CS422 - Programming Language Design

Type Inference

Grigore Roşu

Department of Computer Science
University of Illinois at Urbana-Champaign

Checking that operations are applied on arguments of correct types statically has two major benefits:

- Allows efficient implementations of programming languages by assuming untyped memory models and therefore removing the need for runtime consistency checks, and
- Gives rapid feedback to users, so they can correct errors at early stages.

These advantages are typically considered so major in the programming language community, that the *potential drawbacks* of static typing, namely

- Limiting the way programs are written by rejecting programs which do not type-check, and
- Adding typing information may be space and time consuming,

are often ignored.

The first drawback is usually addressed by rewriting the code in a way that passes the type checking procedure. For example, programs passing a function as an argument to itself, which do not type check because of the recursive nature of the type of that function, can be replaced by equivalent programs using `let rec`.

It is, of course, desirable to require the user to provide *as little type information as possible* as part of a program. In general, depending on the application and domain of interest, there are quite *subtle trade-offs* between the amount of user-provided annotations and the degree of automation of the static analyzer.

In this lecture we address the problem of reducing the amount of required type information by devising automatic procedures that *infer the intended types from how the names are used*.

How much type information can be automatically and efficiently inferred depends again upon the particular domain of interest and

its associated type system. In this lecture we will see an extreme fortunate situation, in which *all* the needed type information can be inferred automatically.

More precisely, we will discuss a classical procedure that *infers all the types* of all the declared names in our functional programming language. A similar technique is implemented as part of the ML and OCaml languages. By "type", we here mean the language specific types, that is, those that were explicitly provided by users to the static type checker that we discussed.

Type Inference

Programs written in the functional language discussed so far contain all the information that one needs in order to infer the intended type of each name. By carefully collecting and using all this information, one can type check programs without the need for the user to provide any auxiliary type information.

Suppose that one writes the expression `x + y` as part of a larger expression. Then one can infer from here that `x` and `y` are both intended to be integers, because the arithmetic operation `_+_` is defined only on integers! Similarly, if

```
if x then y else z
```

occurs in an expression, then one can deduce, thanks to the typing policy associated to conditionals, that `x` has type `bool` and that `y` and `z` have the same type. Moreover, from

```
if x then y else z + t
```

one can deduce that `z` and `t` are both of type `integer`, implying that `y` is also `integer`. Type inference and type checking work smoothly together, in the sense that one implicitly checks the types while inferring information. For example, if

```
if x then y else z + x
```

is seen then one knows that there is a typing error because the type of `x` *cannot* be both `integer` and `bool`.

The type of functions can also be deduced from the way the functions are used. For example, if one uses `f x y` in some program then one can infer that `f` takes two arguments. Moreover, if `f x x` is seen then one can additionally infer that `f`'s arguments have the same type.

There can be possible that the result type of a function as well as the types of its arguments can all be inferred from the context, without even analyzing the definition of the function. For example,

if one writes

```
(f x y) + x + let x = y in x
```

then one first infers that `f` must return an `int` because it is used as an argument of `_+_`, then that `x` is an integer for the same reason, so the type of `f`'s first argument is `int`, and finally, because of the `let` which is used in a context of an `int`, that the type of `y` is also `int`. Therefore, the type of `f` must be `int -> int -> int`. In fact, the types of all the names occurring in the expression above were deduced by just analyzing carefully how they are used.

Let us now consider an entire expression which evaluates properly, for example one defining and using a factorial function:

```
let rec f n = if n eq 0
              then 1
              else n * f(n - 1)
in f 5
```

How can one automatically infer that this expression is type safe? Since `eq` takes two `int`s and returns a `bool`, one can deduce that the parameter of the function is `int`. Further, since the `f` occurs in the context of an integer and takes an expression which is supposed to be an integer as argument, `n - 1`, one can deduce that the type of that `f` is `int -> int`. Because of the semantics of `let rec`, the bound `f` will have the same type, so the type of the entire expression will be `int`.

One can infer/check the types differently, obtaining the same result.

Polymorphic Functions

Let us consider a "projection" function which takes two arguments and always returns its first argument, such as `fun x y -> x`. Without any additional type information, the best one can say about the type of this expression is that it is t_1 `->` t_2 `->` t_1, for some types t_1 and t_2. This function can be now used in any context in which its result type equals that of its first argument, such as, `(int -> int) -> int -> (int -> int)`. Such functions are called *polymorphic*. Their type should be thought of as the *most general*, in the sense of the *least constrained*, type that one can associate them.

Let us consider several other polymorphic functions. For example, the type of the function `fun x y -> x y`, which applies its first argument, expected therefore to be a function, to its second argument, is `(`t_1 `->` t_2`)` `->` t_1 `->` t_2.

How can one infer the most general type of an expression then, by just analyzing it syntactically? The process of doing this is called *type inference* and consists of two steps:

1. Collect information under the form of type parametric constraints by recursively analyzing the expression;
2. Solve those constraints.

