Homework 3

Due October 14

Handout 3
CS242: Autumn 2015
October 7

Problems

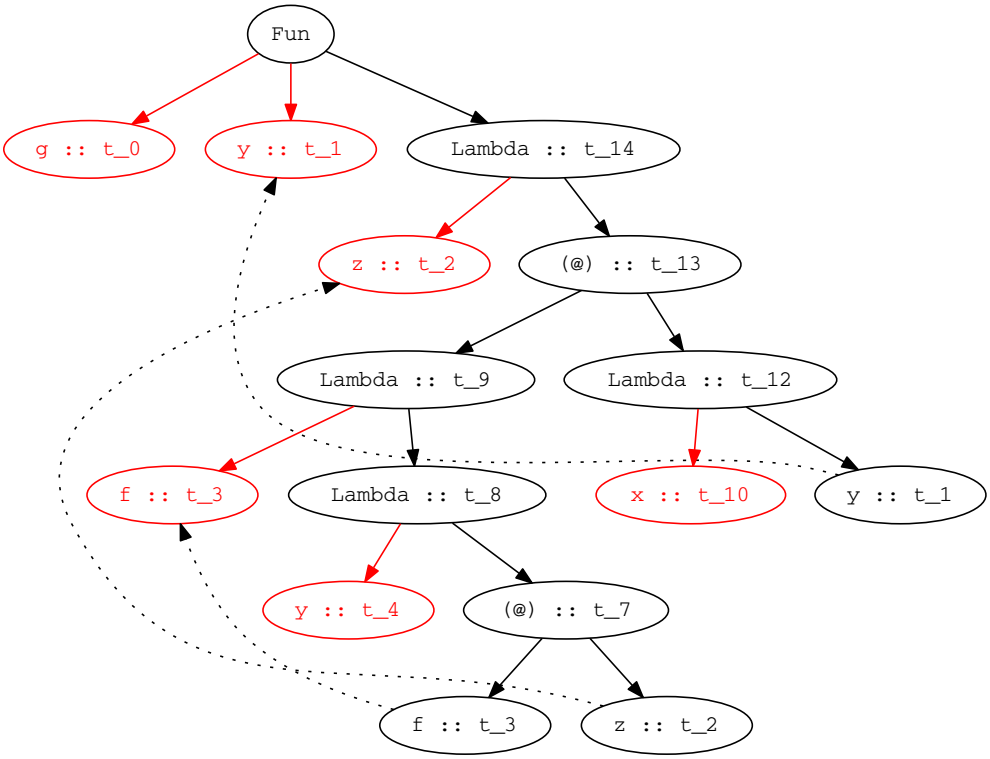
1. Type classes

There is a mini-Haskell lab on type classes. Download it from the course website.

2. Type inference by hand

(a) For this problem, please type up your solutions and submit them along with your lab in a text file named `inference.txt` Here is a Haskell program and its corresponding parse tree:

```
g y = \z -> (\f -> (\y -> f z)) (\x -> y)
```



Write down the seven constraints that Hindley-Milner type inference would generate, using the type variables from the parse tree.

(b) What is the type of `g` when it is used in the context of `g "z" "y1"` (the type of `"z"` is `String`).

3. Type inference by computer

In this lab, you will be implementing Hindley-Milner type inference for μHaskell. The primary components of this lab will be the implementation of *constraint generation* and *constraint solving*.

Running the code The command line interface for your type checker is `cli.js`; it takes a single argument which is the μHaskell file to typecheck. When your implementation is finished, it should give output like this:

```
corn33:~> nodejs cli.js should_compile/id.hs
_t0 = _t1 -> _t1
type: _t1 -> _t1
```

Input syntax The type checker supports a *very* restricted subset of Haskell. Here are the salient points:

- Expressions and pattern matches support exactly the syntax in the μHaskell grammar.
- Only one function may be defined per file.
- Declarations *must* take a single argument to pattern match on: `f x y = ...` is not supported (although you can write `f x = \y -> ...`).
- Whitespace sensitive layout is not supported: thus, if you write a declaration, it must be on a single line.
- The only supported operators are == and +.

We've written a few sample programs for you in the `should_compile` and `should_fail` directory which you can use as models if you would like to create more tests. You can run them with `cli.js`, e.g., `node cli.js should_compile/id.hs`.

