Homework 3: Simply-Typed Lambda Calculus

Due date: October 27 (Thursday) at 11:59pm

Overview

This homework consists of two parts:

1. Implement a type inference/checking algorithm for the simply-typed lambda calculus.
2. Write a few functions in the simply-typed lambda calculus.

Your assignment will be graded on the `myth.stanford.edu` cluster, so while you can develop anywhere, be sure to test your code there. When running code on the myth machines, use the command `python3` so you test your code with the same version of Python we will use to grade your code.

Part 1: Type Inference for the Simply-Typed Lambda Calculus

Recall from lecture the definition of the simply-typed lambda calculus for terms e, constants c and types τ:

$$
\begin{aligned}
e &::= x \mid \lambda x{:}\,\tau.\, e \mid e\ e \mid c \\
c &::= \text{ifz} \mid + \mid - \mid 0 \mid 1 \mid 2 \ldots \\
\tau &::= \text{int} \mid \tau \to \tau \mid \alpha
\end{aligned}
$$

(See part 2 for a definition of the constants c, such as ifz.)

It is inconvenient to have to write down the types for every variable. For example, consider:

$$(\lambda x{:}\,\text{int}.\ +\ x\,1)\,2$$

Because the function is applied to the argument 2, we know the function parameter x must have type int. A type checker can verify that this program is well typed, but a type inference algorithm will *infer* the type of x, allowing us to write:

$$(\lambda x.\ +\ x\,1)\,2$$

Generating Constraints

Recall the typing rules for type inference, where A is the type context and α and β are type variables *fresh* in each rule:

$$\frac{}{A, x{:}\,\tau \vdash x : \tau}(\text{Var}) \quad \frac{c{:}\,\tau}{A \vdash c : \tau}(\text{Const}) \quad \frac{A, x{:}\,\alpha \vdash e : \tau}{A \vdash (\lambda x.\, e) : \alpha \to \tau}(\text{Abs}) \quad \frac{A \vdash e : \tau \quad A \vdash e' : \tau' \quad \tau = \tau' \to \beta}{A \vdash e\ e' : \beta}(\text{App})$$

Implement these rules by writing a function that is recursive on the structure of the term e, taking arguments for A and e and returning the type τ. A simple way to generate fresh type variables (for α, β) is to increment a global counter (and make names like `a0`, `a1`, etc.). As a side effect of your function, collect all the type constraints ($\tau = \tau' \to \beta$) produced by the App rule.

Saturating Constraints

Recall the rules for saturating a set of type constraints S, where α is a type variable and τ, τ_i are arbitrary types:

$$\frac{S, \tau = \alpha}{S, \tau = \alpha, \alpha = \tau}(\text{Refl}) \quad \frac{S, \alpha = \tau_1, \alpha = \tau_2}{S, \alpha = \tau_1, \alpha = \tau_2, \tau_1 = \tau_2}(\text{Tran}) \quad \frac{S, \tau_1 \to \tau_2 = \tau_3 \to \tau_4}{S, \tau_1 \to \tau_2 = \tau_3 \to \tau_4, \tau_1 = \tau_3, \tau_2 = \tau_4}(\text{Struct})$$

Each of these rules adds one or two new constraints to the set, but the new constraint may already be present. Be sure to only continue applying rules if a new, non-redundant constraint is discovered (see below).

Typechecking

With the saturated constraints, we can now perform typechecking. We first need to check $\tau \to \tau' = \text{int} \notin S$ and $\text{int} = \tau \to \tau' \notin S$, i.e. there is no equality between int and a function type. If there is, the constraints have no solution, and the program is ill-typed.

Next we need to perform the substitution to obtain the final type for each definition. We do this substitution in a way that obtains the same result as the back-substitution described in lecture without changing the set of constraints. We define a recursive function C for *canonicalizing* a type τ with respect to constraints S:

$$\begin{array}{ll}
C(S, \text{int}) = \text{int} & \\
C(S, \tau \to \tau') = C(S, \tau) \to C(S, \tau') & \\
C(S, \alpha) = C(S, \tau) & \text{if } (\alpha = \tau \in S \text{ or } \tau = \alpha \in S) \text{ and } \tau \text{ is not a type variable} \\
C(S, \alpha) = C(S, \beta) & \text{if } (\alpha = \beta \in S \text{ or } \beta = \alpha \in S) \text{ and } \beta < \alpha \\
C(S, \alpha) = \alpha & \text{otherwise}
\end{array}$$

Note that canonicalizing a type variable α results in a variable exactly when α is only equal to other variables, and we use $<$ to pick one representative for each such equivalence class of type variables. We can compare the names of the variables as strings to obtain a suitable order.

