CS 3110 Spring 2016
Problem Set 5
Version 2 (last modified April 19, 2016)

PS5: OCalf

Deadline: April 28, 2016

This assignment should be done with one partner. Sharing of code is permitted only between you and your partner; sharing among larger groups is prohibited.

A baby camel is called a "calf." In this assignment you will implement an interpreter for OCalf, a functional language containing many of the core features of OCaml.

Overview

Here is a Backus-Naur form (BNF) grammar for the syntax of OCalf expressions, to give you a sense of what features are in the language:

```
e ::=  (* expressions *)
    | () | i | b | s | x
    | e1 bop e2
    | if e1 then e2 else e3
    | let x = e1 in e2
    | let rec x = e1 in e2
    | e1 e2
    | fun x -> e
    | (e1,e2)
    | C e
    | match e with p1 -> e1 | ... | pn -> en
bop ::=  (* binary operators *)
    | + | - | * | > | < | = | >= | <= | <> | ^
p ::=  (* patterns *)
    | () | i | b | s | x | C p | (p1,p2)
i ::= integers
b ::= booleans
s ::= strings
x ::= variable identifiers
C ::= constructor identifiers
```

Some of the ways that OCalf differs from OCaml are the following:

- All OCalf variants must be non-constant.
- OCalf has pairs, but not k-tuples for $k \geq 3$.
- OCalf does not have built-in (syntactic support for) lists.
- OCalf does not have a module system.

- OCalf has no imperative features.

However, in all other respects, remember OCalf is a subset of OCaml, so when in doubt, do what OCaml does.

Some of the OCalf interpreter is already implemented for you. Your task is to implement evaluation and type inference, as well as to build a test suite that you will submit. You will need to understand the big-step environment model of OCaml to complete the interpreter.

Objectives

- Understand the design of an interpreter.
- Implement a big-step environment model semantics, including closures.
- Better understand OCaml itself by implementing some of its main features.
- Know the important building blocks of type inference, and implement one of them.
- Formulate a test suite that can detect errors in implementations other than your own.
- Work in a collaborative environment, and use Git version control.
- Acquire experience working on a large software engineering project.

Git

You are required to use Git (or another version control system). Throughout your development of PS5, commit your changes to a shared repository. Use those checkins to provide checkpoints, in case you need to restore your development to a previous point. Sync with a remote repository to communicate code between you and your partner.

You will submit a log of your version control activity, which will be a minor portion of your grade for this assignment. We expect proper use of version control; just commiting all your code at the end will not receive credit for this portion of the assignment.

Private repos are of the utmost importance. A public repo would share your code with the entire world, including your classmates, thus violating the course policy on academic integrity. Therefore we require that you keep all your CS 3110 related code in private repos.

You can get private repos from the following services:

- Github Student (5 private repos)
- Bitbucket (unlimited private repos)
- Gitlab (unlimited private repos)

Recommended reading

- The CS 3110 Style Guide
- The Try Git Tutorial
- Tower's Git Cheat Sheet
- The scheldule for office hours. Start early and ask questions.

Tasks

- Implement the evaluation of OCalf expressions, as described below under Part 1.
- Submit a test suite for evaluation, as described below under Part 2.
- Implement the type inference and checking of OCalf expressions, as described below under Part 3.
- Write code that has good style and is well documented
- Use Git (or another version control system) as part of your development process, and submit your commit log.

We must be able to compile and run your interpreter in the CS3110 virtual machine. You may not change the names and types appearing in `.mli` files. Solutions that do not obey these stipulations will receive minimal credit.

What we provide

In the release code you will find these files:

- Many `.mli` and `.ml` files, which are described below under Part 0.
- A template file `written.txt` for submitting your written feedback on the assignment.

What to turn in

Submit files with these names on CMS:

- `ps5src.zip`, containing your solution code, your test suite, and your written feedback.
- `vclog.txt` containing your version control log.

To prepare `ps5src.zip` for submission

From the directory that contains `main.ml`, bundle all your source code and your test suite into a zip file with this command:

```
$ zip ps5src.zip *.ml{,i,y,l} *.txt
```

Do not include any compiled bytecode files, otherwise your submission might become too big to upload. Double check that you got all your files with this command:

```
$ zipinfo -1 ps5src.zip
```

To prepare `vclog.txt` for submission

Run the following command in the directory containing `main.ml`:

```
$ git log --stat > vclog.txt
```

Permitted OCaml features

Imperative data structures are now permitted. You can use `refs`, `arrays`, `for` and `while` loops if you want. Side effects are not necessarily bad, but keep in mind that over-use of imperative features is bad style. We urge you to think carefully before choosing to use imperative data structures, and leverage the functional features as much as you can.