The μHaskell data structures We've defined data structures for the various abstract syntax trees you will be manipulating using the `adt` library. There are structures for patterns, declarations and types. You can view these definitions in `lib/expr.js`, `lib/pat.js`, `lib/decl.js` and `lib/type.js`. The fields of these objects follow this convention: `e.isBool` tests if an expression is a boolean literal, `e.val` extracts the actual boolean, and `Expr.Bool(true)` constructs a literal representing true. For nullary constructors like `Expr.Nil`, it's not necessary to call the function (so use `Expr.Nil`, not `Expr.Nil()`).

Here is a summary of all the types and their field names:

```
Expr.Var(name)              // variables, e.g. x
    .Int(val)               // integer literal, e.g. 0
    .Bool(val)              // boolean literal True/False
    .If(cond, then, else_)  // if-then-else (note underscore)
    .Nil                    // empty list literal []
    .Lambda(name, body)     // lambda abstraction, e.g. \x -> e
    .App(fun, arg)          // function application, e.g. f x
    .Pair(fst, snd)         // pair constructor, e.g. (x, y)

Pat.Var(name)
   .Pair(fst, snd)          // (fst,snd) pattern match
   .Cons(head, tail)        // (head:tail) pattern match
   .Nil                     // [] pattern match

Decl.Decl(name, pat, body)  // name pat = body
```

```
Type.Var(name)
    .Arrow(from, to)        // from -> to
    .Int                    // Int
    .Bool                   // Bool
    .List(elem)             // [elem]
    .Pair(fst, snd)         // (fst, snd)
```

Use `Decl.parse` to parse entire declarations, as in `cli.js`. An unfortunate problem with the `adt` library is that its default `console.log` is not very informative. You should use the `pretty()` method instead to get a nice representation of the data type in question. You can also test for deep equality using the `equals()` method.

Here are some examples:

```
>>> Type.Arrow(Type.Var("a"), Type.Var("a")).pretty()
'a -> a'
>>> Expr.Nil.isNil
true
>>> Decl.parse('add (x, y) = x + y').pretty()
'add = (x,y) = ((+) x) y'
```

Note that infix operators have been internally converted to function application.

You will also need to use immutable contexts in this lab. A typing context are maps from names (`String`) to types (`Type`). Here are some useful methods for manipulating them:

```
c = Ctx([{name : n1, type : t1}, ...]) // Construct an immutable context
c.add(n, t) // Returns copy of c with n set to t
c.binds(n)  // Returns true if n is bound in context
c.get(n)    // Returns the type at n in the context
c.toArray() // Returns the underlying array
```

The context constructor can be called with no arguments to create an empty context. See `lib/ctx.js` for more details.

Writing the code The file you will be editing in this lab is `lib/solver.js`.

Task 0: Orientation We've provided some code to help you get started with the type checker. The main entry point is `inferTypes`, which takes a list of `Decl` objects and returns the `Type` of the top-level function defined by the declarations. This function operates in two steps: first, it generates constraint typing rules, calling `constraintTypeOf` on each declaration. The returned typing rules (`tr`) is an object with two properties `type` and `constraints` that respectively correspond to the (not yet reconstructed) type and array of of constraints collected.* Second, it solves the constraints, calling the `solveConstraints` function, which takes a list of constraints and returns a substitution (`Ctx` which contains a mapping from variable names to types). We've provided some glue code for managing constraints between multiple declarations, which we didn't discuss in lecture.

As a warm-up, modify this function so that, prior to constraint solving, your function prints out the list of generated constraints and the type of the top-level declaration. Now when you run `nodejs cli.js should_compile/id.hs`, you should get:

*We use array of two `Type`s to denote a constraint, i.e., `[t1, t2]` encodes that `t1` and `t2` are the same type.

```
_t0 = _t1 -> _t1
function has (unsolved) type: _t0

/path/to/cs242-fall14-types/lib/solver.js:135
    throw new Error("not implemented yet");
          ^
```

This part won't be graded.

Task 1: Constraint generation (expressions) Implement `Expr.prototype.typeOf`. This function takes two arguments: `eq`, the mutable array of constraints being collected, and `ctx`, an immutable context mapping in-scope variable names to types. It returns the type of the overall expression and possibly adds additional constraints to the `eq` array. For example, the type of the literal `4` is `Int`, with no additional constraints. (Alternately, its type is fresh type variable `_t1`, with the constraint that `_t1 = Int`). We've already provided the variable lookup case for you: the type of a variable is specified by looking up the variable in the context `ctx` (or erroring if the variable is not found.)

