

Principles of Programming Languages

Type Inference System

In the previous lecture, we have introduced the Type Equations algorithm to perform Type Inference over Scheme ($L5$) expressions.

In this lecture, we present two distinct implementations of this algorithm - one which is a “literal” application of the algorithm, and one which is an optimized transformation of the algorithm relying on more compact data structures and less traversals of the program.

The Type Inference with Type Equations system builds on the *L5* AST abstract syntax defined for the Type Checker. It introduces two new modules:

- Substitution ADT
- Type Equation Algorithm

The Substitution ADT is a direct implementation of the mathematical substitution object introduced in the previous lecture:

Definition: Type Substitution

A type-substitution s is a mapping from a finite set of type variables to a finite set of type expressions, such that $s(T)$ does not include T .

As usual, when defining substitutions and partial functions (we have already implemented such data structures when talking about environments), we adopt an inductive implementation to define the Type Substitution data type, which is the union of two disjoint types:

- The empty substitution
- Non-empty substitutions:

`sub(tvars: TVar[], texps: TExp[])`

The functional interface of the Substitution data type includes:

- Value constructors for the empty substitution `makeEmptySub` and non-empty substitutions `makeSub(tvars, texp)`.
- Value constructor for composing two substitutions `combineSub(sub1, sub2)`.

Both the functions `makeSub(tvars, texps)` and `combineSub(sub1, sub2)` return a non-empty substitution. They also enforce the **key invariant of substitutions**: for any variable T , `sub(T)` does not include T .

This invariant check is performed by the
function

`checkNoOccurrence(tvar, texp)`.

This function is a standard AST traversal of the
type expression **texp** looking for any instance
of the type variable **tvar** at any level.

This process is called **occurrence check** and is central to all unification based methods. It is a computationally expensive component of the algorithm.

The Substitution ADT

```
// 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.  
const checkNoOccurrence = (tvar: TVar, te: TExp): Result<true> => {  
  const check = (e: TExp): Result<true> =>  
    isTVar(e) ? ((e.var === tvar.var) ? makeFailure(...) : 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);  
};
```

Similarly to environments, we define the non-empty substitution as a linked-list of bindings mapping variables to values. This value type is computed using the `extendSub(sub, tvar, texp)` method.

Note that the method requires two “complications” in addition to extending the base-substitution with the new binding:

1. The existing right-hand-side of the base substitutions are updated with the new substitution
2. We perform an occurrence check on the resulting substitution (by invoking **makeSub** which includes an occurs-check).

The Substitution ADT

```
// Purpose: extend a substitution with one pair (v, te)  
// Calls to makeSub to do the occur-check  
const extendSub = (sub: Sub, v: TVar, te: TExp): Result<Sub> =>  
  bind(makeSub([v], [te]), (sub2: Sub) => {  
    const updatedTEs = map(partial(applySub, [sub2]), sub.tes);  
    return map(prop('var'), sub.vars).includes(v.var)  
      ? makeSub(sub.vars, updatedTEs)  
      : makeSub(cons(v, sub.vars), cons(te, updatedTEs));  
  });
```

In addition, we explicitly “compile” extended substitutions which are the result of composing two substitutions into a flat representation, using the value constructor **combineSub**.

Recall that substitution composition $s \circ s'$ is defined as:

- s' is applied to the type-expressions of s , *i.e.*, for every variable T' for which $s'(T')$ is defined, occurrences of T' in type expressions in s are replaced by $s'(T')$.

- A variable T in s' , for which $s(T)$ is defined, is removed from the domain of s' , *i.e.*, $s'(T)$ is not defined on it anymore.
- The modified s' is added to s .

- Identity bindings, *i.e.*, $s(T) = T$, are removed.
- If for some variable, $(s \circ s')(T)$ includes T , the combination fails.

