Type Inference

COMP 105 Assignment

Due Tuesday, April 4, 2017 at 11:59PM

Contents

Overview **1**

Setup **2**

Reading comprehension (10%) **2**

Exercises to do on your own (10%) **4**

Exercises you may do with a partner (80%) **4**

Extra Credit **6**

What and how to submit: Individual problems **6**

What and how to submit: Pair problems **6**

Hints and guidelines **7**
- Building the standalone interpreter 7
- The constraint solver 7
- Type inference 7

Avoid common mistakes **8**

How your work will be evaluated **9**
- Names 10
- Code structure 11

Overview

In this assignment you will implement a constraint-based inference algorithm for the Hindley-Milner type system, 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

Setup

Clone the book code:

```
git clone linux.cs.tufts.edu:/comp/105/build-prove-compare
```

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

Reading comprehension (10%)

These problems will help guide you through the reading. We recommend that you complete them before starting the other problems below. You can download the questions[1]. The last two questions have a lot of parts, but that's because the assignment itself requires that you understand a lot of parts: types, type schemes, substitutions, constraints, constraint solving, and inference. To build up a complete picture, you are best off learning one part at a time.

1. In English, explain the type `∀α, β . α → β.`

   You are ready for exercise 1.

2. Read Sections 7.3.2 and Section 7.4.1, starting on pages page 473 and page 474, respectively. We have seen the symbols ρ, τ, and σ before, but not with exactly this usage.

   (a) In this new context, what does ρ represent?

   How does it differ from the way we used ρ before?

   (b) In this new context, what does τ represent?

   How does it differ from the way we used τ in typed μScheme?

   (c) In this new context, what does σ represent?

   How does it differ from the way we used σ before?

   (d) Say briefly what, in nano-ML, is the difference between τ and σ.

   You are ready for exercise 2. And you are preparing for exercise 19.

3. Read the first two pages of Section 7.4.3, which explain "substitutions" and "instances."

   (a) Write a single substitution that replaces type variables α and β with types `sym` and `bool`, respectively.

   (b) Write a type τ that is an instance of the polymorphic type scheme `∀α . α list → int`.

   You have a foundation on which to get ready for exercises 18 and S.

4. Read the first page of Section 7.5.2, which shows the form of a constraint. Then skip to the first page of Section 7.5.3, which explains how to apply a substitution to a constraint.

   Define substitution `θ = (α₁ ↦ int)` and constraint

   ```
   C = α₁ ~ α₂ /\ α₂ ~ α₃ list /\ α₄ ~ α₃ list list.
   ```

[1] ./cqs.ml-inf.txt

Now define C' as the result of applying θ to C.

(a) Write C':

(b) Do you think C' has a solution? Justify your answer.

(c) What if new substitution `(α₂ ↦ int)` is applied to C? Does the resulting constraint have a solution? Justify your answer.

You are getting ready for exercises 18 and S.

5. Now read all of Section 7.5.3, which explains how to solve constraints.

   To demonstrate your understanding, reason about solving these four constraints:

   ```
   C₁ = α ~ int

   C₂ = α ~ bool

   C₃ = C₁ /\ C₂

   C₄ = α₁ ~ α₂ /\ α₂ list ~ α₁
   ```

   (a) Write a substitution `θ₁` that solves constraint `C₁`:

   (b) Write a substitution `θ₂` that solves constraint `C₂`:

   (c) Does the composition `θ₂ ∘ θ₁` solve constraint `C₃ = C₁ /\ C₂`? Justify your answer.

   (d) Explain why constraint `C₄` can't be solved:

   (e) Can constraint `C₃` be solved? Justify your answer.

   You are ready for excercises 18 and S.

6. Read the paragraphs that describe the typing rules for `lambda` and for "Milner's Let", which you will find on page 481. Don't overlook the small paragraph following the `lambda` rule.

   Now look at the `val` definition of `too-poly` in code chunk 481. The right-hand side of the `val` definition is a `lambda` expression with the name `empty-list` playing the role of `x₁`.

   (a) The rule for `lambda` says that we can pick any type `τ₁` for `empty-list`. After we've chosen `τ₁`, what is the *type scheme* to which `empty-list` (playing `x₁`) is bound in the extended environment which is used to check e?

   (b) Given that the rule for `lambda` says that we can pick any type `τ₁` for `empty-list`, why can't we pick a `τ₁` that makes the `lambda` expression type-check?

   Now look at the definition of `not-too-poly` in code chunk 482. The right-hand side is an example of Milner's `let` with `empty-list` playing the role of x, the literal `'()` playing the role of e', and an application of `pair` playing the role of e. Suppose that Γ □ `'()` : β list, where β is a type variable that does not appear anywhere in Γ. That is to say, the literal `'()` is given the type β list, which is playing the role of `τ`'.

   (c) What are the free type variables of `τ`'?

