Assignment 2: Types

Due: 10/2/25 at 11:59pm

2a: Type Theory (30 points)

For these theory problems, you will work with a formalized subset of the Rice source-level language. This language includes constants, tuples, and let-bindings, but excludes features like functions and structs. The language is defined in the figures at the end of this handout: syntax (Figure 1), static semantics (Figure 2), and dynamic semantics (Figure 3).

Submission

Submit your work as a PDF on Gradescope at this link: https://www.gradescope.com/courses/1107485/assignments/6778392

The LaTeX source for this handout will be provided to help you write your solutions.

Problem 1 (3 points)

A well-typed expression has a type derivation and an execution trace. For example, consider the expression "hello" ^ " world". Its typing derivation is:

$$\dfrac{\dfrac{}{\vdash \text{“hello”} : \mathsf{string}}\ \textsc{T-String} \qquad \dfrac{}{\vdash \text{“ world”} : \mathsf{string}}\ \textsc{T-String}}{\vdash \text{“hello” ^ “ world”} : \mathsf{string}}\ \textsc{T-Concat}$$

And its execution trace is:

$$\begin{aligned} &\text{“hello” ^ “ world”} \\ \hookrightarrow\ &\text{“hello world”} \end{aligned}$$

Provide the typing derivation and execution trace for the following expression:

$$\mathsf{let}\ x = 1 + 2\ \mathsf{in}\ \mathsf{let}\ y = \text{“hello world”}\ \mathsf{in}\ (x, y)$$

Problem 2 (5 points)

Imagine a hypothetical language Rice2 which is the same as Rice, except the E-Let does not have the premise e_1 val. Describe what language property Rice has that Rice2 lacks. Give an example of an expression which demonstrates how this property differs between the two languages.

Problem 3 (10 points)

The Rice language specification provided below has a critical bug. Describe the bug in a sentence. Provide an expression which demonstrates how this bug violates preservation. Provide a different expression which demonstrates how this bug also violates progress.

Problem 4 (12 points)

Say that we want to add if-expressions to our formalized subset of Rice. If-expressions have the following syntax:

$$\text{Expr } e ::= \ldots \mid \text{if } e_1 \text{ then } e_2 \text{ else } e_3$$

Provide the static and dynamic semantics for if-expressions that matches its semantics in the informal Rice language.

Prove that progress and preservation holds for your if-expression semantics.

2b: Type Inference (70 points)

Your task is to implement type inference for the Rice compiler. The parser already implements the ability to specify holes in types. For example:

```
fn stringify(t: _) -> _ {
  let tup: (_, _) = t in
  tup.0 ^ tup.1
}
```

The underscore indicates a type hole. This is implicitly converted into `TypeKind::Hole` with a numeric parameter assigned by a global counter. (Note the global `counter: &Cell<usize>` parameter to the LALRPOP grammar.) The AST for the above function will print as:

```
fn stringify(t: ?0) -> ?1 {
  let tup: (?2, ?3) = t in
  (tup.0 ^ tup.1)
}
```

The challenge for type inference is to fill these holes, i.e., determine a concrete type for every hole in the program, or raise an error if this cannot be done. For `stringify`, because the concatentation operator requires that its inputs and output are strings, then the inferred types should be:

```
fn stringify(t: (string, string)) -> string {
  let tup: (string, string) = t in
  (tup.0 ^ tup.1)
}
```

Submission

Submit your work as a ZIP file of your codebase on Gradescope at this link: https://www.gradescope.com/courses/1107485/assignments/6775804

You should use the `submit.sh` script from the starter code from Assignment 1. Your code will be graded like before: there are hidden tests which either should pass or fail, and your compiler should be consistent with the tests.

Specification

Type inference for Rice is *global*. For example, in this program both holes are inferred to be `int` because of the call in `main`.

```
fn id(x: _) -> _ { x }
fn main() { id(0); () }
```

The tricky case for Rice type inference is dealing with types of variable size: tuples and functions. Variable-sized types must have a constraint which exactly determines their size, or else type inference should fail. For example, this function on its own is not well-typed, because `x` could be a tuple of any size greater than zero:

```
fn foo(x: _) -> _ { x.0 }
```

However, if it called with an argument of known size, then this program is well-typed:

```
fn foo(x: _) -> _ { x.0 }
fn main() { foo((1, 2)); () }
```

You may make the following simplifying assumptions in your implementation.

- Types in casts will always be fully specified, i.e., not contain holes.
- You can ignore the fragment of the language dealing with structs and interfaces since we haven't had time to cover it. That is, assume your inference algorithm will only be tested on programs without structs or interfaces.

Implementation

I would design your solution to follow roughly these steps:

1. **Collect constraints**: in the compiler's initial pass through the program (i.e., the various check functions on `Tcx`), collect all the constraints on types required for inference.
2. **Solve constraints**: for constraints of the form $\tau_1 = \tau_2$, directly solve them with unification.
3. **Solve goals**: for goals of forms such as $\tau_1 = \tau_2.i$, attempt to solve them if τ_2 can be normalized to a concrete type.
4. **Check additional constraints**: for constraints of the form "τ_1 is castable to τ_2", check those for validity.
5. **Apply the solution**: walk through the program and replace every type with its maximally concrete form, erroring if any hole is left unfilled.

