Type Inference

COMP 105 Assignment

Due Tuesday, April 2, 2019 at 11:59PM

Contents

Overview

Many programmers want the flexibility of an untyped scripting language and the reliability of a statically typed language, combined. This combination is provided by type inference. You are most likely to encounter it in Webby languages like Hack, TypeScript, and Elm, but it is also heavily used in systemsy languages like Haskell and OCaml, as well as researchy languages like Agda, Idris, and Coq/Gallina. All these languages, and many more to come, are based on the Hindley-Milner type system, which you will implement in this homework.

Setup

Clone the book code:

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

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

Dire warnings

The usual prohibitions against `open`, `null`, `hd`, `tl`, `length`, and so on continue to apply.

Except possibly as an argument to `map` (which we recommend against), **none of the code you write may use `fst` or `snd`.**[1] You may not define and use a helper function with the same contract as `fst` or `snd`. Submissions violating this rule will earn **No Credit**.

What are you supposed to do? Pattern match:

```
val (left, right) = ... expression that evaluates to a pair ...
```

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[2].

1. (Not on the reading.) Based on feedback you received on previous homeworks, what's *one* thing you did wrong that you can correct, or one skill you'd like to improve on?

2. Read sections 7.3.2 and 7.4.1, which start on pages page 486 and page 487, respectively.
   We have seen the symbols ρ, τ, and σ before, but not used exactly in this way.

   Here is a list of semantic and type-related concepts you have seen written using single symbols:

   - an expression
   - a name
   - a location
   - a value
   - a type
   - a type scheme (new in this chapter)

[1] These functions are defined and used in the interpreter *only* to be passed to higher-order functions. They are never called directly.

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

- a mapping from names to locations
- a mapping from names to values
- a mapping from names to types
- a mapping from names to type schemes (new in this chapter)

There are lots of concepts and only so many symbols to go around. Please identify, from the preceding list, what each symbol stands for in the theory of nano-ML:

(a) ρ

(b) τ

(c) σ

(d) Γ

And finally,

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

You are preparing for exercise 19.

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

   (a) Yes or no: does the substitution `(α → sym) ∘ (β → bool) ∘ (γ → int)` replace type variable `α` with type `sym`?

   (b) Yes or no: does the substitution `(α → sym) ∘ (β → bool) ∘ (γ → int)` replace type variable `β` with type `bool`?

   (c) Yes or no: does the substitution `(α → sym) ∘ (β → bool) ∘ (γ → int)` leave the type `γ list` unchanged?

   (d) Which of the following are *instances* of the polymorphic type scheme `∀α . α list → int`? For each one, please indicate whether it is an instance of the type scheme (True) or whether it is not an instance of the type scheme (False).

   `int list` True or False

   `int list list` True or False

   `int list list → int` True or False

   `int * int list → list` True or False

You have a foundation on which to build for exercises 18 and C.

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.

   We start with a substitution θ and a constraint C:

   $\theta =$ `(α₁ ↦ int)`

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

   Now define $C' = \theta(C)$.

   (a) Write C':

(b) Does C' have a solution? Answer yes or no.

Now define C'' as the result of applying substitution `(α₂ ↦ int)` to C.

(c) Write C'':

(d) Does C'' have a solution? Answer yes or no.

You are getting ready for exercises 18 and C.

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`? Answer yes or no.

   (d) Can constraint `C₃` be solved? Answer yes or no.

   (e) Can constraint `C₄` be solved? Answer yes or no.

   You are ready for excercises 18 and C.

6. Read the first two pages of section 7.5.2, which starts on page 502. Pay special attention to the Apply rule. Also read the footnote at the bottom of page 26 of *Seven Lessons in Program Design*[3].

   Now consider type inference for the following expression e:

   ```
   (f 3 #t)
   ```

   For this question, assume the following:

   - Expression `3` has type `int`, with a trivial constraint.
   - Expression `#t` has type `bool`, with a trivial constraint.
   - Trivial constraints can be ignored.
   - Every type variable *except* `'a`, `'b`, and `'c` is "fresh."

