Type Inference System

The Type Inference System is a TypeScript Implementation of the algorithm for Type Checking and Inference using Type Equations.

System Architecture

The system is built from the following modules:

- Abstract Syntax for L5 Expressions and for Type Expressions
- The key data types manipulated in the algorithm, Type Equations and Type Substitutions.
- The logic of the algorithm is implemented in three functional packages: Exp-to-Pool, Exp-to-Equations and the top-level algorithm Solve.

The key interfaces of each package are the following:
Layer 1: Languages Definitions

The Type Inference algorithm maps expressions in one language (L5 expressions) into expressions in another language (Type Expressions). The lower level of the Type Inference system implements the Abstract Syntax interfaces for expressions in these two languages.

L5-AST:
Implements parser and abstract syntax for the following BNF concrete syntax:

BNF

```plaintext
// <program> ::= (L5 <exp>+)                  / Program(exp:List(exp))
// <exp> ::= <define> | <cexp>                / DefExp | CExp
// <define> ::= ( define <var-decl> <cexp> )  / DefExp(var:VarDecl, val:CExp)
// <var> ::= <identifier>                     / VarRef(var:string)
// <cexp> ::= <number>                        / NumExp(val:number)
//         | <boolean>                       / BoolExp(val:boolean)
//         | <string>                        / StrExp(val:string)
//         | <prim-op>                       / PrimOp(op:string)
//         | <var-ref>                      / ProcExp(params:VarDecl[], body:CExp[], returnTE: TExp))
//         | ( if <cexp> <cexp> <cexp> )     / IfExp(test: CExp, then: CExp, alt: CExp)
//         | ( quote <sexp> )                / LitExp(val:SExp)
//         | ( lambda ( <var-decl>* ) (: <TExp>)? <cexp>+ ) / ProcExp(params:VarDecl[], body:CExp[], returnTE: TExp))
//         | ( set! <var> <cexp> )           / SetExp(var: varRef, val: CExp)
// <binding> ::= ( <var-decl> <cexp> )         / Binding(var:VarDecl, val:CExp)
// <prim-op> ::= + | - | * | / | < | > | = | not | eq? | string?
//                  | cons | car | cdr | list? | number?
//                  | boolean? | symbol? | string?
//                  | display | newline
// <num-exp> ::= a number token
// <bool-exp> ::= #t | #f
// <var-ref> ::= an identifier token         / VarRef(var)
// <var-decl> ::= an identifier token | (var : TExp) / VarRef(var, TE: TExp)
// <sexp> ::= symbol | number | bool | string | ( <sexp>* )
```

Note that we introduce optional type annotations in the L5 syntax in 2 places only:

- As part of VarDecl in the form (var : type-expression)
- As part of ProcExp in the return type of the procedure in the form:
  (lambda ((x : number) (y : number)) : number …)

TExp-AST

```plaintext
// Type Language
// <TExp> ::= <AtomicTExp> | <CompoundTExp> | <TVar>
// <AtomicTExp> ::= <NumTExp> | <BoolTExp> | <StrTExp> | <VoidTExp>
// <NumTExp> ::= number // NumTExp()
// <BoolTExp> ::= boolean // BoolTExp()
// <StrTExp> ::= string // StrTExp()
// <VoidTExp> ::= void // VoidTExp()
// <CompoundTExp> ::= <ProcTExp> | <TupleTExp>
```
The definition of the TExp datatypes, the parser and the unparsers of TExp is provided in file TExp.ts.

Layer 2: Substitution and Equation ADTs

The Type Inference algorithm is defined in terms of 2 basic formal tools:

- **Type Substitutions** are finite mappings of Type Variables to Type Expressions.
- **Equations** are pairs that state that a left Type Expression is to be held equivalent to a right Type Expression. Equations are the tool we use to represent a typing statement in the algorithm.

Substitution ADT

The interface of the Substitution ADT includes:

- `makeSub(variables, type-expressions)`
- `subGet(sub, var): lookup a variable in a substitution`
- `extendSub(sub, var, te): create a new substitution by adding a single pair (var:te) to sub`
- `applySub(sub, te): replace type variables inside te according to sub`
- `combineSub(sub1, sub2): compute a new sub such that:
  applySub(sub, te) = applySub(sub2, applySub(sub1, te)) for all te.`

Equation ADT

The interface includes:

```typescript
export interface Equation { left: T.TExp, right: T.TExp; }
export const makeEquation = (l: T.TExp, r: T.TExp): Equation => ({left: l, right: r});
```
Layer 3: Exp-to-Pool, Exp-to-Equations, Solve

`expToPool(exp)` traverses the AST of a Scheme Expression and maps each sub-expression in the AST to distinct Type Variables.

`poolToEquations(pool)` applies type formulae to propagate constraints from the syntactic structure of the L5 Expression to constraints (equations) among the type variables of the pool. There are specific constraints derived for each type of syntactic construct according to the semantics of the L5 programming language.