This method is implemented literally in the `combineSub(sub1, sub2)` procedure - which has a structure similar to a reduce of the `extendSub` operation over all the pairs in `sub2`:

The Substitution ADT

```
// Purpose: Returns the composition of substitutions s.t.:  
// applySub(result, te) == applySub(sub2, applySub(sub1, te))  
const combineSub = (sub1: Sub, sub2: Sub): Result<Sub> =>  
  isEmptySub(sub1) ? makeOk(sub2) :  
  isEmptySub(sub2) ? makeOk(sub1) :  
  combine(sub1, sub2.vars, sub2.tes);  
  
const combine = (sub: Sub, vars: TVar[], tes: TExp[]): Result<Sub> =>  
  isEmpty(vars) ? makeOk(sub) :  
  bind(extendSub(sub, first(vars), first(tes)),  
    (extSub: Sub) => combine(extSub, rest(vars), rest(tes)));
```

The type equations module follows the definition of the algorithm:

- Rename bound variables in the expression e .
- Assign type variables to all sub-expressions.
- Construct type equations.
- Solve the equations.

Renaming of all bound variables in e is performed in the same way as we introduced when discussing the Substitution Model for Operational Semantics. We do not repeat this code here, and instead make the assumption that in the following, all bound variables have distinct names (the same variable name is never used twice in different scopes).

We implement the assignment of type variables to all sub-expressions by defining a data structure which we call the **pool** which contains a list of pairs (**exp**, **TVar**) for every node in the expression AST.

The pool is a list of pairs which exhaustively includes all the sub-expressions in the AST. It is built using the function **expToPool**. As usual, this function is an AST traversal method. Whenever a node in the AST is visited, we allocate a fresh Type Variable for it.

Pay attention to the way variable declarations and variable references are processed when constructing the pool:

The method `extendPool(exp, pool)` generates a fresh type variable (one that was never used before) and adds the mapping from `exp` to that new type variable to the pool.

When we enter a new scope in the expression (during its traversal), we need to keep track of the variable declarations - and map the variable name to the type of the variable declaration.

Recall that when we parse an *L5* expression, we consider type annotations optional. If they are provided, the **VarDecl** node stores the declared type expression in the **VarDecl.texp** field. If they are not provided by the programmer, the parser generates a new fresh variable and associates it to **VarDecl.texp**.

When we continue the traversal of the AST, and we later meet a **VarRef** node which refers to this **VarDecl** node, we want to associate it to the existing **VarDecl** type declaration (whether it is provided by the programmer or allocated by the parser).

To achieve this mechanism, when we meet a `VarDecl` node, we use the procedure `extendPoolVarDecl(varDecl, pool)` which adds the pair `(VarRef(varDecl.var), varDecl.texp)` to the pool. When we later reach a `VarRef` in the scope of this `VarDecl`, we find that the pair `(VarRef tvar)` already exists in the pool, and we do not allocate any new fresh type variable for it.

This mechanism crucially depends on the fact that `expToPool` traverses the expression AST top-down (traverse the `VarDecl` before the corresponding `VarRef` nodes are met) and the expression has been renamed before so that all `VarRefs` with a given name refer to the single `VarDecl` with the same name.

`expToPool` uses the function `reducePool` to accumulate the pairs `(AST-node, TVar)` into the pool without repetitions. This function is a variant of the reduce family of higher-order functions.

We use in these functions the `inPool` function which checks whether an expression is already present in the pool. In case the expression is found, we return the associated `TExp`, else we need to return a value that indicates the expression was not found. This is a typical situation where we have a search operator which can fail.

`Optional<T>` and `maybe`

We adopt the standard `Optional` generic type to represent this return type in a type-safe manner. In the same way as `Result` represents a call which may fail, `Optional` represents a call which may either return a value or the legitimate case of a missing value (which should not be considered an error). The two options are wrapped as `Some<T>` and `None` (parallel to `Ok<T>` and `Failure` for `Result<T>`).

