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.

Dire warning

Except possibly as an argument to map (which we recommend against), none of the code you write may use \texttt{fst} or \texttt{snd}.\footnote{These functions are defined and used in the interpreter only to be passed to higher-order functions. They are never called directly.} You may not define and use a helper function with the same contract as \texttt{fst} or \texttt{snd}. Submissions violating this rule will earn \textbf{No Credit}.

What are you supposed to do? Pattern match:

\begin{verbatim}
val (left, right) = ... expression that evaluates to a pair ...
\end{verbatim}

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\footnote{/cqs.ml-inf.txt}.

1. Read sections 7.3.2 and 7.4.1, which start on pages 480 and 481, respectively.

We have seen the symbols $\rho$, $\tau$, and $\sigma$ 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:

\begin{itemize}
  \item an expression
  \item a name
  \item a location
  \item a value
  \item a type
  \item a type scheme (new in this chapter)
  \item a mapping from names to locations
  \item a mapping from names to values
  \item a mapping from names to types
  \item a mapping from names to type schemes (new in this chapter)
\end{itemize}

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:
And finally,

(e) Say briefly what, in nano-ML, is the difference between $\tau$ and $\sigma$:

You are preparing for exercise 19.

2. Read the first two pages of section 7.4.3, which explain “substitutions” and “instances.”

(a) Yes or no: does the substitution $(\alpha \rightarrow \text{sym}) \circ (\beta \rightarrow \text{bool}) \circ (\gamma \rightarrow \text{int})$ replace type variable $\alpha$ with type sym?

(b) Yes or no: does the substitution $(\alpha \rightarrow \text{sym}) \circ (\beta \rightarrow \text{bool}) \circ (\gamma \rightarrow \text{int})$ replace type variable $\beta$ with type bool?

(c) Yes or no: does the substitution $(\alpha \rightarrow \text{sym}) \circ (\beta \rightarrow \text{bool}) \circ (\gamma \rightarrow \text{int})$ leave the type $\gamma$ list unchanged?

(d) Which of the following are instances of the polymorphic type scheme $\forall \alpha . \alpha \text{ list } \rightarrow \text{int}$? 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 get ready for exercises 18 and S.

3. 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 $\theta$ and a constraint $C$:

\[
\theta = (\alpha_1 \rightarrow \text{int})
\]
\[
C = \alpha_1 \sim \alpha_2 /\ \alpha_2 \sim \alpha_3 \text{ list } /\ \alpha_4 \sim \alpha_3 \text{ list list}.
\]

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 $(\alpha_2 \rightarrow \text{int})$ to $C$.

(c) Write $C''$:

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

You are getting ready for exercises 18 and S.
4. Now read all of section 7.5.3, which explains how to solve constraints.

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

\[
\begin{align*}
C_1 &= \alpha \sim \text{int} \\
C_2 &= \alpha \sim \text{bool} \\
C_3 &= C_1 \land C_2 \\
C_4 &= \alpha_1 \sim \alpha_2 \land \alpha_2 \text{ list } \sim \alpha_3 \\
\end{align*}
\]

(a) Write a substitution \( \theta_1 \) that solves constraint \( C_1 \):

(b) Write a substitution \( \theta_2 \) that solves constraint \( C_2 \):

(c) Does the composition \( \theta_2 \circ \theta_1 \) solve constraint \( C_3 = C_1 \land C_2 \)? Answer yes or no.

(d) Can constraint \( C_3 \) be solved? Answer yes or no.

(e) Can constraint \( C_4 \) be solved? Answer yes or no.

You are ready for exercises 18 and S.

5. Read the paragraphs that describe the nondeterministic typing rules for \( \lambda \) and for "Milner’s Let", which you will find on page 489. Especially, read the small paragraph following the \( \lambda \) rule.

Now look at the \texttt{val} definition of \texttt{too-poly} in code chunk 489. The right-hand side of the \texttt{val} definition is a \texttt{lambda} expression with the name \texttt{empty-list} playing the role of \( x_1 \).