Since `eq` is just some mutable state, you will need to thread `eq` through recursive invocations of `typeOf`. To add a new constraint, `push` onto `eq`. For example, to state that `t1` and `t2` should be equal:

```
eq.push([t1, t2]);
```

In lecture, we generated fresh type variables for all nodes. You can get a fresh type variable using the `fresh()` function. If you like, you can avoid generating a fresh type variable if the type of an expression is obvious.

After implementing this, you should be able to generate constraints for all test programs which don't involve pattern matching. Be especially sure you have the right constraints for the examples including lambdas. For example, program `should_compile/curried_add.hs`:

```
add x = \y -> x + y
```

generates the following constraints (or something equivalent):

```
_t1 -> _t3 = Int -> Int -> Int
_t2 -> _t4 = _t3
_t0 = _t1 -> _t2 -> _t4
function has (unsolved) type: _t0
```

Task 2: Constraint generation (pattern matching) Implement `Pat.prototype.typeOf`. This function takes two arguments: `eq`, the mutable array of constraints being collected, and `m_ctx`, which mutable context of bindings that the pattern match has brought into scope. Don't worry too much about `m_ctx`, the interesting case for variables is already implemented—just thread it through the recursive subcalls in the same way as `eq`.[†]

Once implemented, you should be able to get constraints for the examples using pattern matching. For instance, `should_compile/redundant_pattern.hs`:

```
foo (x,[])      = 1
foo ((x:xs),[]) = 1
```

[†]If you look at `Decl.prototype.typeOf`, this array will be used to generate the initial `ctx` for the call to `typeOf` on an `Expr`.

generates the following constraints (or something equivalent):

```
_t0 = (_t1, [_t2]) -> Int
[_t4] = _t5
_t3 = ([_t4], [_t6]) -> Int
_t0 = _t3
function has (unsolved) type: _t0
```

Task 3: Occurs check Implement `Type.prototype.occurs`, which takes a name and checks if that name occurs in the type. For example:

```
Type.Arrow(Type.Int, Type.Var("b")).occurs("a")                  === false
Type.Arrow(Type.Var("a"), Type.List(Type.Var("b"))).occurs("b") === true
```

Task 4: Constraint solving Implement `solveConstraints`, which takes an array of constraints and returns a solution for them, i.e., a substitution in the form of context, mapping variable names to types. For example, if you are given the constraints `_t1 = Int` and `Int -> Int = _t3 -> _t4`, you should return the context:

```
Ctx().add("_t1", Type.Int).add("_t3", Type.Int).add("_t4", Type.Int)
```

Note that `Ctx` can take as its argument a list of objects each with properties `name` and `type`.

In the event of a unification error, throw an exception (`throw new UnificationError(constraint)`).

You might find the following program template a useful way to structure your solver:

```
function solveConstraints(todo) {
  var solved = [];
  var c;
  while (c = todo.pop()) {
      ...
  }
  return Ctx(_.map(solved, function (constraint) {
    return {name: constraint[0], type: constraint[1]};
  }));
}
```

We collect constraints into the mutable `solved` array, and then convert (_) `map` function to apply the function converting constraints into variable name-type objects to all the elements of `solved`.

When your solver is working, you should be able to infer types for all of test programs in `should_compile`, and fail with an error for all programs in `should_fail`.

Example success:

```
corn33:~> node cli.js should_compile/test0.hs
[_t1] = _t2
Int -> _t3 = Int -> Int -> Int
_t2 -> _t4 = _t0
_t4 -> _t5 = _t3
_t0 = [_t1] -> _t5
function has (unsolved) type: _t0
type: [_t1] -> Int
```

Example failure:

```
corn33:~> node cli.js should_fail/test1.hs
[_t1] = _t2
_t1 -> _t3 = Int -> Int -> Int
_t2 -> _t4 = _t3
_t0 = [_t1] -> _t4
function has (unsolved) type: _t0

/home/d/courses/cs242-fall14-types/lib/solver.js:209
            throw new UnificationError(c);
                  ^
UnificationError: Could not unify '[Int]' with 'Int'
    ...
```