Collecting Type Information

In order to collect type information from an expression, we traverse the expression recursively, assign generic types to certain names and expressions, and then constrain those types. Let us consider, for example., the expression

```
fun x y -> if x eq x + 1 then x else y
```

and let us manually simulate the type information collecting

algorithm.

We have a function expression. Without additional information, the best we can do is to assume some generic types for its parameters and then calculate the type of its body. Let t_x and t_y be the generic types of `x` and `y`, respectively. If t_e is the type of function's body expression, then the type of the function will be

$$t_x \texttt{ -> } t_y \texttt{ -> } t_e.$$

Let us now calculate the type t_e while gathering type information as we traverse the body of the function.

The body of the function is a conditional, so we can now state a first series of constraints by analyzing the typing policy of the conditional: its condition is of type `bool` and its two branching expressions must have the same type, which will replace t_e. Assuming that t_b, t_1 and t_2 are the types of its condition and branching expressions, respectively, then we can state the type

constrains

t_b = `bool`, and
t_1 = t_2.

We next calculate the types t_b, t_1 and t_2, collecting also the corresponding type information. The types t_1 and t_2 are easy to calculate because they can be simply extracted from the type environment: t_x and t_y, respectively (thus, the type equation t_1 = t_2 above is in fact t_x = t_y).

To type the condition `x eq x + 1`, one needs to use the typing policy of `_eq_`: takes two arguments of type `int` and returns type `bool`. Thus t_b is the type `bool`, but two new type constraints are generated, namely

t_3 = `int`, and
t_4 = `int`,

where t_3 and t_4 are the types of the two subexpressions of `_eq_`. t_3

evaluates to t_x; in order to evaluate t_4 one needs to apply the typing policy of `_+_`: takes two `int` arguments and returns an `int`. These generate the constraint

t_x `=` `int`.

Therefore, after "walking" through all the expression and analyzing how names were used, we collected the following useful typing information:

t_x `=` t_y,
t_x `=` `int`, and
the type of the original expression is t_x `->` t_y `->` t_x.

After we learn how to solve such type constraint equational systems we will be able to infer from here that the expression is correctly typed and its type is `int -> int -> int`.

Let us now consider the polymorphic function

`fun x y -> (x y) + 1`.

After assigning generic types t_x and t_y to its parameters, while applying the typing policy on `_+_` one can infer that the type of the expression `(x y)`, say $t_{(xy)}$, must be `int`. Also, the result type of the function must be `int`. When calculating $t_{(xy)}$, by applying the typing policy for function application, which say that the type of `(F E)` for expressions `F` and `E` is `T` whenever the type of `F` is `Tp -> T` and that of `E` is `Tp`, one infers the type equational constraint t_x `=` t_y `->` $t_{(xy)}$. We have accumulated the following type information:

$t_{(xy)}$ `= int`,
t_x `=` t_y `->` $t_{(xy)}$, and
the type of the function is t_x `->` t_y `-> int`.

We will be able to infer from here that the type of the function is `(`t `-> int) ->` t `-> int`, so the function is *polymorphic in* t.

The Constraint Collecting Technique

One can collect type constraints in different ways. We will do it by recursively traversing the expression to type only once, storing a type environment as well as type constrains. If the expression is

- a *name* then we return its type from the type environment and collect no auxiliary type constraint;
- an integer or a bool constant, then we return its type and collect no constraint;
- an *arithmetic operator* then we recursively calculate the types of its operand expressions accumulating the corresponding constraints, then add the type constraints that their type is int, and then return `int`;
- a *boolean operator* then we act like before, but return `bool`;

- a *conditional* then calculate the types of its three subexpressions accumulating all the type constraints, then add two more constraints: one stating that the type of its first subexpression is `bool`, and another stating that the types of its other two subexpressions are equal.
- a *function* then generate a fresh generic type for its parameter, if any, bind it in the type environment accordingly, calculate the type of the body expression in the new environment accumulating all the constraints, and then return the corresponding function type;
- a *function application* then generate a fresh type t, calculate the type of the function expression, say t_f, and that of the argument expression, say t_p accumulating all the type constraints, and then add the type constraint t_f `=` t_p `->` t.

- a *declaration* of names in a `let`, then calculate the types of the bound expressions, bind them to the corresponding names in the type environment, and then calculate and return the type of `let`'s body; there are no additional type constrains to accumulate (besides those in the bound expressions and body);
- a *declaration* of names in a `let rec`, then assign some generic types to these names in the type environment, calculate the types of bound expressions and accumulate the constraint(s) that these must be equal to the generic types assigned to `let rec`'s names, and finally calculate and return the type of `let rec`'s body;

- a *list expression*, then calculate the types of all the elements in the list accumulating all the corresponding constraints, and then add the additional constraints that all these types must be equal (let `T` be that type) and finally return the type of the list expression as `list T`;
- a *list operation*, such as `car`, `cdr`, `cons` or `null?`, then calculate the types of the corresponding subexpressions accumulating the constraints and then adding new, straightforward type constraints; for example, in the case of `cons(E,E')`, add the constraint that the type of `E'` must be `list T`, where `T` is the type of `E`;

- an *assignment* `X := E`, then calculate the types of `X` and `E`, and then add the constraint that these two types must be equal.
- a *sequential composition* `E ; E'`, then calculate the types of `E` and `E'` accumulating all the type constraints, then return the type of `E'` as the type of the composition;
- a *block*, then return the type `none` if the block is {}, and the type of `E` if the block is {`E`};
- a *loop* `while Cond Body`, then calculate the types of `Cond` and `Body`, add the constraint that the type of `Cond` must be `bool`, and then return the type `none`.