   Answer both parts:

   (a) Assume that `f` is bound in Γ to the type scheme `∀.'a × 'b → 'c.` (The `∀` is supposed to be empty.) In judgment C, Γ ⊢ e : τ, what does the type checker output for `τ`?

   And what does the type checker output for C?

   (b) Assume that `f` is bound in Γ to the type scheme `∀.'a`. In judgment C, Γ ⊢ e : τ, what does the type checker output for `τ`?

   And what does the type checker output for C?

[3] ../design/lessons.pdf

You are ready for the easy parts of exercise 19.

7. Read the paragraphs that describe the nondeterministic typing rules for `lambda` and for "Milner's Let", which you will find on page 495. Especially, read the small paragraph following the `lambda` rule.

   Now look at the `val` definition of `too-poly` in code chunk 495. The right-hand side of the `val` definition is a `lambda` expression with the name `empty-list` playing the role of x_1.

   (a) The rule for `lambda` says that we can pick any type τ_1 for `empty-list`. After we've chosen τ_1, what is the *type scheme* to which `empty-list` (playing x_1) is bound in the extended environment which is used to check e? (Hint: this type scheme comes from the `lambda` rule, as per the discussion in the small paragraph, and it is *different* from the type scheme of the `empty-list` that appears in the top-level `val` binding.)

   (b) Given that the rule for `lambda` says that we can pick any type τ_1 for `empty-list`, why can't we pick a τ_1 that makes the `lambda` expression type-check? Put the word YES next to the best explanation:

   - Parameter `empty-list` has to have type `(forall ('a) (list 'a))`, but τ_1 is not a `forall` type.
   - Parameter `empty-list` has type τ_1 = `(list 'a)`, which is not the same as `(list bool)`.
   - Parameter `empty-list` can have any type τ_1 but no τ_1 can be equivalent to both `(list int)` and `(list bool)`.
   - Parameter `empty-list` has type τ_1 = `(list bool)`, which is not the same as `(list int)`.
   - Parameter `empty-list` has type τ_1 = `(list int)`, which is not the same as `(list bool)`.

   You are ready for exercise 3 and for one of the hard parts of exercise 19.

8. Now look at the definition of `not-too-poly` in code chunk 496. 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 $\Gamma \vdash$ `'()` : 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'.

   (a) If τ' is β list, what are its free type variables?

   (b) What set plays the role of { α_1, ..., α_n }, which is ftv(τ')-ftv(Γ)?

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

   You are ready for all of exercise 19.

Exercises you may do with a partner (90%)

Either on your own or with a partner, please work Exercises 3, 18, 19, and 20 from pages 536 to 540 of *Build, Prove, and Compare*, and the two exercises C and T below.

3. *Algorithmic rules for* Begin *and* Lambda*.* Do exercise 3 on page 536 of *Build, Prove, and Compare*. This exercise fills in a key step between the nondeterministic rules in the book and the deterministic rules you will need to implement type inference.

Please put your solution in file `rules.pdf`.

Hints:

- In your Begin rule, emulate the constraint-based rules for If and TypesOf that you will find in section 7.5.2, which starts on page 502.
- To write a Lambda rule, you will need to figure out what to put in the environment in place of the unknown types $\tau_1, \ldots, \tau_n$, and what to do with the constraints you get back from the recursive call.

  Like Let, Lambda introduces new variables into the typing environment Γ. But Lambda is much simpler, because it does not "generalize" any types.

Related reading: The first part of section 7.5.2, which starts on page 502, up to and including the part labeled "Converting nondeterministic rules to use constraints."

18. *Implementing and testing a constraint solver.* Do exercise 18 on page 539 of *Build, Prove, and Compare*. 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.* You may also want to show your solver code to the course staff.