Optional<T> and maybe

To manipulate **Optional** values, we use the **maybe** operator which passes the **Optional<T>** two possible continuations: a method receiving a **T** value in case the value was found, and one receiving no argument in case none was found. **maybe** allows type-safe composition of functions returning **Optional<T>** values in the same way as **bind** allows type-safe composition of **Result<T>** values.

We also implement **bind** and **safe2** versions for **Optional<T>** values with the same behavior as that we adopted for **Result<T>**.

`expToPool` code can be found [here](#).

The post-condition met at the end of the pool construction is that every node in the AST is mapped to a type variable - while preserving scoping relations (different occurrences of the same **VarRef** are all merged as a single pair mapping the **VarRef** to its declared type - which can be a non-instantiated type variable).

The next step of the algorithm consists of transforming the pool of $\{e: \text{exp}, te: \text{TVar}\}$ pairs into a set of equations. This part of the algorithm is the one that encapsulates the semantics of the type system used in the programming language.

The procedure `poolToEquations` performs this mapping. This procedure accumulates the transformation of all pairs `{e: exp, te: TVar}` in the pool to equations. The equation ADT is a pair left-hand-side / right-hand-side of type expressions (that is, each equation is encoded as a pair `{left: TExp, right: TExp}`).

The heart of the typing algorithm is the operation `makeEquationsFromExp` which covers the typing rules of the programming language. This implements the typing rules for procedure expressions, application expressions and atomic expressions.

For example, given a pair (**app-exp**, **TVar**), the procedure derives a type equation which mandates that the type variable associated to the operator of the application must be made equal to the type expression $(T_1 * \dots * T_n \rightarrow TVar)$ where T_i is the type variable associated to the i -th argument of the application expression.

For the base cases of primitive operators, we reuse the procedure **typeofPrim** which we defined in the type checker.

See the code for `makeEquationsFromExp`
[here](#).

Solving the Type Equations System

The last stage of the algorithm consists of solving the set of equations collected by the equations generator. The procedure `solve(equations, substitution)` is a direct implementation of the `solve` algorithm presented in the last lecture. It computes the unifier of all the equations - that is, it computes a type substitution which when it is applied to both sides of all the equations, makes the two sides identical.

This unifier substitution is computed incrementally, by processing each equation in turn.

Reminder of the algorithm from last lecture:

Solving the Type Equations System

Input: A set of type equations.

Output: A type substitution or FAIL.

Initialization:

1. substitution := { }
2. Order the set of input equations in any sequential order.
3. equation := te1 = te2, the first equation.

Solving the Type Equations System

Loop:

1. Apply the current substitution to the equation:

Let $te1s := te1 \circ substitution$

$te2s := te2 \circ substitution$

$equation := te1s = te2s$

2. If $te1s$ and $te2s$ are atomic types:

If $te1s \neq te2s$:

sub = FAIL

otherwise:

Do nothing.

Solving the Type Equations System

3. Without loss of generality:

If $te1s = T$, i.e., a type variable,
and $te1s \neq te2s$:

$substitution := substitution \circ \{T = te2s\}$

That is, apply the equation to substitution,
and add the equation to the substitution.
If the application fails (circular mapping),
 $substitution := FAIL$.

Solving the Type Equations System

4. If `te1s` and `te2s` are composite types:
 - If they have the same type constructor:
 - Split `te1s` and `te2s` into component type expressions, create equations for corresponding components, and add the new equations to the pool of equations.
 - otherwise:
 - `substitution := FAIL`

Solving the Type Equations System

5. Without loss of generality:
 If `te1s` is an atomic type
 and `te2s` is a composite type:
 `substitution := FAIL`
6. If there is a next equation:
 `equation := next(equations)`

Until `substitution = FAIL` or no more equations

Return: `substitution`

The algorithm is implemented in the procedures `solveEquations` and `solve`, which can be found [here](#).

Solving the Type Equations System

The logic of the unification is implemented in the procedures **canUnify** and **splitEquation**. These procedures transform an equation of the form:

$$[T_1 \times \dots \times T_n \rightarrow T] = [U_1 \times \dots \times U_n \rightarrow U]$$

into $n + 1$ equations of the form:

$$T_1 = U_1, \dots, T_n = U_n, T = U$$

The code for these procedures can be found [here](#).