   (d) What set plays the role of `{ α₁, ..., αₙ }`?

   (e) What is the *type scheme* to which `empty-list` (playing x) is bound in the extended environment which is used to check e?

Look at the VAR rule on page 480 and at the definition of the *instance* relation <: on page 476:

(f) Given the type scheme of `empty-list` in `not-too-poly`, and given that `empty-list` is used in two different places, what type τ do you choose at each use of the VAR rule?

(g) Explain informally why `not-too-poly` type checks.

In exercises 19 and T, you are now ready to implement typing rules for syntactic forms that use `generalize`, including the VAL and LET forms.

Exercises to do on your own (10%)

On your own, please work Exercise 1 on page 522 and Exercise 2 on page 522 of Ramsey. These exercises explore some implications of type inference.

1. *Exploring the meaning of polymorphic types I*. Do Exercise 1 on page 522 of Ramsey.

Related reading: If you need to review quantified types, look at section 6.6.3.

2. *Exploring the meaning of polymorphic types II*. Do Exercise 2 on page 522 of Ramsey.

Related reading: To familiarize yourself with the type system, read section 7.4.1. The rules for evaluating definitions are explained in section 7.3.2. Code chunk 512d can be found at the top of page 513.

Exercises you may do with a partner (80%)

Either on your own or with a partner, please work Exercises 18, 19, and 20 from pages 525–526 of Ramsey, and the two exercises S and T below.

18. *Implementing a constraint solver*. Do Exercise 18 on page 525 of Ramsey. This exercise is probably the most difficult part of the assignment. *Before proceeding with type inference, make sure your solver produces the correct result on our test cases and on your test cases.* Before proceeding to type inference, you may also want to show your solver code to the course staff.

Related reading:

- The second bullet in the opening of Section 7.5, which introduces constraints.
- The opening of Section 7.5.2, which explains constraints and shows them in the typing rules. If you understand the IF rule, you can stop there.
- The explanation of constraint solving in Section 7.5.3.
- The table showing the correspondence between nano-ML's type sytem and code on page 502.
- The definition of `con` and the utility functions in Section 7.6.4.
- The definition of function `solves` on page 508, which you can use to verify solutions your solver claims to find.

S. *Test cases for the solver*. In file `solver-tests.sml`, write 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`, write your test cases as three successive calls to `constraintTest`. Do *not* define `constraintTest` yourself.

Here is a sample set of test cases:

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

Naturally, you will supply your own test cases, different from these.

You can typecheck your file on the Unix command line by running

```
105-check-constraints solver-tests.sml
```

19. *Implementing type inference*. Do Exercise 19 on page 526 of Ramsey.

- Even though you won't be writing all the cases yourself, recapitulate the same step-by-step procedure used for Typed μScheme[2]. Especially remember to disable the predefined functions at the start and to re-enable them at the end.
- Adapt your regression tests from the Typed μScheme homework[3]. Use `check-principal-type` instead of `check-type`.

Related reading:

- The nondeterministic typing rules of nano-ML, in Section 7.4.5, which starts on page 482 of Ramsey
- The constraint-based typing rules in section 7.5.2
- The summaries of the typing rules on pages page 529 to page 530
- Explanation and examples of `check-type` and `check-principal-type` in Section 7.4.6, which starts on page 483

T. *Test cases for type inference*. Create a file `type-tests.nml`, and in that file, write three unit tests for nano-ML type inference. At least two of these tests must use `check-type-error`. The third may use either `check-type-error` or `check-principal-type`. If you wish, your file may include `val` bindings or `val-rec` bindings of names used in the tests. *Your file must load and pass all tests using the reference implementation of nano-ML:*

```
nml -q < type-tests.nml
```

If you submit more than three tests, we will use only the *first* three.

Here is a complete example `type-tests.nml` file:

```
(check-type-error (lambda (x y z) (cons x y z)))
(check-type-error (+ 1 #t))
(check-type-error (lambda (x) (cons x x)))
```

[2] typesys.html#how-to-build-a-type-checker
[3] typesys.html

You will supply your own test cases, different from these.

Related reading:

- Concrete syntax for types and for unit tests, in Figure 7.1 on 468
- As above, the explanation and examples of `check-type` and `check-principal-type` in Section 7.4.6, which starts on page 483.

20. *Adding primitives*. Do Exercise 20 on page 526 of Ramsey.

Related reading: Read about primitives in section 7.6.7.

Extra Credit

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

- Mutation, as in Exercise 23(a), (b), and possibly (c)
- Explicit types, as in Exercise 25
- Better error messages, as in Exercise 24(a), (b), and possibly (c)

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

What and how to submit: Individual problems

Submit these files:

- A `README` file containing
  - The names of the people with whom you collaborated
- A file `cqs.ml-inf.txt` containing your answers to the reading-comprehension questions.
- A file `meaning.nml` containing your code for Exercise 1 on page 522 and Exercise 2 on page 522. Your answers to Exercise 2 should appear in a comment.