Testing: Your constraint solver can be tested only by internal `Unit` tests. To help with this testing, here are some useful functions:

```
val eqsubst : subst * subst -> bool   (* arguments are equivalent *)
val hasSolution     : con -> bool
val hasNoSolution   : con -> bool
val hasGoodSolution : con -> bool
val solutionEquivalentTo : con * subst -> bool
       (* solution to constraint is equivalent to subst *)
```

You will use these functions in `Unit` tests, as in the following examples:

```
val () = Unit.checkAssert "int ~ bool cannot be solved"
         (fn () => hasNoSolution (inttype ~ booltype))

val () = Unit.checkAssert "bool ~ bool can be solved"
         (fn () => hasSolution (booltype ~ booltype))

val () = Unit.checkAssert "bool ~ bool is solved by the identity substitution"
         (fn () => solutionEquivalentTo (booltype ~ booltype, idsubst))

val () = Unit.checkAssert "bool ~ 'a is solved by 'a |--> bool"
         (fn () => solutionEquivalentTo (booltype ~ TYVAR "'a",
                                         "'a" |--> booltype))
```

You will want additional tests—at least one for each of the nine cases in the constraint solver. To get you started, here are two more constraints:

```
TYVAR "a" ~ TYVAR "b" /\ TYVAR "b" ~ TYCON "bool"
```

```
CONAPP (TYCON "list", [TYVAR "a"]) ~ TYCON "int"
```

The useful functions are implemented by this code, which you will need to copy:

```
fun eqsubst (theta1, theta2) =
  let val domain  = union (dom theta2, dom theta1)
      fun eqOn a = (varsubst theta1 a = varsubst theta2 a)
  in  List.all eqOn domain
  end

fun hasSolution     c = (solve c; true) handle TypeError _ => false
fun hasGoodSolution c = solves (solve c, c) handle TypeError _ => false
val hasNoSolution : con -> bool = not o hasSolution
fun solutionEquivalentTo (c, theta) = eqsubst (solve c, theta)
```

Related reading:

- Section 7.4.1, which starts on page 487. It will familiarize you with the type system.
- The second bullet in the opening of section 7.5, which introduces constraints.
- The opening of section 7.5.2, which starts on page 502. This section explains constraints and shows them in the typing rules. If you understand the constraint-based IF rule, in both its simple form and its TypesOf form, you can stop there.
- The explanation of constraint solving in section 7.5.3, which starts on page 511.
- The table showing the correspondence between nano-ML's type sytem and code on page 516.
- The definition of `con` and the utility functions in section 7.6.4, which starts on page 520.
- The definition of function `solves` on page 522, which you can use to verify solutions your solver claims to find.

C. *Difficult constraints*. In file `constraints.sml`, write three constraints that are difficult to solve. At least two should have no solution. Write your constraints in a list in a single `val` definition of `constraints`:

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

Supply your own test cases, different from these. You are welcome to reuse constraints from your solver's unit tests.

To make sure it is well formed, typecheck your file by running the Unix command

```
105-check-constraints constraints.sml
```

19. *Implementing type inference*. Do exercise 19 on page 540 of *Build, Prove, and Compare*. Submit your solution as part of the interpreter source file `ml.sml`,

- Even though you won't be writing all the cases yourself, recapitulate the same step-by-step procedure used for Typed μScheme[4]. Especially remember to disable the predefined functions[5] at the start and to re-enable them at the end.
- We recommend against using `Unit` tests for this problem. Instead, create regression tests, which we recommend that you adapt from the Typed μScheme homework[6]. But don't use `check-type`; instead, use `check-principal-type`.

  Please put your regression tests in file `regression.nml`.