If there are infinite solutions to S, then a naive recursive implementation of C will enter an infinite loop. To detect this situation, we can track the set X of all the types we are currently in the process of canonicalizing. At the beginning of $C(S, \tau)$, we check if the argument τ is in X: if so, we have an infinite loop. If not, we add τ to X and continue canonicalization. Finally, after any recursive calls, we remove τ from the set. (This last removal step is important when canonicalizing e.g. $\alpha \to \alpha$.) This approach is a standard pattern for detecting loops in a depth-first traversal.

To check the whole program for finite solutions, it is sufficient to call your canonicalize function on the left hand side of each equation in S. If there are no equations between int and a function type and no loops, then the program is well-typed; otherwise, it is ill-typed. Further, if it is well-typed then the canonicalization algorithm will produce solutions to the type constraints. In particular, if we obtained $A \vdash e : \tau$ with the type constraints S, then the final type of e after solving S is $C(S, \tau)$.

Complete Algorithm

To finish describing the algorithm, we need to describe how to process a whole program. We represent a program as a sequence of definitions:

$$p ::= \text{def } x = e;\ p \mid \epsilon$$

where ϵ means the empty string. We can then define rules for typechecking a program:

$$\frac{A \vdash e : \tau \qquad A, x{:}\,\tau \vdash p}{A \vdash \text{def } x = e;\ p}(\text{Def}) \qquad \frac{}{A \vdash \epsilon}(\text{Empty})$$

These rules essentially say that e in each definition should be checked using the rules above, and that the resulting type should be available in later definitions. One detail to note is that we didn't put x into

the environment when checking e, so like the previous assignment, definitions can't be recursive. You can implement these rules with a simple loop that accumulates types into a dictionary representing the environment. You should wait until the end of this loop before saturating and solving the constraints.

Implementation

To implement the type inference described in this part, fill in the function `typecheck` in `typecheck.py`. This function takes a `Prog`, which is a list of definitions `def x = e;`, and returns a list of `Types`, one for each definition. If the program is not well-typed, the function should raise an exception (`raise TypecheckingError()`, optionally with some helpful error message). Some notes:

- The programs (`p: Prog`) in the simply-typed calculus consist of a list (`p.defns`) of definitions (`d: Defn`). Each definition has a variable `d.s` and an expression `d.e`. Recall the variable is bound only **after** the definition, not inside `d.e`.
- In addition to `Var`, `App`, and `Lam`, expressions can be an `IntConst(i)`, which are always of type int. Look at `src/lam.py` for the full definitions.
- You will need to typecheck constants, which have fixed types given below and in `CONSTS` (in `src/lam.py`). A `Var` is a constant if it is in the dictionary `CONSTS`, and a variable otherwise. (Given a `Var v`, you should look up `v.s` in `CONSTS` first, and then your environment.)
- We have supplied a representation of types, `Type`, with subclasses for int, functions, and type variables (`IntTp()`, `Func(a, b)`, `TpVar('a')`), in `src/lam.py`. These classes can be compared (`a == b`), and stored in sets. Saturating the type constraints will be easy if you represent them as a set of pairs of the left hand and right hand side of the equations (`set([(l,r)])`). Then you can apply the rules until the size of the set stops changing.
- Test your solution by running `python3 src/main.py tests/*.st`, which will run all the tests. Files that start with `y-` are expected to typecheck and files starting with `n-` are expected to fail. You will get warnings if your code does the opposite. You can run `main.py` on individual files as well.

Part 2: Programming in the Simply-Typed Lambda Calculus

The simply-typed calculus introduces some significant restrictions on the programs that we can write, but some programs are still expressible. You will write several functions in the simply-typed calculus, in `problem.st`:

1. **sum** for two Church-encoded numbers. Note that your solution must both typecheck and execute correctly. Not all equivalent ways to write sum will typecheck.
2. **value**, which takes a Church-encoded number and produces the value as a regular int.
3. **is_div**, which takes two ints $x, y > 0$ and returns 1 if x is divisible by y and 0 otherwise.

You can make use of the following constants in your implementation:

- `+` : int $\to$ int $\to$ int where $(\texttt{+}\, x)\, y = x + y$
- `-` : int $\to$ int $\to$ int where $(\texttt{-}\, x)\, y = x - y$
- `/` : int $\to$ int $\to$ int where $(\texttt{/}\, x)\, y = \lfloor x/y \rfloor$
- `*` : int $\to$ int $\to$ int where $(\texttt{*}\, x)\, y = x * y$
- `ifz` : int $\to$ int $\to$ int $\to$ int where $((\texttt{ifz}\, x)\, y)\, z) = y$ if $x = 0$ else z (i.e. the 'if zero' function)

Submission

- Generate a tarball file `solution.tar.gz` by running `python3 src/submit.py` and upload the tarball file to Canvas. Make sure the script gives you no errors or warnings.
- Your solution tarball should include `typecheck.py`, `problem.st`, and `README.txt`.
- Edit `README.txt` to include your student ID number (the 8-digit number).