Part 0: Understand the OCalf codebase

Your preliminary task is to familiarize yourself with the structure of the code we have shipped to you. We provide the following modules, comprising both `.ml` and `.mli` files, in the release code:

- `Eval` and `Infer` have skeleton code for the functions that implement evaluation and type inference. There are some unimplemented helper functions in these files that the course staff found effective in their own solution. You are free to implement them, change them, or remove them entirely.
- `Ast` and `TypedAst` contain the type definitions used in evaluation and type inference, respectively. The `TypedAst` module also contains `annotate` and `strip` functions for converting between the two.
- `Examples` and `Lambda` contain example OCalf expressions. `Examples` contains many small examples from this writeup. `Lambda` contains a large example.
- `Printer` contains functions for printing out values of various types.
- `Parse` contains functions to construct ASTs from strings. It relies on a lexer and parser implemented in `lexer.mll` and `parser.mly`.

Exercise: Skim each of the `.mli` and `.ml` files in the release code, and plan to come back and read them in more detail later as necessary.

Compiling and running

From the directory containing `main.ml`, run

```
$ cs3110 compile main.ml
```

to compile the interpreter. As the parser is compiled, it will produce a message `1 shift/reduce conflict`. This is expected behavior.

To run the intepreter, we will use OCaml's own REPL. After compiling, load `utop`. You can use the OCalf interpreter by directly calling functions that implement it. For example,

```
"if true then 3110 else 0"
|> parse_expr
|> eval []
|> string_of_value
```

causes the string `"if true then 3110 else 0"` representing an OCalf expression to be parsed into an OCalf AST, evaluated to an OCalf value, then converted to an OCaml string suitable for printing. (Note that evaluation won't yet succeed with the release code, because `eval` is unimplemented.)

Note that the reason functions like `eval` are available in utop is that we provide an `.ocamlinit` file in the release code[1]. When utop starts, it automatically `#use`s this file, which automatically

[1]Files whose names start with "`.`" are **hidden**; use `ls -A` to show a directory listing that includes hidden files.

`#load`s and `open`s many of the modules from the release code for your convenience in interactive testing. Whenever you find yourself typing the same thing into the REPL more than once, consider adding it to your `.ocamlinit`!

Part 1: Evaluation

Implement the function

```
eval : Eval.environment -> Ast.expr -> Eval.value
```

in `eval.ml`. This function evaluates OCalf expressions in the big-step environment model semantics. For example,

```
eval [] (parse_expr "if false then 3 + 5 else 3 * 5"));;
- : value = VInt 15
```

(Note that we expose the representation of environments as association lists. Arguably this is poor design: all the client of an environment needs to know is that it is a dictionary, and the particular dictionary representation is irrelevant.)

Whenever evaluation reaches a place where the semantics gets **stuck**, meaning that evaluation could not meaningfully proceed but the expression being evaluated is not yet a value, `eval` should produce the special value `VError`. For example, `3 + true` and `match 3 with | 1 -> false` should both evaluate to `VError`.

Although you are free to tackle the implementation of `eval` in any way you see fit, you are strongly encouraged to follow the plan outlined below. Our test cases will proceed in the plan's order, so following the plan will maximize your chances of partial credit.

Step 0: Remind yourself of the big-step semantics of OCaml, because you are essentially implementing that judgment.

Step 1: Implement eval without LetRec or Match.

Implement `eval` for unit, integers, Booleans, strings, `BinOp`, `If`, `Var`, `Fun`, `Pair`, `Variant`, `App`, and `Let`.

Note: `BinOp` operators should work identically to their OCaml counterparts, with one exception. For simplicity's sake, comparison and equality is **only defined on primitive types (String, Int and Bool).**

Step 2: Implement LetRec.

Implement `eval` for `LetRec`. This is tricky, because a binding is needed in the environment for the defined name before its definition can be evaluated. We'll solve this problem with a technique called **backpatching**.

To evaluate `let rec f = e1 in e2` using backpatching, first evaluate `e1` to a value `v1` in an environment where `f` is bound to a **dummy value**, which may be any value at all. (If the value of `f` is used while evaluating `e1`, it is an error. This would occur, for example, if

the programmer wrote `let rec x = 3 + x in ...`.) Then imperatively update the binding of `f` to `v1`. This "ties the knot," allowing `v1` to refer to itself. Finally evaluate `e2` in the environment where `f` is bound to `v1`.