`infer(L5expr)` applies the whole logic of the type inference algorithm. The main steps of the algorithm are reflected in this top-level fragment:

```javascript
export const inferType = (exp: A.Exp): T.TExp => {
  const pool = expToPool(exp);
  const equations = poolToEquations(pool);
  const sub = solveEquations(equations);
  const texp = inPool(pool, exp);
  if (T.isTVar(texp) && !isError(sub))
    return S.subGet(sub, texp);
  else
    return texp;
}
```

The key logic of the algorithm lies in the `solveEquations` algorithm — which turns a list of Type Equations into a single coherent Type Substitution which satisfies all the constraints. `solveEquations()` relies on the computation of unification between two type expressions — as shown in the following fragment:

```javascript
// Purpose: Solve the type equations and return the resulting substitution
// or error, if not solvable
// Example: solveEquations()
//         poolToEquations(
//         expToPool(
//           parse('(((lambda (x) (x 11)) (lambda (y) y)))')) => sub
export const solveEquations = (equations: Equation[]): S.Sub | Error =>
  solve(equations, S.makeEmptySub());
```

```javascript
// Purpose: Solve the equations, starting from a given substitution.
// Returns the resulting substitution, or error, if not solvable
const solve = (equations: Equation[], sub: S.Sub): S.Sub | Error => {
  const solveVarEq = (tvar: T.TVar, texp: T.TExp): S.Sub | Error => {
    const sub2 = S.extendSub(sub, tvar, texp);
    return isError(sub2) ? sub2 : solve(rest(equations), sub2);
  }
  const bothSidesAtomic = (eq: Equation): boolean =>
    T.isAtomicTExp(eq.left) && T.isAtomicTExp(eq.right);
  const handleBothSidesAtomic = (eq: Equation): S.Sub | Error =>
    (T.isAtomicTExp(eq.left) && T.isAtomicTExp(eq.right) &&
    T.eqAtomicTExp(eq.left, eq.right)) ?
    solve(rest(equations), sub) :
    Error(`Equation with non-equal atomic type ${eq}`);
```
if (A.isEmpty(equations)) return sub;
const eq = makeEquation(S.applySub(sub, first(equations).left),
    S.applySub(sub, first(equations).right));
return T.isTVar(eq.left) ? solveVarEq(eq.left, eq.right) :
    T.isTVar(eq.right) ? solveVarEq(eq.right, eq.left) :
    bothSidesAtomic(eq) ? handleBothSidesAtomic(eq) :
        (T.isCompoundTExp(eq.left) && T.isCompoundTExp(eq.right) && canUnify(eq)) ?
            solve(R.concat(rest(equations), splitEquation(eq)), sub) :
                Error('Equation contains incompatible types ${eq}');

Compound type expressions are unified by unifying their components one by one. This is implemented by these 2 functions:

// Signature: canUnify(equation)
// Purpose: Compare the structure of the type expressions of the equation
const canUnify = (eq: Equation): boolean =>
    T.isProcTExp(eq.left) && T.isProcTExp(eq.right) &&
    (eq.left.paramTEs.length === eq.right.paramTEs.length);

// Signature: splitEquation(equation)
// Purpose: For an equation with unifyable type expressions, create equations for corresponding components.
// Type: [Equation -> List(Equation)]
// Example: splitEquation(
//     makeEquation(parseTExp('(T1 -> T2)'),
//     parseTExp('(T3 -> (T4 -> T4))) =>
//     [ {left:T2, right: (T4 -> T4)},
//     {left:T3, right: T1} ]
// @Pre: isCompoundExp(eq.left) && isCompoundExp(eq.right) && canUnify(eq)
const splitEquation = (eq: Equation) =>
    (T.isProcTExp(eq.left) && T.isProcTExp(eq.right)) ?
        R.zipWith(makeEquation,
            R.prepend(eq.left.returnTE, eq.left.paramTEs),
            R.prepend(eq.right.returnTE, eq.right.paramTEs)) :
        [];

Exercise: Extend the Type Inference system to support "If" expressions

The part of the Type Inference System which "knows" L5 Expressions is the function `makeEquationFromExp(exp, pool)`. The structure of this function is a case for each type of L5 expression. The code we provide supports:

- Procedure
- Application
- Number and Boolean
- Primitive Procedures

Extend the code to support Scheme expressions of the type (if <test> <then> <alt>).

Remember question 2 from last week's practical session.