As soon as you have the files listed above, run `submit105-ml-inf-solo` to submit a preliminary version of your work. Keep submitting until your work is complete; we grade only the last submission.

What and how to submit: Pair problems

Submit these files:

- A `README` file containing
  - The names of the people with whom you collaborated
  - The numbers of any extra credit problems you solved
- File `ml.sml`, implementing a complete interpreter for nano-ML which includes your answers to Exercises 18, 19, and 20.
- File `regression.nml` containing regression tests for your type inference
- File `solver-tests.sml`, containing your answer to Exercise S

- File `type-tests.nml`, containing your answer to Exercise T

As soon as you have the files listed above, run `submit105-ml-inf-pair` to submit a preliminary version of your work. Keep submitting until your work is complete; we grade only the last submission.

Hints and guidelines

Building the standalone interpreter

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`. Than you can run it with `./a.out` and you can run tests by

```
./a.out -q < type-tests.nml         # required
./a.out -q < more-type-tests.nml    # optional
```

The constraint solver

To help you with the solver, once you have implemented `solve`, the following code redefines `solve` into a version that checks itself for sanity (ie, idempotence). It is a good idea to check that the substitution returned by your solver is idempotent before using it in your type inferencer.

```
fun isIdempotent pairs =
    let fun distinct a' (a, tau) = a <> a' andalso not (member a' (freetyvars tau))
        fun good (prev', (a, tau)::next) =
              List.all (distinct a) prev' andalso List.all (distinct a) next
              andalso good ((a, tau)::prev', next)
          | good (_, []) = true
    in  good ([], pairs)
    end

val solve =
    fn c => let val theta = solve c
            in  if isIdempotent theta then theta
                else raise BugInTypeInference "non-standard substitution"
            end
```

Type inference

With your solver in place, type inference should be mostly straightforward.

Follow the same step-by-step procedure[4] you used to build your type checker for Typed μScheme. In particular,

- Start by disabling the initial basis.

[4] ./typesys.html#how-to-build-a-type-checker

- Build on the partially complete implementation of `typeof` from the book.
- Build your implementation of `literal` just as you did for Typed μScheme: numbers, symbols, and Booleans first.
- Create a file of regression tests. Start with literals.
- Look at each case in the code that raises `LeftAsExercise`. Fix these cases one at a time. At each step, add to your regression suite, and run all the tests. Whenever possible, include `check-type-error` tests.
- The two difficult cases are `let` and `letrec`. You can emulate the implementations for `val` and `val-rec`, but **you must split the constraint** into local and global portions. The splitting is covered in detail in the book in the section on "Generalization in Milner's let binding", which is part of Section 7.5.2. Look especially at the sidebar "Generalization with constraints" on page 495.
- Implement list literals toward the end.
- Before you submit your code, re-enable the initial basis and make sure your interpreter infers the proper types for the predefined functions of nano-ML. Write `check-principal-type` tests for functions `map`, `filter`, `exists?`, and `foldr`.

It pays to create a lot of regression tests, of both the `check-principal-type` and the `check-type-error` variety. (The `check-type` test also has its place, but for this assignment, you want to stick to `check-principal-type`.) *The most effective tests of your algorithm will use* `check-type-error`. Assigning types to well-typed terms is good, but most mistakes are made in code that should reject an ill-typed term, but doesn't. Here are some examples of the sorts of tests that are really useful:

```
(check-type-error (lambda (x) (cons x x)))
(check-type-error (lambda (x) (cdr (pair x x))))
```

Once your interpreter is rejecting ill-typed terms, if it can process the predefined functions and infer their principal types correctly, you are doing well. As a larger integration test, I have posted a functional topological sort[5]. It contains some type tests as well as a `check-expect`.

Avoid common mistakes

A 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 nano-ML code didn't type-check.

It's not actually a common mistake, but don't try to handle the exception `BugInTypeInference`. If this exception is raised, your interpreter is *supposed* to crash.

It's a common mistake to disable the predefined functions for testing and then to **submit your interpreter without re-enabling the predefined functions**.

It's a common mistake to call `ListPair.foldr` and `ListPair.foldl` when what you really meant was `ListPair.foldrEq` or `ListPair.foldlEq`.

[5] ../progs/tsort.nml

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.

How your work will be evaluated

Your constraint solving and type inference will be evaluated through extensive testing. 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.

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

Names

We expect you to pay attention to names:

	Exemplary	**Satisfactory**	**Must Improve**
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.

Code structure

We expect you to pay even more attention to *structure*. Keep the number of cases to a minimum!

	Exemplary	**Satisfactory**	**Must Improve**
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 $\alpha \sim \tau$ 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 $\alpha \sim \tau$ 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 $\alpha \sim \tau$ 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.