To support backpatching, the `environment` type contains `binding ref`'s instead of `binding`'s, thus enabling bindings to be mutated.

Step 3: Implement Match.

Implement `eval` for `Match`. You do not need to check whether pattern matching is exhaustive or whether there are unused match cases. Return `VError` if pattern matching fails at run time.

Part 2: Test suite

As part of developing `eval`, you naturally will be constructing test cases that demonstrate the (in)correctness of your implementation. Let's take that one step further. The course staff will develop many buggy implementations of `eval`. Your task is to construct a suite of test cases that finds all our bugs.

Use `test_eval.ml` from the release code as a template. Add your test suite to the file. We will copy that file (and only that file) into each of our buggy interpreters, then run

```
$ cs3110 compile test_eval.ml
$ cs3110 test test_eval.ml
```

We will examine the output to see whether your test suite correctly detects the bugs in our interpreters.

Part 3: Type inference

OCalf's type inference algorithm proceeds as follows:

1. Each node of the AST is **annotated** (aka **decorated**) with a unique type variable.
2. The AST is traversed to **collect** a set of equations between types that must hold for the expression to be well-typed.
3. Finally, those equations are solved to find an assignment of types to the original variables; this step is called **unification**.

If the unification phase discovers that the system is unsatisfiable, then it is impossible to give a type to the expression, so a type error would be reported. If the system is underconstrained, then there will be some "leftover" variables after unification. The typechecker would give them user-friendly names like `'a` and `'b`.

Example. Consider `(fun x -> 3 + x)`. The annotation phase would add a new type variable to each subexpression:

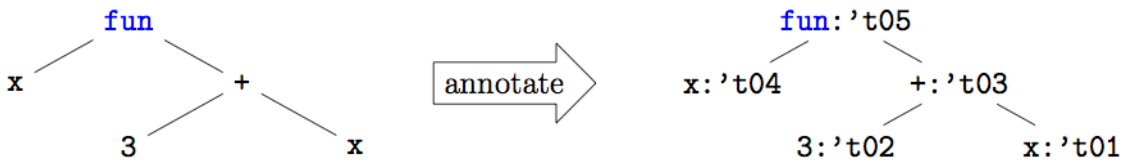

Figure 1: parsing

Next, the collection phase would collect equations from each node:

- We know that `(fun x -> e) : t1 -> t2` if `e:t2` under the assumption `x:t1`. Stated differently, `(fun (x:t1) -> e:t2) : t3` if and only if `t3 = t1 -> t2`. Therefore, the equation `'t05 = 't04 -> 't03` would be collected from `fun` node.
- Similarly, the equations `'t03 = int`, `'t02 = int`, and `'t01 = int` would be collected from the `+` node.
- The equation `'t02 = int` would be collected from the `3` node.
- We know that if `x` had type `t` when it was bound then it has type `t` when it is used. Therefore, `'t01 = 't04` would be collected from the `x: 't01` node.

The collection phase can also raise exceptions if it encounters unresolvable errors during its execution (i.e. an unbound constructor name).

Finally, the system of equations is solved in the unification phase, assigning `int` to `'t01` through `'t04`, and `int -> int` to `'t05`.

We provide the annotation and unification phases of type inference for you; your task is to implement the function

```
collect : variant_spec list -> annotated_expr -> equation list
```

in `infer.ml`. We strongly encourage you to follow this plan:

Step 1: Implement collect without Match or Variant.

Implement `collect` for all syntactic forms except `Variant` and `Match`.

Step 2: Partially implement collect for Match.

Implement `collect` for `Match`, but omit handling of `PVariant` patterns. Make sure the bindings from the patterns are available while checking the bodies of the cases.

Step 3: Implement collect for Variant.

Extend `collect` to handle variants. Deriving the correct constraints for variants is tricky. Consider this code:

```
type 'a list = Cons of 'a * 'a list | Nil of unit
let x = Cons(1,Nil ()) in
let y = Cons(true,Nil ()) in
42
```

An overly-strict implementation of collection might report a type error, if collection generates constraints that force the separate occurrences of `Nil` and `Cons` in binding `x` and `y` to have the same types. A better implementation would generate constraints with distinct type variables, thus permitting the code above.

Have **fun**!