How do we know that the `solve` algorithm terminates given a list of type expression equations?

The main loop of the algorithm has for state the current list of equations and the current substitution.

Let us consider the effect of one iteration
through the main loop:

Either we consume one equation from the current equations set and produce a more complex substitution (this happens when one of the sides of the first equation is an atomic type expression or a type variable).

Or we replace one equation with multiple equations: this happens when both sides of the equation are composite type expressions with compatible structure. In this case, we replace one equation with AST trees of depth D with n equations of depth $D - 1$ where n is the number of children of the ASTs.

In our case, composite ASTs in the type language are **ProcTExp** nodes which represent the type of procedures - with n children elements for the arguments of the procedure and one element for the return type.

Or we fail the solve process when we detect an incompatible equation.

The argument for completion is based on the characterization of the size of the input equation set as a pair (D, N) where D is the maximum height of the ASTs that appear in any equation in the equation set and N is the number of equations in the set.

Each iteration in the loop changes the size to either $(D, N - 1)$ or $(D - 1, N + n)$. When $D = 1$, the transition is necessarily to $(1, N - 1)$ because the only case where we add equations is for composite ASTs. Hence all transitions lead to the completion state of $(1, 0)$.

Putting all Steps Together: `inferType`

Putting all the steps of the algorithm together,
we define the procedures `infer` and
`inferType`. See the code [here](#).

Putting all Steps Together: inferType

Example:

```
infer("(lambda (f g) (lambda (n) (f (g n))))");  
// { tag: 'Ok',  
//   value: '((T_6 -> T_7) * (T_3 -> T_6) -> (T_3 -> T_7))' }
```

Observe the usage of freshly generated type variable names which are different each time we invoke the procedure. This makes it difficult to test the procedure.

To resolve this difficulty, we introduce the procedure `equivalentTEs` in `TExp.ts` which verifies that two type expressions are equivalent up to type variable renaming. This allows us to write tests in a deterministic manner (see [here](#)).

The implementation described above based on type equations follows literally the type equations algorithm. It explicitly manipulates substitution data structures and type equations. In addition, it constructs a map of expression to type variables to ensure the exhaustive traversal of the program to be type-checked.

We present here an optimized version of this algorithm which relies on a slightly modified representation of the type variable data structure. Using this new data structure for type variables, we implement exactly the same algorithm but avoid creating explicit data structures for the pool, the equations and the substitutions.

This leads to a more memory-efficient implementation, which also turns out to be more time efficient, as less traversals of the data structures are required, and operations performed eagerly in the type equations method are turned into lazy operations.

The extension to the **TVar** data type we introduce is implemented as follows:

Type Variable with One-way Assignment

```
// TVar: Type Variable with support for
//      dereferencing (TVar -> TVar)
type TVar = {
  tag: "TVar";
  var: string;
  contents: Box<undefined | TExp>;
};

const isEmptyTVar = (x: any): x is TVar =>
  (x.tag === "TVar") && unbox(x.contents) === undefined;
const makeTVar = (v: string): TVar =>
  ({tag: "TVar", var: v, contents: makeBox(undefined)});
const isTVar = (x: any): x is TVar =>
  x.tag === "TVar";
const eqTVar = (tv1: TVar, tv2: TVar): boolean =>
  tv1.var === tv2.var;
```

In addition to the name of the type variable, we associate a boxed value, initialized to **undefined**. We use this new field in the **TVar** datatype to associate the **TVar** to another type expression, when we derive a constraint that the variable must be bound to another type expression as part of the type inference process.