From Typing Rule for if-expressions the type equations derived from a composite if-expression: (if _p _c _a) we derive 3 equations:

\[ T_c = T_a \]
\[ T_p = \text{boolean} \]
\[ T_{if} = T_c \]

Accordingly, the code of the function `makeEquationFromExp` is extended to process IfExp

```java
// Support if-expressions:
// [if <test> <then> <else>]
// [{T<test> = boolean}, {T<then> = T<alt>}, {T<then> = T<if>}]
A.isIfExp(exp) ? [makeEquation(inPool(pool, exp.test), T.makeBoolTExp()),
                  makeEquation(inPool(pool, exp.then), inPool(pool, exp.alt)),
                  makeEquation(inPool(pool, exp), inPool(exp.alt))]
```

Review the code of the AppExp equation factory:

```java
// An application must respect the type of its operator
// Type(Operator) = [T1 * .. * Tn -> Te]
// Type(Application) = Te
A.isAppExp(exp) ? [makeEquation(inPool(pool, exp.rator),
                    T.makeProcTExp(R.map((e) => inPool(pool, e), exp.rands),
                    inPool(pool, exp))]
```

This should create one equation for each argument and one equation for the exp itself.

Explain why it is not necessary to add an equation for the type of the exp itself.
// Implementation of the Substitution ADT
// ==============================================================
// A substitution is represented as a 2 element list of equal length
// lists of variables and type expression.
// The empty substitution is [[] , []]

export interface Sub { tag: "Sub" ; vars: TVar[] ; tes: TExp[] ; } ;

export const isSub = ( x: any ): x is Sub => x.tag === "Sub" ;

// Constructors:
// Signature: makeSub(vars, tes)
// Purpose: Create a substitution in which the i-th element of 'variables'
// is mapped to the i-th element of 'tes'.
// Example: makeSub(
// map(parseTE, ['x', 'y', 'z']),
// map(parseTE, ['number', "boolean", "(number -> number)"],
// => {tag: 'Sub', vars: [x y z],
// tes: [numTexp, boolTexp, ProcTexp([NumTexp, NumTexp])]
// makeSub(map(parseTE, ['x', "y", "z"]),
// map(parseTE, ['number', "boolean", "(z -> number)"],
// => error makeSub: circular substitution
// Pre-condition: (Length variables) = (Length tes)
// the list “variables” has no repetitions (it is a set)

export const makeSub = ( vars: TVar[] , tes: TExp[] ): Sub | Error => {
    const noOccurrences = zipWith(checkNoOccurrence, vars, tes);
    if (hasNoError(noOccurrences))
        return {tag: "Sub", vars: vars, tes: tes};
    else
        return Error(getErrorMessageMessages(noOccurrences));
};

export const makeEmptySub = (): Sub => ({tag: "Sub", vars: [], tes: []});

Whenever creating a substitution, we verify that the invariant holds that when a TVar is associated to a TExp, it does not occur inside the TExp. This is enforced by this function:

// Purpose: when attempting to bind tvar to te in a sub - check whether tvar occurs in te.
// Return error if a circular reference is found.
export const checkNoOccurrence = ( tvar: TVar, te: TExp ): true | Error => {
    const check = ( e: TExp ): true | Error =>
        isTVar(e) ? ((e.var === tvar.var) ?
            Error(‘Occur check error - circular sub ${tvar.var} in ${unparseTExp(te)}’): true):
        isAtomicTExp(e) ? true :
        isProcTExp(e) ? (hasNoError(map(check, e.paramTEs)) ?
            check(e.returnTE) :
            Error(getErrorMessageMessages(map(check, e.paramTEs)))) :
            Error(‘Bad type expression ${e} in ${te}’);
    return check(te);
};
Note on how we compare two substitutions to perform tests

In order to compare two substitutions, we “normalize” them by sorting their string representation and then compare the strings. This in effect achieves set equality:

```javascript
export const subToStr = (sub: S.Sub): string => {
  const vars = sub.vars.map(varName => `\${varName}:\${unparseTEExp(varName)}`);
  const te = sub.te.map(te => `\${unparseTEExp(te)}`);
  const sortedVars = vars.sort();
  const sortedTE = te.sort();
  return sortedVars.join(",") + "\n" + sortedTE.join(",");
}
```

// Compare 2 subs encoded as VarTe (set equality)
export const eqSub = (sub1: S.Sub, sub2: S.Sub): boolean =>
  deepEqual(subToStr(sub1), subToStr(sub2));

Understand how to use applySub

From L5-substitutions-ADT-tests.ts:

```javascript
// applySub
{
  const sub1 = sub(["T1", "T2"], ["number", "boolean"]);
  const te1 = parseTE("(T1 * T2 -> T1)"),
  if (! isInvalid(te1)) {
    const te2 = S.applySub(sub1, te1);
    assert(unparseTEExp(te2) === "(number * boolean -> number)");
  }
}
```

Understand how to use combineSub

```javascript
// {T1:(number -> S1), T2:(number -> S4)} o {T3:(number -> S2)} =>
// {T1:(number -> S1), T2:(number -> S4), T3:(number -> S2)}
{
  const sub1 = sub(["T1", "T2"],
    ["(number -> S1)", "(number -> S4)"]);
  const sub2 = sub(["T3"], ["(number -> S2)"]);
  const expected = sub(["T1", "T2", "T3"],
    ["(number -> S1)", "(number -> S4)", "(number -> S2)"]);
  const res = S.combineSub(sub1, sub2);
  assertEqSub(res, expected);
}