A few additional tips:

- The vast majority of your changes will be to the type checking code in `src/tir/typeck.rs`. You may need to propagate additional information from the parser to the typechecker.
- In experimenting with this assignment, I tried using an off-the-shelf union-find implementation (e.g. petgraph has one), but found it didn't really make life easier. I would recommend rolling your own, even if it's less efficient than a hyper-optimized algorithm.
- `Tcx::ty_equiv` and `Tcx::ty_constraint` have been helpfully factored out for your convenience.

Number $n \in \mathbb{Z}$ Index $i \in \mathbb{N}$ String s Bool $b ::= \mathsf{false} \mid \mathsf{true}$ Const $c ::= n \mid s \mid b$ Variable x

$$\begin{aligned}
\text{Expr } e &::= c \mid e_1 \oplus e_2 \mid (e_1, \ldots, e_n) \mid e.i \mid \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 \mid x \\
\text{Type } \tau &::= \mathsf{int} \mid \mathsf{string} \mid \mathsf{bool} \mid (\tau_1, \ldots, \tau_n) \\
\text{Binop } \oplus &::= + \mid \hat{} \mid =
\end{aligned}$$

Figure 1: Rice syntax.

$\boxed{\Gamma \vdash e : \tau}$

$$\frac{}{\vdash n : \mathsf{int}}\ \textsc{T-Number} \qquad \frac{}{\vdash s : \mathsf{string}}\ \textsc{T-String} \qquad \frac{}{\vdash b : \mathsf{bool}}\ \textsc{T-Bool}$$

$$\frac{\Gamma \vdash e_1 : \mathsf{int} \qquad \Gamma \vdash e_2 : \mathsf{int}}{\Gamma \vdash e_1 + e_2 : \mathsf{int}}\ \textsc{T-Add} \qquad \frac{\Gamma \vdash e_1 : \mathsf{string} \qquad \Gamma \vdash e_2 : \mathsf{string}}{\Gamma \vdash e_1 \hat{} e_2 : \mathsf{string}}\ \textsc{T-Concat}$$

$$\frac{\Gamma \vdash e_1 : \tau \qquad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1 = e_2 : \mathsf{bool}}\ \textsc{T-Eq} \qquad \frac{\Gamma \vdash e_1 : \tau_1 \qquad \cdots \qquad \Gamma \vdash e_n : \tau_n}{\Gamma \vdash (e_1, \ldots, e_n) : (\tau_1, \ldots, \tau_n)}\ \textsc{T-Tuple}$$

$$\frac{\Gamma \vdash e : (\tau_1, \ldots, \tau_n) \qquad i \leq n}{\Gamma \vdash e.i : \tau_i}\ \textsc{T-Project} \qquad \frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma, x : \tau_1 \vdash e_2 : \tau_2}{\Gamma \vdash \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 : \tau_2}\ \textsc{T-Let}$$

$$\frac{}{\Gamma \vdash x : \tau}\ \textsc{T-Var}$$

Figure 2: Rice static semantics.

$\boxed{e \text{ val}}$

$$\frac{}{c \text{ val}} \text{ V-Const} \qquad \frac{e_1 \text{ val} \quad \cdots \quad e_n \text{ val}}{(e_1, \ldots, e_n) \text{ val}} \text{ V-Tuple}$$

$\boxed{e \hookrightarrow e'}$

$$\frac{e_1 \hookrightarrow e_1'}{e_1 \oplus e_2 \hookrightarrow e_1' \oplus e_2} \text{ E-Binop-L} \qquad \frac{e_1 \text{ val} \quad e_2 \hookrightarrow e_2'}{e_1 \oplus e_2 \hookrightarrow e_1 \oplus e_2'} \text{ E-Binop-R}$$

$$\frac{e_1 \text{ val} \quad e_2 \text{ val} \quad e_3 = e_1 \oplus e_2}{e_1 \oplus e_2 \hookrightarrow e_3} \text{ E-Binop} \qquad \frac{e_1 \text{ val} \quad \cdots \quad e_{i-1} \text{ val} \quad e_i \hookrightarrow e_i'}{(e_1, \ldots, e_i, \ldots, e_n) \hookrightarrow (e_1, \ldots, e_i', \ldots, e_n)} \text{ E-Tuple}$$

$$\frac{e \hookrightarrow e'}{e.i \hookrightarrow e'.i} \text{ E-Proj-Step} \qquad \frac{i \leq n}{(e_1, \ldots, e_n).i \hookrightarrow e_i} \text{ E-Proj}$$

$$\frac{e_1 \hookrightarrow e_1'}{\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 \hookrightarrow \mathsf{let}\ x = e_1'\ \mathsf{in}\ e_2} \text{ E-Let-Step} \qquad \frac{e_1 \text{ val}}{\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 \hookrightarrow e_2[x \mapsto e_1]} \text{ E-Let}$$

Figure 3: Rice dynamic semantics.