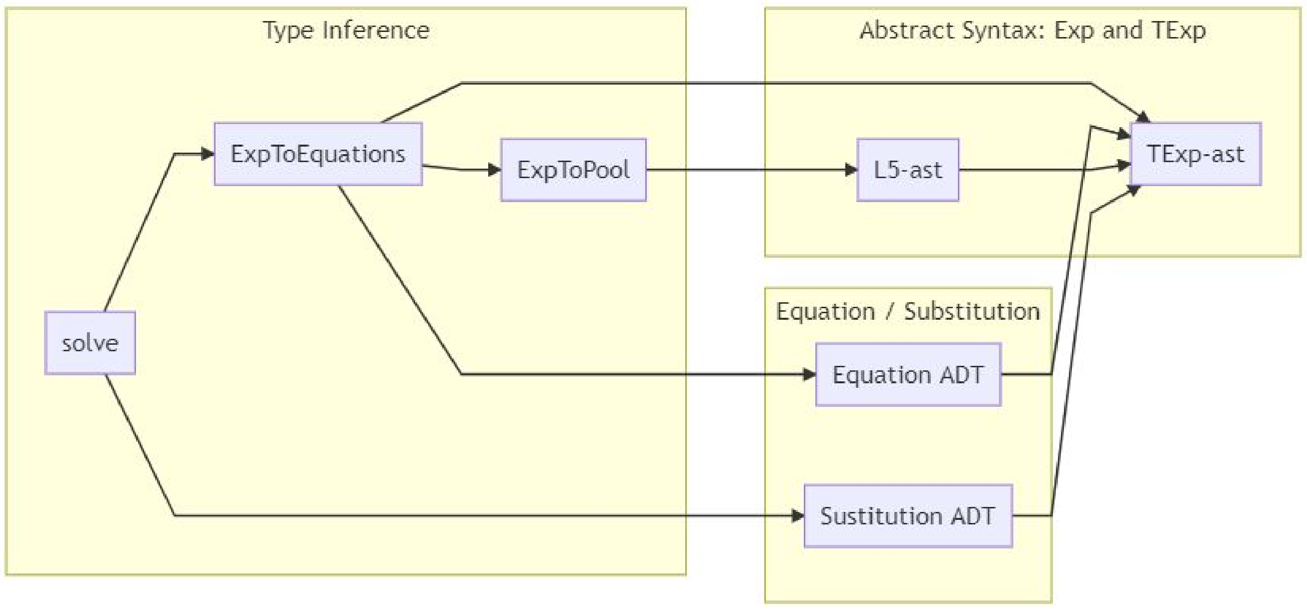
עקרונות שפות תכנות
תרגול 8 Type Inference System

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:



The layers of the system are:

- 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.

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

```
;;
// <program> ::= (L5 <exp>+)                    / Program(exps: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)
//         |  <var-ref>
//         |  ( lambda ( <var-decl>* ) <TExp>? <cexp>+ ) / ProcExp(params:VarDecl[],
body:CExp[], returnTE: TExp))
//         |  ( if <cexp> <cexp> <cexp> )     / IfExp(test: CExp, then: CExp, alt: CExp)
//         |  ( quote <sexp> )                  / LitExp(val:SExp)
//         |  ( <cexp> <cexp>* )                / AppExp(operator:CExp, operands:CExp[]))
//         |  ( let ( <binding>* ) <cexp>+ )  / LetExp(bindings:Binding[], body:CExp[]))
//         |  ( letrec ( binding*) <cexp>+ )  / LetrecExp(bindings:Bindings[], body: CExp)
//         |  ( set! <var> <cexp>)             / SetExp(var: varRef, val: CExp)
// <binding>  ::= ( <var> <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) ##### L5
// <sexp>     ::= symbol | number | bool | string | ( <sexp>* )              ##### L3
*/
```

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

```
/*
;; TExp AST
;; ========
;; Type checking language
;; Syntax with optional type annotations for var declarations and function return types.

;; Type language
;; <texp>         ::= <atomic-te> | <compound-te> | <tvar>
;; <atomic-te>    ::= <num-te> | <bool-te> | <void-te>
;; <num-te>       ::= number   // num-te()
;; <bool-te>      ::= boolean  // bool-te()
;; <str-te>       ::= string   // str-te()
;; <void-te>      ::= void     // void-te()
;; <compound-te>  ::= <proc-te>
;; <non-tuple-te> ::= <atomic-te> | <proc-te> | <tvar>
;; <proc-te>      ::= [ <tuple-te> -> <non-tuple-te> ] // proc-te(param-tes: list(te),
return-te: te)
;; <tuple-te>     ::= <non-empty-tuple-te> | <empty-te>
;; <non-empty-tuple-te> ::= ( <non-tuple-te> *)* <non-tuple-te> // tuple-te(tes: list(te))
;; <empty-te>     ::= Empty
;; <tvar>         ::= a symbol starting with T // tvar(id: Symbol, contents;
Box(string|boolean))
```

