Implementing uML: Hindley-Milner Type Inference

~~Due Saturday, March 30 at 11:59PM.~~
Deadline extended! New deadline Monday, April 1 at 11:59PM.

In this assignment you will implement Hindley-Milner type inference, which represents the current ``best practice'' for flexible static typing. The assignment has two purposes:

- To help you develop a deep understanding of type inference
- To help you continue to build your ML programming skills

Complete Exercises S and T below, and from Chapter 7 in Ramsey, complete Exercises 1, 2, 4, 13, 16, and 17.

Getting the code

To get the code,

```
git clone linux.cs.tufts.edu:/comp/105/book-code
```

If you have a version that is older than March 26, 2013, you will need to bring your version up to date using `git pull`.

The code you need is in `bare/uml/ml.sml`.

Two exercises to do by yourself (10%)

Complete Exercises 1 and 2 on page 334 of Ramsey.
These exercises explore some implications of type inference. The answers to both exercises should go into file `1-2.uml`; your answer to Exercise 2 should appear in a comment.
In your answer to question 1, **do not use `letrec`**.

Six exercises to do with a partner (90%)

Complete Exercises 4, 13, 16, and 17 from pages 334–337 of Ramsey, and the two exercises S and T below.
For the coding exercises you'll be modifying the interpreter in `book-code/bare/uml/ml.sml`.

S. Test cases for the solver.
Submit three test cases for the constraint solver. At least two of these test cases should be constraints that have no solution. Assuming that *we provide a function* `constraintTest : con -> answer`, put your test cases in file `stest.sml` as three successive calls to `constraintTest`. Do *not* define `constraintTest` yourself.

Here is a sample `stest.sml` file:

```
val _ = constraintTest (TYVAR "a" =*= TYVAR "b")
val _ = constraintTest (CONAPP (TYCON "list", [TYVAR "a"]) =*= TYCON "int")
val _ = constraintTest (TYCON "bool" =*= TYCON "int")
```

Naturally, you will supply your own test cases.

T. Test cases for type inference.
Submit three test cases for type inference. At least two of these test cases should be for terms that fail to type check. Each test case should be a definition written in uML. Put your test cases in a file `ttest.uml`. Here is a sample `ttest.uml` file:

```
(val weird (lambda (x y z) (cons x y z)))
(+ 1 #t)
(lambda (x) (cons x x))
```

Naturally, you will supply your own test cases.

For the remaining exercises, here are some additional remarks and suggestions.

- Understand the most difficult part of a constraint solver.
  Complete Exercise 13 on page 336 of Ramsey.
- Implement a constraint solver
  Complete Exercise 16 on page 337 of Ramsey. Take advantage of the debugging redefinition of `standardize` shown below. Be sure your solver produces the correct result on our three test cases and also on your three test cases.
  This exercise is probably the most difficult part of the assignment. Before proceeding with type inference, try to show your solver code to the course staff.
- Plan for type inference
  Complete Exercise 4 on page 334 of Ramsey. Put your answer in file `rules.pdf`.
- Implement type inference
  Complete Exercise 17 on page 337 Ramsey.

This is one assignment where it pays to run a lot of tests, of both good and bad definitions. *The most effective test of your algorithm is not that it properly assign types to correct terms, but that it reject ill-typed terms.* This assignment is your best chance to earn the large bonuses available by finding bugs in the instructor's code. I have posted a functional topological sort that makes an interesting test case.

Incidentally, if you call your interpreter `ml.sml`, you can build a standalone version in `a.out` by running `mosmlc ml.sml` or a faster version in `ml` by running `mlton -output a.out ml.sml`.

Hints, guidelines, and test code

The type-inference code will be easier to write with the aid of a summary of the μML type system which I have placed online.