Type Variable with One-way Assignment

We extend the **TVar** data type with the following methods:

```
const tvarContents = (tv: TVar): undefined | TExp =>
  unbox(tv.contents);
export const tvarSetContents = (tv: TVar, val: TExp): void =>
  setBox(tv.contents, val);
export const tvarIsNonEmpty = (tv: TVar): boolean =>
  tvarContents(tv) !== undefined;
```

The assignment managed by **TVar** is **one-way** - we can only assign a value to an empty type variable.

Type Variable with One-way Assignment

In many occurrences, we will bind a **TVar** to another **TVar**. When this is the case, we are interested in accessing the type expression to which the referenced **TVar** refers. That is, we create a graph of **TVar** references to other **TVars** which eventually lead to non-**TVar** expressions. We want to follow the path of references from any **TVar** to a non-**TVar** value (which may be empty).

Type Variable with One-way Assignment

The method `tvarDeref` performs this graph traversal:

```
const tvarDeref = (te: TExp): TExp => {  
  if (! isTVar(te)) return te;  
  const contents = tvarContents(te);  
  if (contents === undefined)  
    return te;  
  else if (isTVar(contents))  
    return tvarDeref(contents);  
  else  
    return contents;  
};
```

Using the one-way assignment **TVar** data structure, we return to the procedure **checkEqualType** which we defined in the Type Checker implementation. The original implementation applied the **invariance check** and verified that two type expressions are identical (using the **equals** predicate from Ramda). When they were, the type checker proceeded - otherwise the type checking failed.

In the case of the type inference, we replace this type equal test with a different procedure, which instead of testing that two types are equal, attempts to **make them equal** when they contain type variables. The way two type expressions are made equal is by **unifying them** - that is, finding a substitution which when applied to both sides makes them equal.

But instead of representing the substitution as an explicit data structure (the **Sub** type we defined in `L5-substitution-adt.ts`), we encode the substitution bindings within the **TVar** data structure. When `tvar1` is bound to a type expression `s(tvar1)`, we invoke `tvarSetContents(tvar1, te)`.

The following procedure effectively computes the MGU (most general unifier) in-place given two expressions `te1` and `te2` which may contain type variables.

In exactly the same manner as we had to deal with the **occurs-check** case in the substitution data type, we must also avoid creating circular references in the graph of **TVar** references.

This is enforced in the **checkTVarEqualTypes** procedure which binds a **TVar** to a value - and makes sure the reference type expression does not include a reference to **TVar**.

Surprisingly, the type inference algorithm is **exactly the same code** as the Type Checker - except for the transformation of the procedure `checkEqualType` from a test of equality to the unification building version presented above.

The program that we obtain is in fact an implementation of the **type equation** algorithm
- with the following transformations:

There is no explicit pool representation - instead, we pre-allocate type variables in all possible **VarDecl** and procedure return positions as part of the expression parsing (in **parse**).

Application nodes and procedure nodes in the program AST are not explicitly annotated with type variables - but the type checking algorithm enforces exhaustive traversal of the AST in depth-first order. Each time an application or procedure node is encountered, the corresponding type equation is verified, and solved in place by invoking **checkEqualType** eagerly.

Note that when we invoke `checkEqualType` - the types may not yet be known, and an expression may still be attached to an unbound **TVar**. This happens for example when we infer types for the expression `((lambda (x) x) 1)` - when the operator component of this application is analyzed - there is not sufficient information to derive the type of the parameter `x`.

Later, when the typing rule of the application syntactic construct is applied (the top level node in the AST), the **TVar** associated to **x** will be bound to the type expression of the numeric atomic value. This will propagate the inferred information that **x** is a **NumTExp** type from the application to the procedure expression.

This propagation of information was not necessary in the case of the type checking algorithm - because we could rely on the fact that all variable references (**VarRef**) are explicitly typed.

We do not explicitly represent substitutions, instead we rely on the graph of **TVar** references as a representation of the substitution object.

We do not need the explicit renaming of the program as we can rely on the **TEnv** mechanism to capture scoping relations.

The implementation of unification through one-way variable assignment is a powerful technique, which we will revisit in Chapter 5 when we survey Logic Programming.