Related reading:

- The nondeterministic typing rules of nano-ML, which start on page 494 of *Build, Prove, and Compare*.
- The constraint-based typing rules in section 7.5.2
- The summaries of the typing rules from page 543 to page 544
- Explanation and examples of `check-type` and `check-principal-type` in section 7.4.6, which starts on page 497

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)))
```

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

Related reading:

- Concrete syntax for types and for unit tests, in Figure 7.1 on 482
- As above, the explanation and examples of `check-type` and `check-principal-type` in section 7.4.6, which starts on page 497.

20. *Adding primitives*. Do exercise 20 on page 540 of *Build, Prove, and Compare*.

Related reading: Read about primitives in section 7.6.7.

[4] typesys.html#how-to-build-a-type-checker
[5] typesys.html#disable_predefined
[6] typesys.html

Extra Credit

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

- Exercise 1 on page 536
- Mutation, as in exercise 23(a), (b), and possibly (c)

  For 23(b), please put the code in your README file.
- Better error messages, as in exercise 24(a), (b), and possibly (c)
- Explicit types, as in exercise 25

If you work with a partner on the main problems but you complete extra credit by yourself, please let us know in your README file.

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

What and how to submit: Reading comprehension

Using `submit105-ml-inf-solo`,submit this file:

- A file `cqs.ml-inf.txt` containing your answers to the reading-comprehension questions

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
- A file `rules.pdf` containing your constraint-based typing rules for Begin and Lambda
- 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 `constraints.sml`, containing your answer to Exercise C
- 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

Testing

If you call your interpreter `ml.sml`, you can build a standalone version in `a.out` by running

```
mosmlc -I /comp/105/lib ml.sml
```

Don't overlook the "`c`" at the end of `mosmlc`. Now you can run your interpreter with `./a.out`, and you can run tests by

```
./a.out -q < /dev/null              # runs Unit tests
./a.out -q < regression.nml         # runs required regression tests
./a.out -q < type-tests.nml         # runs three selected tests (required)
```

The nine-step design process

Working on larger codes, it's easy to lose track of the design process. Here's what we recommend:

- For the type checker, use the specialized techniques described in design lesson 5 (program design with typing rules).
- For the constraint solver, **the standard nine-step process applies**. In particular,
  - There are 9 forms of simple type-equality constraint (formed with the ~ value constructor). For most forms, You will want two examples: one that is solvable and one that is not. (Some forms have only unsolvable examples.) You will want a unit test for each.
  - There are also two other forms of constraint: conjunction constraints and the trivial constraint. You will want many examples of conjunction constraints, but to develop these examples, you will rely less on forms of data and more about ideas on substitution that you will explore in recitation.

The constraint solver

A simple type-equality constraint has nine possible cases. We recommend unit testing each one. Not all cases are solvable, but for each case that may be solvable, we recommend two tests: one on a solvable constraint and one on an unsolvable constraint.

We also recommend unit testing the conjunction case. Examples from the book are a good place to start.

Once you have passed unit tests, we recommend an additional sanity check: The following code redefines `solve` into a version that checks itself for sanity (ie, idempotence). To make sure that every solution generated during type inference is in fact sane, use this code *before* `typeof`.

```
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-idempotent substitution"
            end
```

Type inference

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

Follow the same step-by-step procedure[7] you used to build your type checker for Typed μScheme. In particular,

- Start by disabling the predefined functions[8].
- 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 509.
- Implement list literals toward the end.
- Before you submit your code, re-enable the predefined functions 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[9]. It contains some type tests as well as a `check-expect`.

Debugging

If you need to look at internal data structures, I suggest using `eprint` and `eprintln` to print values. These functions expect strings, which you can produce using these functions:

```
val expString   : exp -> string
val defString   : def -> string
val typeString  : ty -> string
val constraintString : con -> string
val substString : subst -> string
```

The first four functions are included in the interpreter's source code. You'll need to define the fifth as follows:

```
fun substString pairs =
```

[7] ./typesys.html#how-to-build-a-type-checker
[8] typesys.html#disable_predefined
[9] ../progs/tsort.nml

```
separate ("idsubst", " o ")
(map (fn (a, t) => a ^ " |--> " ^ typeString t) pairs)
```

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 true of almost every function that consumes a list, 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. **Ouch!**

It's a common mistake to call `ListPair.foldr` and `ListPair.foldl` when what you really meant was `ListPair.foldrEq` or `ListPair.foldlEq`. The same applies to `zip` and `map`; you want `ListPair.zipEq` and `ListPair.mapEq`.

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 -I /comp/105/lib 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.