The algorithm for standardizing a constraint calls for a lot of case analysis. I found it easier to implement this case analysis by splitting `standardize` into two functions:

- Function `standardize` handles all three forms of constraint: trivial constraints, simple type-equality constraints and conjunctions. When it sees a conjunction, it does a case analysis on the *left* conjunct.
- Function `standardize` does not do anything with simple type equalities. When `standardize` sees a simple type equality, either at top level or as the left child of `/\`, it calls the following function:

  ```
  val standardizeEqAnd : ty * ty * con -> (name * ty) list
  ```

  A call to standardizeEqAnd(***tau_1***, ***tau_2***, *C*) returns a standard form of the constraint ***tau_1*** =*= ***tau_2*** ∧ *C*. This function implements all the rules in Figure 7.1 that relate to simple type-equality constraints.

To help you with the solver, once you have implemented `standardize`, the following code *redefines* standardize into a version that checks itself for sanity. It guarantees that the standard form you produce *implies* the constraint you were given. This is not quite as good as checking for equivalence, but it is a lot better than no checking at all.

```
val standardize = fn c =>
  let val std = standardize c
      val theta = standardSubstitution std
  in  if solves (theta, c) then
        std
      else
        let fun eqAnd ((a, t'), c) = TYVAR a =*= t' /\ c
            val msgs =
                  [ "Constraint ", constraintString (untriviate c), "\n  reduces to"
                  , constraintString (untriviate (foldr eqAnd TRIVIAL std))
                  , "\n  but substitution yields "
```

```
                    , constraintString (untriviate (consubst theta c)), "\n"
                    ]
          in  raise BugInTypeInference (concat msgs)
          end

    end
```

A prudent person might extend this code with an additional test

```
  val _ = if isInStandardForm std then ()
          else raise BugInTypeInference "not in standard form"
```

I would expect *you* to write the function `isInStandardForm`.

With your solver in place, the type inference should be straightforward, with two exceptions: `let` and `letrec`. You can emulate the implementations for `val` and `val-rec`, but **you must split the constraint**. The splitting is covered in detail in the book, on pages 310–312. This math is subtle here, and you may find this part of the book heavy going. The last lecture on type inference will be on how to implement this splitting.

Extra Credit

For extra credit, you may complete any of the following:

- Mutation, as in Exercise 22(a)(b) and possibly (c)
- Explicit types, as in Exercise 23
- Better error messages, as in Exercise 19(a)(b) and possibly (c)
- Tuples, as in Exercise 20
- Generative types, as in Exercise 21

Of these exercises the most interesting are probably Mutation (easy) and Explicit types (not easy).

Testing

The course interpreter is located in `/comp/105/bin/uml`. If your interpreter can process the initial basis and infer correct types, you are doing OK.

The *real* test of your interpreter is that it should reject incorrect definitions. You should prepare a dozen or so definitions that should not type check, and make sure they don't. For example:

```
(val bad (lambda (x) (cons x x)))
(val bad (lambda (x) (cdr (pair x x))))
```

Pick your toughest three test cases to submit for Exercise T.

Avoid common mistakes

Here some common mistakes:

- As you write `standardize`, it's a common mistake to overlook the lessons of Exercise 13. Check the types that are inferred for the functions in the initial basis—if they are too general, the fault is probably here.
- Another common mistake is to create **too many fresh variables** or to fail to constrain your fresh variables.
- Another surprisingly common mistake is to include **redundant cases** in the code for inferring the type of a list literal. As is almost always true of functions that consume lists, it's sufficient to write one case for `NIL` and one case for `PAIR`.
- It's a common mistake to **define a new exception and not handle it**. If you define any new exceptions, make sure they are handled. It's not acceptable for your interpreter to crash with an unhandled exception just because some μML code didn't type-check.

- It's a common mistake to omit the initial basis for testing and then to **forget to include an initial basis in the interpreter you submit**.

There are also some common assumptions which are mistaken:

- It is a mistake to assume that an element of a literal list always has a monomorphic type.
- It is a mistake to assume that `begin` is never empty.

What to submit

For your solo work, run `submit105-ml-inf-solo` to submit file `1-2.uml`.

