COMP105 Assignment: Lambda Calculus

Due Monday, May 2 at 11:59PM.

This assignment will give you practice with lambda calculus, including how substitution, alpha-conversion, and reduction work. These concepts are fundamental to the lambda calculus and form the basis of the semantics for many programming languages.

Setup

Make a clone of the the following git-repository:

```
git clone linux.cs.tufts.edu:/comp/105/git/lambda
```

which should give you a directory `lambda` with files `linterp.sml`, `Lhelp.ui`, `Lhelp.uo`, `Makefile`, and `basis.lam`.

Introduction to the Lambda Interpreter

You will be implementing a small, interactive interpreter for the lambda calculus. This section explains how to use the interpreter and the syntax it expects. A reference implementation of the interpreter is available in `/comp/105/bin/linterp`.

Concrete syntax

Every definition must be terminated with a semicolon. Comments are C++ style line comments, starting with the string `//` and ending at the next newline. A definition can be a term, a binding, or a `use` statement.

A `use` statement loads a file into the interpreter as if it had been typed in directly. A `use` statement is of the form

```
-> use filename;
```

When a term is entered at the toplevel, any free variables in the term which appear in the environment are substituted with their bindings in the environment, then the term is reduced to normal form (if possible) and the result is printed.

```
-> term;
```

A binding first evaluates the term on the right hand side as above, and then binds the name on the left hand side to the resulting term in the environment. It is an error for the term on the right hand side to contain any free variables which are not bound in the environment. Bindings are of the form:

```
-> name = term;
```

or

```
-> noreduce name = term;
```

If the `noreduce` keyword appears, the term on the right-hand side is not normalized. This tactic can be useful for terms that have no normal form, such as

```
noreduce bot = (\x.x x)(\x.x x);
noreduce Y   = \f.(\x.f(x x))(\x.f(x x));
```

A lambda term can be either a variable, a lambda abstraction, an application, or a parenthesized lambda term. Precedence is as in ML. A lambda abstraction is written as follows. Note that each lambda abstraction abstracts over exactly one variable.

```
\name.term
```

Application of one term to another is written:

```
term1 term2
```

The lambda interpreter is very liberal about names of variables. A name is any string of characters that contains neither whitespace, nor control characters, nor any of the following characters: `\ ( ) . = /`. Also, the string `use` is reserved and is therefore not a name. So for example, the following are all legal:

```
1    = \f.\x.f x;
True = \x.\y.x;
one  = True 1;
```

A short example transcript

A healthy lambda interpreter should be capable of something like the following:

<transcript>=
```
-> true  = \x.\y.x;
-> false = \x.\y.y;
-> pair  = \x.\y.\f.f x y;
-> fst = \p.p (\x.\y.x);
-> snd = \p.p (\x.\y.y);
-> true;
\x.\y.x
-> fst (pair true false);
\x.\y.x
-> snd (pair true false);
\x.\y.y
-> if = \x.\y.\z.x y z;
if
-> (if true fst snd) (pair false true);
\x.\y.y
-> (if false fst snd) (pair true false);
\x.\y.y
```

For more example definitions, see the `basis.lam` file distributed with the assignment.

Pair Problems

For these problems, you may work on your own or with a partner. Please provide implementations for Exercises 1-3, described below. The purpose of these problems is to help you learn about substitution and reduction, the fundamental operations of the lambda calculus. The problems will also give you a little more practice in continuation passing, which is an essential technique in lambda-land.

Problem Details

For these problems, define appropriate types and functions in `linterp.sml`. When you are done, you will have a working lambda interpreter. Some of the code we give you (`Lhelp.ui` and `Lhelp.uo`) is object code only, so you will have to build the interpreter using Moscow ML. Typing `make` should do it.

1. *Evaluation—Basics*. Using ML, create a type definition for a type `term`, which should represent a term in the untyped lambda calculus. Using your representation, define the following functions with the given types:

```
lam : string -> term -> term    (* lambda abstraction *)
app : term -> term -> term      (* application        *)
var : string -> term            (* variable           *)
cpsLambda :                     (* observer           *)
  forall 'a .
```

```
term ->
(string -> term -> 'a) ->
(term -> term -> 'a) ->
(string -> 'a) ->
'a
```

These functions must obey the following algebraic laws:

```
cpsLambda (lam x e)  f g h = f x e
cpsLambda (app e e') f g h = g e e'
cpsLambda (var x)    f g h = h x
```

string that converts a term to a string in uScheme syntax. Your `toString` function should be *independent* of your representation. That is, it should use the functions above instead of directly accessing your `term` representation. -->

The official solution to this problem is under 15 lines of ML code.