(a) The rule for \texttt{lambda} says that we can pick any type \( \tau_1 \) for \texttt{empty-list}. After we’ve chosen \( \tau_1 \), what is the \textit{type scheme} to which \texttt{empty-list} (playing \( x_1 \)) is bound in the extended environment which is used to check \( e \)? (Hint: this type scheme comes from the \texttt{lambda} rule, as per the discussion in the small paragraph, and it is different from the type scheme of the \texttt{empty-list} that appears in the top-level \texttt{val} binding.)

(b) Given that the rule for \texttt{lambda} says that we can pick any type \( \tau_1 \) for \texttt{empty-list}, why can’t we pick a \( \tau_1 \) that makes the \texttt{lambda} expression type-check? Put the word \texttt{YES} next to the best explanation:

- Parameter \texttt{empty-list} has to have type \( \forall \alpha (\text{list } \alpha) \), but \( \tau_1 \) is not a \( \forall \) type.
- Parameter \texttt{empty-list} has type \( \tau_1 = (\text{list } \alpha) \), which is not the same as \( (\text{list bool}) \).
- Parameter \texttt{empty-list} can have any type \( \tau_1 \) but no \( \tau_1 \) can be equivalent to both \( (\text{list int}) \) and \( (\text{list bool}) \).
- Parameter \texttt{empty-list} has type \( \tau_1 = (\text{list bool}) \), which is not the same as \( (\text{list int}) \).
- Parameter \texttt{empty-list} has type \( \tau_1 = (\text{list int}) \), which is not the same as \( (\text{list bool}) \).

6. Now look at the definition of \texttt{not-too-poly} in code chunk 490. The right-hand side is an example of Milner’s \texttt{let} with \texttt{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 \( \Gamma \vdash '() : \beta \text{list} \), where \( \beta \) is a type variable that does not appear anywhere in \( \Gamma \). That is to say, the literal `'()` is given the type \( \beta \text{list} \), which is playing the role of \( \tau' \).

(a) If \( \tau' \) is \( \beta \text{list} \), what are its free type variables?

(b) What set plays the role of \( \{ \alpha_1, \ldots, \alpha_n \} \), which is \( \text{ftv}(\tau')-\text{ftv}(\Gamma) \)?

(c) What is the type scheme to which \( \text{empty-list} \) (playing \( x \)) is bound in the extended environment which is used to check \( e \)?

Exercises to do on your own (10%)

On your own, please work exercise 1 on page 530 and exercise 2 on page 530 of Build, Prove, and Compare. These exercises explore some implications of type inference.

1. Exploring the meaning of polymorphic types I. Do exercise 1 on page 530 of Build, Prove, and Compare.

   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 530 of Build, Prove, and Compare.

   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.

Exercises you may do with a partner (80%)

Either on your own or with a partner, please work Exercises 3, 18, 19, and 20 from pages 530 to 534 of Build, Prove, and Compare, and the two exercises S and T below.

3. Algorithmic rules for Begin and Lambda. Do exercise 3 on page 530 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.

   Hints:

   • Even though the Begin rule appears second, do it first. You will want to emulate the constraint-based rules for If and TypesOf that you will find in section 7.5.2, which starts on page 496.

   • For the 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 \( \Gamma \). 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 496, up to and including the part labeled “Converting nondeterministic rules to use constraints.”
18. **Implementing a constraint solver.** Do exercise 18 on page 533 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.