For your work with a partner, run `submit105-ml-inf-pair` to submit these files:

- `README`, telling us with whom you collaborated, how long you worked, what parts you finished (including any extra credit), and so on.
  Your `README` file should also contain your answer to Exercise 13.
- `stest.sml`, containing your answer to Exercise S
- `ttest.uml`, containing your answer to Exercise T
- `rules.pdf`, containing your answer to Exercise 4
- `ml.sml`, containing a completely interpreter for µML which includes your answers to Exercises 16 and 17.

In the `README`, please tell us what parts of the assignment you have completed, including any extra-credit parts.

Your solutions are going to be evaluated automatically. We must be able to compile your solution in Moscow ML by typing, e.g.,

```
mosmlc ml.sml
```

If there are errors or warnings in this step, your work will earn No Credit for functional correctness.

How your work will be evaluated

Your typing rules will be evaluated using the usual criteria for inference rules, which I won't repeat here.

We will focus most of our evaluation on your constraint solving and type inference.

	Exemplary	**Satisfactory**	**Must improve**
Form	• The code has no offside violations. • *Or*, the code has just a couple of minor offside violations. • Indentation is consistent everywhere. • The submission has no bracket faults. • The submission has a few minor bracket faults. • *Or*, the submission has no bracketed names, but a few bracketed conditions or other faults.	• The code has several offside violations, but course staff can follow what's going on without difficulty. • In one or two places, code is not indented in the same way as structurally similar code elsewhere. • The submission has some redundant parentheses around function applications that are under infix operators (not checked by the bracketing tool)	• Offside violations make it hard for course staff to follow the code. • The code is not indented consistently. • The submission contains more than a handful of parenthesized names as in `(x)` • The submission contains more than a handful of parenthesized `if` conditions.

		• *Or*, the submission contains a handful of bracketing faults. • *Or*, the submission contains more than a handful of bracketing faults, but just a few bracketed names or conditions.	
Names	• Type variables have names beginning with `a`; types have names beginning with `t` or `tau`; constraints have names beginning with `c`; substitutions have names beginning with `theta`; lists of things have names that begin conventionally and end in `s`.	• Types, type variables, constraints, and substitutions mostly respect conventions, but there are some names like `x` or `l` that aren't part of the typical convention.	• Some names misuse standard conventions; for example, in some places, a type variable might have a name beginning with `t`, leading a careless reader to confuse it with a type.
Structure	• The nine cases of simple type equality are handled by these five patterns: `TYVAR`/any, any/`TYVAR`, `CONAPP`/`CONAPP`, `TYCON`/`TYCON`, other. • The code for solving Î±â– â â– Ï has exactly three cases. • The constraint solver is implemented using an appropriate set of helper functions, each of which has a good name and a clear contract. • Type inference for list literals has no redundant case analysis. • Type inference for expressions has no redundant case analysis. • In the code for type inference, course staff see how each part of the code is necessary to implement the algorithm correctly. • Wherever possible appropriate, submission uses `map`, `filter`, `foldr`, and `exists`, either from `List` or from `ListPair`	• The nine cases are handled by nine patterns: one for each pair of value constructors for `ty` • The code for Î±â– â â– Ï has more than three cases, but the nontrivial cases all look different. • The constraint solver is implemented using too many helper functions, but each one has a good name and a clear contract. • The constraint solver is implemented using too few helper functions, and the course staff has some trouble understanding the solver. • Type inference for list literals has one redundant case analysis. • Type inference for expressions has one redundant case analysis. • In some parts of the code for type inference, course staff see some code that they believe is more complex than is required by the typing rules. • Submission sometimes uses a fold where `map`, `filter`, or `exists` could be used.	• The case analysis for a simple type equality does not have either of the two structures on the left. • The code for Î±â– â â– Ï has more than three cases, and different nontrivial cases share duplicate or near-duplicate code. • Course staff cannot identify the role of helper functions; course staff can't identify contracts and can't infer contracts from names. • Type inference for list literals has more than one redundant case analysis. • Type inference for expressions has more than one redundant case analysis. • Course staff believe that the code is significantly more complex than what is required to implement the typing rules. • Submission includes one or more recursive functions that could have been written without recursion by using `map`, `filter`, `List.exists`, or a `ListPair` function.