2. *Evaluation—Substitution*. Implement substitution on your `term` representation. To implement substitution, you should define the following two functions:

- `freeVars : term -> string list` which returns a list of all the free variables in a term.
- `subst : string * term -> term -> term`. To compute the substitution *M[x*|–>*N]*, you should call `subst (x, N) M`.

You should use the function `cpsLambda` to define these functions. Additional helper functions are likely useful. The official solution to this problem is just under 40 lines of ML code.

3. *Evaluation—Reductions*. In this problem, use your substitution function to implement two different evaluation strategies:

a. Implement normal-order reduction on terms. That is, write a function `reduceN : term -> term` that takes a term, performs a single reduction step (either beta or eta) in normal order, and returns a new term. If the term you are given is already in normal form, your code should raise the exception `NormalForm`, which you should define.
b. Implement applicative-order reduction on terms. That is, write a function `reduceA : term -> term` that takes a term, performs a single reduction step (either beta or eta) in applicative order, and returns a new term. If the term you are given is already in normal form, your code should raise the exception `NormalForm`, which you should reuse from the previous part.

The official solution to this problem is under 20 lines of ML code.

What to submit: Pair Problems

You should submit two files:

- a `README` file containing
  - the names of the people with whom you collaborated
  - the numbers of the problems that you solved
  - the number of hours you worked on the assignment.
- `linterp.sml`, containing your modifications to a lambda calculus interpreter.

How to submit: Pair Problems

When you are ready, run `submit105-lambda-pair` to submit your work. Note you can run this script multiple times; we will grade the last submission.

Hints and guidelines

The lambda intepreter

Here are some common mistakes to avoid in implementing the lambda calculus interpreter:

- For substitution, remember you need to implement *capture-avoiding* substitution. In certain circumstances, you will need to come up with fresh names. It is standard practice to use an imperative reference variable to generate fresh names.
- Test your substitution code before implementing your reduction rules. Bugs in your substitution code will make your normalization functions have behavior that is very difficult to understand.
- For debugging purposes, you might find it useful to define versions of your reduction functions that print a status report after a given number of reductions.
- In your reduction functions, just because a sub-expression is in normal form doesn't mean that the entire expression is in normal form. If it may not be and if a recursive call raises the `NormalForm` exception, you need to catch the exception and respond appropriately.
- Don't think that an application is in normal form just because one subterm is in normal form—you have to check **both** subterms.
- Don't forget **the eta rule**:

  ```
  \x.M x --> M   provided x is not free in M
  ```

  Here is a reduction in two eta steps:

  ```
  \x.\y.cons x y --> \x.cons x --> cons
  ```

  Your interpreters *must* eta-reduce when possible.
- Don't forget to reduce under lambdas.
- **Don't clone and modify** your code for reduction strategies; too many people who do this wind up with wrong answers. The code should not be that long; just type in every case.
- Don't try to be clever about a divergent term; just reduce it. (It's a common mistake to try to detect the possibility of an infinite loop. Mr. Turing proved that you can't detect an infinite loop, so please don't try.)
- If you test an interpreter before implementing reduction, you may see some alarming-looking terms that have extra lambdas and applications. This is because the interpreter uses lambda to implement the substitution that is needed for the free variables in your terms. Here's a sample:

  <transcript of interpreter without reductions>=
  ```
  -> thing = \x.\y.y x;
  thing
  -> thing;
  (\thing.thing) \x.\y.y x
  ```

  Everything is correct here except that the code claims something is in normal form when it isn't. If you reduce the term by hand, you should see that it has the normal form you would expect.

How your work will be evaluated

Your ML code will be judged by the usual criteria, emphasizing

- Correct implementation of the lambda calculus
- Good form
- Names and contracts for helper functions
- Structure that exploits standard basis functions, especially higher-order functions, and that avoids redundant case analysis

Back to the class home page