The definition of the TExp data types, the parser of TExp is provided in file **TExp.ts**

Examples of Types Expressions:

- number
- boolean
- void
- (number -> boolean)
- (number * number -> boolean)
- (number -> (number -> boolean))
- (empty -> number) // (empty -> void)

```
// Purpose: Compute the type of an expression
// Traverse the AST and check the type according to the exp type.
export const typeofExp = (exp: A.Parsed, tenv: E.TEnv): Result<T.TExp> =>
    A.isNumExp(exp) ? makeOk(T.makeNumTExp()) :
    A.isBoolExp(exp) ? makeOk(T.makeBoolTExp()) :
    A.isStrExp(exp) ? makeOk(T.makeStrTExp()) :
    A.isPrimOp(exp) ? TC.typeofPrim(exp) :
    A.isVarRef(exp) ? E.applyTEnv(tenv, exp.var) :
    A.isIfExp(exp) ? typeofIf(exp, tenv) :
    A.isProcExp(exp) ? typeofProc(exp, tenv) :
    A.isAppExp(exp) ? typeofApp(exp, tenv) :
    A.isLetExp(exp) ? typeofLet(exp, tenv) :
    A.isLetrecExp(exp) ? typeofLetrec(exp, tenv) :
    A.isDefineExp(exp) ? typeofDefine(exp, tenv) :
    A.isProgram(exp) ? typeofProgram(exp, tenv) :
    // Skip isSetExp(exp) isLitExp(exp)
    makeFailure("Unknown type");
```

- Atomic types
- Compound types

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

```
export interface Sub {tag: "Sub"; vars: TVar[]; tes: TExp[]; }
export const isSub = (x: any): x is Sub => x.tag === "Sub";
```

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

```
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.

What is Pool ?

```
export interface PoolItem {e: A.Exp, te: T.TExp}
export type Pool = PoolItem[];
```

```
export const expToPool = (exp: A.Exp): Pool => {
    const findVars = (e: A.Exp, pool: Pool): Pool =>
        A.isAtomicExp(e) ? extendPool(e, pool) :
        A.isProcExp(e) ? extendPool(e, reducePool(findVars, e.body,
mapPoolVarDecls(extendPoolVarDecl, e.args, pool))) :
        A.isCompoundExp(e) ? extendPool(e, reducePool(findVars, A.expComponents(e),
pool)) :
        makeEmptyPool();
    return findVars(exp, makeEmptyPool());
};
```

```
// Purpose: Traverse the abstract syntax tree L5-exp
//          and collect all sub-expressions into a Pool of fresh type variables.
// Example:
// bind(bind(p('(+ x 1)'), parseL5Exp), e => makeOk(TE.expToPool(e))) =>
// Ok([[AppExp(PrimOp(+), [VarRef(x), NumExp(1)]), TVar(16)],
//     [NumExp(1), TVar(15)],
//     [VarRef(x), TVar(14)],
//     [PrimOp(+), TVar(13)]])
```

- **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.

```
// Purpose: Return a set of equations for a given Exp encoded as a pool
// @Pre: pool is the result of expToPool(exp)
export const poolToEquations = (pool: Pool): Opt.Optional<Equation[]> => {
    // VarRef generate no equations beyond that of var-decl - remove them.
    const poolWithoutVars: Pool = R.filter(R.propSatisfies(R.complement(A.isVarRef), 'e'),
pool);
    return Opt.bind(Opt.mapOptional((e: A.Exp) => makeEquationsFromExp(e, pool),
R.map(R.prop('e'), poolWithoutVars)),
                    (eqns: Equation[][]) => Opt.makeSome(R.chain(R.identity, eqns)));
};
```

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

```
// Signature: inferType(exp)
// Purpose: Infer the type of an expression using the equations method
// Example: unparseTExp(inferType(parse('(lambda (f x) (f (f x)))')))
//          ==> '((T_1 -> T_1) * T_1 -> T_1)'
export const inferType = (exp: A.Exp): Opt.Optional<T.TExp> => {
    const pool = expToPool(exp);
    const equations = poolToEquations(pool);
    const sub = Opt.bind(equations, (eqns: Equation[]) =>
                                    Res.resultToOptional(solveEquations(eqns)))
    const texp = inPool(pool, exp);
    return Opt.safe2((sub: S.Sub, texp: T.TExp) =>
                    Opt.makeSome(T.isTVar(texp) ? S.subGet(sub, texp) : texp))
                                              (sub, 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:

```
export const solveEquations = (equations: Equation[]): Res.Result<S.Sub> =>
    solve(equations, S.makeEmptySub());

// Signature: solve(equations, substitution)
// 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): Res.Result<S.Sub> => {
    const solveVarEq = (tvar: T.TVar, texp: T.TExp): Res.Result<S.Sub> =>
       Res.bind(S.extendSub(sub, tvar, texp), sub2 => solve(rest(equations), sub2));

    const bothSidesAtomic = (eq: Equation): boolean =>
        T.isAtomicTExp(eq.left) && T.isAtomicTExp(eq.right);

    const handleBothSidesAtomic = (eq: Equation): Res.Result<S.Sub> =>
        T.isAtomicTExp(eq.left) && T.isAtomicTExp(eq.right) && T.eqAtomicTExp(eq.left,
eq.right)
        ? solve(rest(equations), sub)
        : Res.makeFailure(`Equation with non-equal atomic type ${eq}`);

    if (isEmpty(equations)) {
        return Res.makeOk(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) :
           Res.makeFailure(`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): Equation[] =>
    (T.isProcTExp(eq.left) && T.isProcTExp(eq.right)) ?
        R.zipWith(makeEquation,
                  cons(eq.left.returnTE, eq.left.paramTEs),
                  cons(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 **makeEquationsFromExp(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).

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

- Tp = boolean
- Tc = Ta
- Tif = Ta

