Type Inference

CS 105 Assignment

Due Tuesday, November 15, 2022 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.

The homework exercises (but not reading-comprehension questions) may be done with a partner.

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 ...
```

Your `ml.sml` must pass `sml-lint` with no complaints:

```
  sml-lint ml.sml
```

Your internal `Unit` testing must not issue any messages. In particular,

- Your interpreter must not fail any internal `Unit` tests.
- Your interpreter must not call `Unit.report`. (Call `Unit.reportWhenFailures` instead.)

Reading comprehension

Before starting the programming problems, answer the reading-comprehension questions for this module, which you will find online. It is also OK to alternate between reading-comprehension questions and related homework questions.

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

Exercises, which you may do with a partner

Either on your own or with a partner, please work Exercises 8, 18, 19, and 21 from pages 457 to 460 of *Build, Prove, and Compare*, and the two exercises C and T below.

---

8. *Algorithmic rules for* Begin *and* Lambda. Do exercise~8 on page~457 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~430.
- 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~430, 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~459 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.

Remember that your ML code must compile *without errors or warnings*, and it must be accepted without comment by `sml-lint`.

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

```
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"
```

Before you can run unit tests, you will need to disable the predefined functions in the interpreter.

Related reading:

- Section~7.4.1, which starts on page~418, which 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~430. 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~438.
- The table showing the correspondence between nano-ML's type sytem and code on page~443.
- The definition of `con` and the utility functions in section~7.6.3, which starts on page~447.
- The definition of function `solves` on page~448, 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.

By itself, this file won't compile. **Don't try to fix it by copying the definition of `con`.** Instead, typecheck your file by running the Unix command

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

---

19. *Implementing type inference*. Do exercise~19 on page~460 of *Build, Prove, and Compare*. Submit your solution as part of the interpreter source file `ml.sml`,

Remember that your ML code must compile *without errors or warnings*, and it must be accepted without comment by `sml-lint`.

- Even though you won't be writing all the cases yourself, recapitulate the same step-by-step procedure used for Typed μScheme. Especially remember to disable the predefined functions 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. 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~423 of *Build, Prove, and Compare*.
- The constraint-based typing rules in section~7.5.2
- The summaries of the typing rules from page$_{463 \text{ to page}}$464
- Explanation and examples of `check-type` and `check-principal-type` in section~7.4.6, which starts on page~427

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 414
- As above, the explanation and examples of `check-type` and `check-principal-type` in section~7.4.6, which starts on page~427.

---

21. *Adding primitives*. Do exercise~21 on page~460 of *Build, Prove, and Compare*. (In this exercise you extend nano-ML with a pair type and with primitive functions `pair`, `fst`, and `snd`.)

Related reading: Read about primitives in section 7.6.5

Representation of pairs
You may represent a value of pair type any way you like. Quite a few students like to use `PAIR`, for example. **Your representation may print differently from the one in `/comp/105/bin/nml`, and that's OK.** The main thing is for your code to have the right types, and for it to obey the defining algebraic laws:

```
(fst (pair x y)) == x
(snd (pair x y)) == y
```

If these laws hold, then even if a human being sees a difference in the output, there is no nano-ML program that can tell the difference between one representation and another. They are "observationally equivalent."

Extra Credit

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

- Exercise~1 on page~456. Put your answers in your README file.
- Mutation, as in exercise~23(a) and (b). For additional credit you can add (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

Except as noted above, just add your answers to your file `ml.sml`.

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

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 21.
- 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 internal Unit tests
./a.out -q < regression.nml         # runs required regression tests
./a.out -q < type-tests.nml         # runs three selected tests (required)
```

A common mistake is to run *your* regression tests against *my* working `nml` interpreter. To help you test your own code, we provide a script called `test-my-nml`. If you're running on the server and have run `use comp105`, then just run the command

```
test-my-nml
```

The script searches the current working directory for your compiled nano-ML interpreter, which must be called `a.out`. Then it runs `a.out` on your `regression.nml` file. (If the interpreter or the regression tests are missing, the script complains.) Or you can supply another test file:

```
test-my-nml moretests.nml
```

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,
  - As presented in lecture, 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. Start with examples from the book.

Once you have passed unit tests, we recommend an additional consistency check: The following code redefines `solve` into a version that checks itself for idempotence. To make sure that every solution generated during type inference is in fact idempotent, 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 you used to build your type checker for Typed μScheme. In particular,

- Start by disabling the predefined functions.
- 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~436.
- 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:

```
(check-principal-type map    (forall ['a 'b] (('a -> 'b) (list 'a) -> (list 'b))))
(check-principal-type  filter   (forall  ['a]      (('a  ->  bool)  (list  'a)  -
> (list 'a))))
(check-principal-type exists? (forall ['a]    (('a -> bool) (list 'a) -> bool)))
(check-principal-type foldr  (forall ['a 'b] (('a 'b -> 'b) 'b (list 'a) -> 'b)))
```

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)))) ; only after you implement pair
```

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, we have posted a functional topological sort. It contains some type tests as well as a `check-expect`.

Debugging

If you need to look at internal data structures, use functions `eprint` and `eprintln` to print values. These printing functions expect strings, which you can produce using these string-conversion functions:

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

All these functions are included in the interpreter's source code.

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.