = " ++ (toStr exp)
```

You will see that we have created a type-class `ToStr` in an effort to overload the function `toStr`. We let `Haskell` create the standard `show`, so that printing this gives you the type constructors; this could help you in debugging and in understanding the structure of your data. The `toStr` function, on the other hand, prints out something that looks a lot more like standard `Haskell` .

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. (Note in the below that `parseExp` is a function that we have provided you that takes and parses an input string and returns an `Either` in a way we will describe later.)

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

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

```
*Main Lib> parseExp "x + 1"
Right (OpExp "+" (VarExp "x") (IntExp 1))
*Main Lib> let (Right exp) = parseExp "x + 1" in toStr exp
"(x + 1)"
```

3 The Math

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.

Declarations

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

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.)

Integers and Variables

$$[\![i]\!]_k = k\ i$$
$$[\![v]\!]_k = k\ v$$

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

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

We will talk about the almost part in a minute.

Applications

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

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, but that would cause unwanted α-capture. Therefore, we need to generate new names whenever we make a λ-expression. We will explain how that is done in the next section.

If Expressions

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

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.

[1]Most automatic CPS transforms do not make this distiction: 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!

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

For $\otimes$ a binary operator:

$$\begin{aligned}
\llbracket \overline{e_1} \otimes \overline{e_2} \rrbracket_k &= k\ (e_1 \otimes e_2) \\
\llbracket e_1 \otimes \overline{e_2} \rrbracket_k &= \llbracket e_1 \rrbracket_{(\lambda v.k(v \otimes e_2))}, \text{ where } v \text{ is fresh.} \\
\llbracket \overline{e_1} \otimes e_2 \rrbracket_k &= \llbracket e_2 \rrbracket_{(\lambda v.k(e_1 \otimes v))}, \text{ where } v \text{ is fresh.} \\
\llbracket e_1 \otimes e_2 \rrbracket_k &= \llbracket e_1 \rrbracket_{(\lambda v_1.\llbracket e_2 \rrbracket_{(\lambda v2.k(v_1 \otimes v_2))})}, \text{ where } v_1 \text{ and } v_2 \text{ are fresh.}
\end{aligned}$$

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. This rule is written for plus, but it applies to all the other operators in the language.

4 Your Assignment

4.1 Using Stack

There is a very nice build system for `Haskell` called `stack`, and many of you recommended we use it for the course. You can get the stack tool from

http://docs.haskellstack.org/en/stable/README/

There are installers for the three common operating systems. You can install Stack via Homebrew on a Mac by running `brew install haskell-stack` in your Terminal.

There are two commands you will likely use:

- `stack repl` — This will start a REPL from GHCI. Use this for your interactive development.
- `stack test` — This will run the test cases.

4.2 Given Files

The files live in a directory `mp3-cps`. This is where you will want to run your `stack` commands. The file you want to edit and that you will be turning in is `app/Main.hs`. The parser and types are defined in `src/Lib.hs`. There are a few basic test cases in `test/Spec.hs`. We will be using our own for grading, and we encourage you to develop your own tests.

4.3 The Functions

The first two are just simple functions that we want you to write in continuation passing style. The next three are functions for doing the translation.

Problem 1. factk: Write the factorial function in continuation passing style.

```
*Main Lib> :t factk
factk :: Integer -> (Integer -> a) -> a
*Main Lib> factk 10 id
3628800
```

Problem 2. evenoddk: Write the function `evenoddk` in continuation passing style.

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

No additions should be performed until a continuation is called.

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> :t evenoddk
evenoddk :: [Int] -> (Int -> t) -> (Int -> t) -> t
*Main Lib> evenoddk [2,4,6,1] id id
1
```

[2]Annoying a grader is bad luck.

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

Problem 3. 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 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 `LamExps`.

```
*Main Lib> :t isSimple
isSimple :: Exp -> Bool
*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
```

Problem 4. cpsDecl and cpsExp

These two functions do all the translation, as described above in Section 3. We will call `cpsDecl` on a declaration. 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.

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

In Section 3, we said we *almost* had a partial definition for `cpsExp`. Compared with that definition, here we have an extra argument to `cpsExp`, 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. These integers 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." You did something similar to this in the interpreter MP, and you will need to use that technique with your `cpsExp` function.

Typing in these expressions using the data constructors can get tedious, so se have also given you two utility functions: `parseExp` and `parseDecl`. They return an `Either`, where `Right` will contain the `Decl` or the `Exp` in the case of a successful parse, and `Left` will contain an error message otherwise.

```
*Main Lib> :t parseExp
parseExp :: [Char] -> Either Text.Parsec.Error.ParseError Exp
*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 "("

*Main Lib> :t parseDecl
parseDecl :: [Char] -> Either Text.Parsec.Error.ParseError Stmt
*Main Lib> parseDecl "f x = x + 1"
Right (Decl "f" ["x"] (OpExp "+" (VarExp "x") (IntExp 1)))
```

5 Sample Run

There is a `main` function that simply calls `repl`, inputs a declaration and transforms it for you. If you use Stack to compile a standalone executable this is what it will do. We think most people will use `stack repl` and run `main` by hand.

Here is a sample run:

```
% stack repl
Configuring GHCi with the following packages: mp3-cps
GHCi, version 7.10.2: http://www.haskell.org/ghc/  :? for help
[1 of 2] Compiling Lib
[2 of 2] Compiling Main
Ok, modules loaded: Lib, Main.
*Main Lib> main
 CPS> mapplus f a b = f a + f b
Pretty Result:
mapplusk f a b k = f a (\v1 -> f b (\v2 -> k (v1 + v2)))
Details:
Decl "mapplusk" ["f","a","b","k"] (AppExp (AppExp (VarExp "f") (VarExp "a"))
    (LamExp "v1" (AppExp (AppExp (VarExp "f") (VarExp "b"))
        (LamExp "v2" (AppExp (VarExp "k") (OpExp "+" (VarExp "v1") (VarExp "v2")))))))
```

The REPL uses `toStr` to show you a pretty result, and then `show` to expand out the actual representation.

6 Monads

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`**. Our tests will call `cpsDecl`, and not call `cpsExp` directly, so it is safe to change its interface.

You may want to investigate `Control.Monad.State`. After the MP is due, we will release an alternate version of the solutions that show how to monadify the code.