```
// Purpose: compute the type of an if-exp
// Typing rule:
//   if type<test>(tenv) = boolean
//      type<then>(tenv) = t1
//      type<else>(tenv) = t1
// then type<(if test then else)>(tenv) = t1
export const typeofIf = (ifExp: IfExp, tenv: TEnv): Result<TExp> => {
    const testTE = typeofExp(ifExp.test, tenv);
    const thenTE = typeofExp(ifExp.then, tenv);
    const altTE = typeofExp(ifExp.alt, tenv);
    const constraint1 = bind(testTE, testTE => checkEqualType(testTE,
                                                  makeBoolTExp(), ifExp));
    const constraint2 = safe2((thenTE: TExp, altTE: TExp) =>
                            checkEqualType(thenTE, altTE, ifExp))(thenTE, altTE);
    return safe2((_c1: true, _c2: true) => thenTE)(constraint1, constraint2); }
```

נספח לעיון ADT – עבור. substitution

```
/// 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], [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)
//                variables has no repetitions (set)

export const makeSub = (vars: TVar[], tes: TExp[]): Result<Sub> =>
    bind(zipWithResult(checkNoOccurrence, vars, tes), _ =>
                                  makeOk({ tag: "Sub", vars: vars, tes: tes }));

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): Result<true> => {
    const check = (e: TExp): Result<true> =>
        isTVar(e) ? ((e.var === tvar.var) // T1 = T1->T2
           makeFailure(`Occur check error - circular sub ${tvar.var} in
                                          ${unparseTExp(te)}`): makeOk(true) :

        isAtomicTExp(e) ? makeOk(true) :
        isProcTExp(e) ? bind(mapResult(check, e.paramTEs), _ => check(e.returnTE)) :
        makeFailure(`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:

```
export const subToStr = (sub: Sub): Result<string> =>
    bindResult(zipWithResult((v, t) => bindResult(unparseTExp(t),
              up =>makeOk(`${v.var}:${up}`)), sub.vars, sub.tes),
               (vts: string[]) => makeOk(vts.sort().join(", ")));
```

Example of applySub From L5-substitutions-ADT-tests.ts:

```
// ============================================================
// Purpose: apply a sub to a TExp
"
export const applySub = (sub: Sub, te: TExp): TExp =>
    isEmptySub(sub) ? te :
    isAtomicTExp(te) ? te :
    isTVar(te) ? subGet(sub, te) :
    isProcTExp(te) ? makeProcTExp(map(curry(applySub)(sub), te.paramTEs),
applySub(sub, te.returnTE)) :
    te;
```

```
 describe('applySub', () => {
        it('applies a substitution on a type expression', () => {
            const sub1 = sub(["T1", "T2"], ["number", "boolean"]);
            const te1 = parseTE("(T1 * T2 -> T1)");
            const unparsed = safe2((sub: S.Sub, te: TExp) =>
                                unparseTExp(S.applySub(sub, te)))(sub1, te1);
            expect(unparsed).to.deep.equal(makeOk("(number * boolean ->
number)"));
        });
    });
```

Example of applySub From L5-substitutions-ADT-tests.ts:

```
    describe('combineSub', () => {
        it('combines substitutions (the o operator)', () => {
            // {T1:(number -> S1), T2:(number -> S4)} o {T3:(number -> S2)} =>
            // {T1:(number -> S1), T2:(number -> S4), T3:(number -> S2)}
            const sub11 = sub(["T1", "T2"], ["(number -> S1)", "(number -> S4)"]);
            const sub12 = sub(["T3"], ["(number -> S2)"]);
            const expected1 = bind(sub(["T1", "T2", "T3"], ["(number -> S1)",
"(number -> S4)", "(number -> S2)"]), subToStr);
            const res1 = bind(safe2(S.combineSub)(sub11, sub12), subToStr);
            expect(res1).to.deep.equal(expected1);
     });
});
```