**Related reading:**

- Section 7.4.1, which starts on page 481. 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 498. 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 507.
- The table showing the correspondence between nano-ML’s type system and code on page 510.
- The definition of con and the utility functions in section 7.5.4, which starts on page 514.
- The definition of function solves on page 516, 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:

```sml
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 534 of *Build, Prove, and Compare*.

- Even though you won’t be writing all the cases yourself, recapitulate the same step-by-step procedure used for Typed μScheme3. 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 homework4. Use `check-principal-type` instead of `check-type`.

**Related reading:**

- The nondeterministic typing rules of nano-ML, which start on page 488 of *Build, Prove, and Compare*.
- The constraint-based typing rules in section 7.5.2

---

3 typesys.html#how-to-build-a-type-checker
4 typesys.html
• The summaries of the typing rules from page 537 to page 538
• Explanation and examples of check-type and check-principal-type in section 7.4.6, which starts on page 491

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 will supply your own test cases, different from these.

Related reading:
• Concrete syntax for types and for unit tests, in Figure 7.1 on 476
• As above, the explanation and examples of check-type and check-principal-type in section 7.4.6, which starts on page 491.


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)
  For 23(b), please put the code in one of your READMEs. If you completed it with your partner, put it in the pair one; otherwise, put it in the solo one.
• 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 530 and exercise 2 on page 530.
  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
• 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 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. Then 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\(^5\) 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 503.
- 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:

(\{(\text{check-type-error} (\lambda (x) (\text{cons} x x)))\})
(\{(\text{check-type-error} (\lambda (x) (\text{cdr} (\text{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\(^6\). It contains some type tests as well as a check-expect.

\(^5\)\text{typesys.html#how-to-build-a-type-checker}
\(^6\)\text{../progs/tsort.nml}
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. Ouch!

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

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:

<table>
<thead>
<tr>
<th>Exemplary</th>
<th>Satisfactory</th>
<th>Must Improve</th>
</tr>
</thead>
<tbody>
<tr>
<td>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.</td>
<td>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.</td>
<td>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.</td>
</tr>
</tbody>
</table>
Code structure

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

<table>
<thead>
<tr>
<th>Exemplary</th>
<th>Satisfactory</th>
<th>Must Improve</th>
</tr>
</thead>
<tbody>
<tr>
<td>Structure</td>
<td></td>
<td></td>
</tr>
<tr>
<td>• The nine cases of simple type equality are handled by these five patterns: TYVAR/any, any/TYVAR, CONAPP/CONAPP, TYCON/TYCON, other.</td>
<td>• The nine cases are handled by nine patterns: one for each pair of value constructors for ty.</td>
<td>• The case analysis for a simple type equality does not have either of the two structures on the left.</td>
</tr>
<tr>
<td>• The code for solving $\alpha \sim \tau$ has exactly three cases.</td>
<td>• The code for $\alpha \sim \tau$ has more than three cases, but the nontrivial cases all look different.</td>
<td>• The code for $\alpha \sim \tau$ has more than three cases, and different nontrivial cases share duplicate or near-duplicate code.</td>
</tr>
<tr>
<td>• The constraint solver is implemented using an appropriate set of helper functions, each of which has a good name and a clear contract.</td>
<td>• The constraint solver is implemented using too many helper functions, but each one has a good name and a clear contract.</td>
<td>• Course staff cannot identify the role of helper functions; course staff can’t identify contracts and can’t infer contracts from names.</td>
</tr>
<tr>
<td>• Type inference for list literals has no redundant case analysis.</td>
<td>• Type inference for list literals has one redundant case analysis.</td>
<td>• Type inference for list literals has more than one redundant case analysis.</td>
</tr>
<tr>
<td>• Type inference for expressions has no redundant case analysis.</td>
<td>• Type inference for expressions has one redundant case analysis.</td>
<td>• Type inference for expressions has more than one redundant case analysis.</td>
</tr>
<tr>
<td>• In the code for type inference, course staff see how each part of the code is necessary to implement the algorithm correctly.</td>
<td>• 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.</td>
<td>• Course staff believe that the code is significantly more complex than what is required to implement the typing rules.</td>
</tr>
<tr>
<td>• Wherever possible appropriate, submission uses map, filter, foldr, and exists, either from List or from ListPair.</td>
<td>• Submission sometimes uses a fold where map, filter, or exists could be used.</td>
<td>• Submission includes one or more recursive functions that could have been written without recursion by using map, filter, List.exists, or a ListPair function.</td>
</tr>
</tbody>
</table>