

# CS515: Programming Languages and Compilers I

## Project 2: A Type Reconstructor for TINY

Due date: Tuesday, November 29

### Project Description

Implement a type reconstruction algorithm that computes a type expression for every TINY program. Your type reconstructor should report an error if no valid type expression can be found. You should use the built-in Scheme function `error` to report such errors.

Example: `(error 'unify "*** type conflict ***")` Your Scheme program should return the type expression computed for a given TINY program if no type error is found.

Our TINY language is statically typed. You may assume that all variable names are distinct, avoiding the possible need for renaming. Recursive types are not allowed, for example, `(lambda(x) (x x))` should result in a type error. This means that you will need to implement an “occurs check”.

### Syntax

The TINY language is defined as follows

---

$x \in \text{variables}$	
$n \in \text{integers}$	
$c ::= n \mid \#t \mid \#f \mid \text{add1} \mid \text{sub1} \mid \text{zero?} \mid \text{and} \mid \text{or} \mid \text{not}$	constants
$e ::= c \mid x \mid (\text{lambda } (x) e) \mid (e e)$	expressions

---

`add1` adds the value 1 to its argument. `sub1` subtracts the value 1 from its argument.

### Type System

Assume that  $E$  is finite map from variables to type expressions, i.e.,  $E: \text{variables} \rightarrow \text{type\_expressions}$ . Type expressions may contain type variables (e.g.:  $\alpha, \beta$ ). The notation  $E[x \mapsto \alpha]$  means that the type environment

$E$  has been extended by the mapping of variable  $x$  to type expression  $\alpha$ . A program in TINY must be closed (no free variables) and satisfy the following type system.

$$\begin{array}{c}
 E \vdash n : int \quad E \vdash \#t : bool \quad E \vdash \#f : bool \\
 E \vdash add1 : (int \rightarrow int) \quad E \vdash sub1 : (int \rightarrow int) \quad E \vdash zero? : (int \rightarrow bool) \\
 E \vdash and : (bool \rightarrow (bool \rightarrow bool)) \quad E \vdash or : (bool \rightarrow (bool \rightarrow bool)) \\
 E \vdash not : (bool \rightarrow bool) \\
 E \vdash x : A(x) \quad \text{if } A(x) \in \text{type\_expressions} \\
 \\
 \frac{E \vdash x : \alpha \quad E[x \mapsto \alpha] \vdash e : \beta}{E \vdash (\text{lambda}( x ) e) : \alpha \rightarrow \beta} \\
 \\
 \frac{E \vdash e_1 : (\alpha \rightarrow \beta) \quad E \vdash e_2 : \alpha}{E \vdash (e_1 e_2) : \beta}
 \end{array}$$

## Provided Code and Things to Do

The provided Scheme function *parse* maps a TINY program into an AST representation. Your type reconstructor should take as input an AST, a type environment, and a set of constraints:

```
(define TR
  (lambda (ast E C)
    ... ))
```

The Scheme function *newtvar* will produce a “fresh” (type variable) symbol each time it is called. You will need to write a *unify* function that unifies two type expressions and produces a potentially new set of constraints. The definitions given above implicitly define the type expression language that you should use. As you can see, all functions have only a single argument, i.e., *and* and *or* are curried. HINT: Although *add1*, *sub1*, *zero?*, *and*, *or* and *not* are constants, they can be treated as variables in an initial type environment:

```
(define init_E
  '((add1 (int -> int)) (sub1 (int -> int))
    (zero? (int -> bool)) (not (bool -> bool))
    (and (bool -> (bool -> bool))) (or (bool -> (bool -> bool)))))
```

Start early since the project is not 100% defined, i.e., there will be problems along the way that you need to identify and solve.

Good luck!