Solving the Type Constraints

We can pre-type all the other language constructs in a very similar way (this will be part of your homework). Thus, the operator `preType` will take an expression and return a pair `{T,S}`, where `T` is a term of sort `Type` and `eqns(S)` contains all the *type constraints* accumulated by traversing the program and applying the typing policy. We refer to terms `t(0)`, `t(1)`, ..., as *type variables*.

We should understand the result `{T,S}` of `preType(E)` as follows:

> The (final) type of `E` will be `T` in which all the type variables will be *substituted* by the types obtained after *solving* the system of equations in `S`. `E` is *not guaranteed* a type *a priori*! If any *conflict* is detected while solving the system then we say that the process of typing `E` failed, and we conclude that the expression is not correctly typed.

If `Eqns` is a set of type equations, that is, a term of sort `Equations`, and `T` is a type potentially involving type variables, then we let `Eqns[T]` denote the type obtained after solving `Eqns` and then substituting the type variables accordingly in `T`.

Once such a magic operation `_[_] : Equations Type -> [Type]` is defined, we can easily find the desired type of an expression:

```
ceq type(E) = eqns(S)[T] if {T,S} := preType(E) .
```

We next describe a simple equational technique to define the operation `_[_] : Equations Type -> [Type]`.

Homework Exercise 1 *Complete the definition of the type inferencer in* `fun-type-inference-semantics.maude`. *You need to pre-type the remaining language constructs and then to formalize the technique that will be next presented.*

The Type Inference Procedure by Examples

Example 1

Let us consider the following expression:

```
(fun x y -> x y) (fun z -> 2 * z * z * 3) 3
```

After pre-typing (do it, using `preType` on it), we get the pre-type `t(5)` and a state containing the four equations:

```
int = t(3),
t(0) = t(1) -> t(2),
t(4) = int -> t(5),
t(0) -> t(1) -> t(2) = (t(3) -> int) -> t(4)
```

How can we solve the system of equations above? We can immediately notice that `t(3) = int`, so we can just substitute `t(3)` accordingly in the other equations, obtaining:

```
t(0) = t(1) -> t(2),
t(4) = int -> t(5),
t(0) -> t(1) -> t(2) = (int -> int) -> t(4)
```

Moreover, since `t(0) = t(1) -> t(2)`, by substitution we obtain

```
t(4) = int -> t(5),
(t(1) -> t(2)) -> t(1) -> t(2) = (int -> int) -> t(4)
```

And further, by substituting `t(4)` into the last equation we get

```
(t(1) -> t(2)) -> t(1) -> t(2) = (int -> int) -> int -> t(5)
```

Both types in the equation above are function types, so their source and target domains must be equal. This observation allows us to generate two other type equations, namely:

```
(t(1) -> t(2)) -> t(1) = (int -> int) -> int,
t(2) = t(5).
```

Now, substituting `t(2)` by `t(5)` in the first equation we get:

```
(t(1) -> t(5)) -> t(1) = (int -> int) -> int
```

that is, an equality of function types, which yields the following:

```
t(1) -> t(5) = int -> int,
t(1) = int.
```

Substituting `t(1)` in the first equation and then equaling the source and target types of the function type, we get

```
int = int,
t(5) = int.
```

The first equation is useless and will be promptly removed by the simplifying rule `(T = T) = none` in `EQUATIONS`. The second equation gives the desired type of our expression, `int`.

Example 2

Let us now see an example expression which cannot be typed, e.g.:

```
let x = fun x -> x in (x x)
```

After pre-typing, we get the type `t(1)` constrained by:

```
t(0) -> t(0) = (t(0) -> t(0)) -> t(1)
```

We can already see that this equation is problematic, but let us just follow the procedure blindly to see what happens. Since both types in the equation are function types, their source and target types must be equal, so we can generate the following two equations:

```
t(0) = t(0) -> t(0),
t(0) = t(1)
```

The first equation is obviously problematic because of its "recursive" nature, but let us ignore it and substitute the second into the first, obtaining:

```
t(1) = t(1) -> t(1).
```

There is no way to continue. The variable `t(1)` cannot be assigned a type, so the original expression *cannot be typed*.

Unification

The technique that we used to solve the type equations is called *unification*. In general the equational unification problem can be stated as follows, where we only consider the mono-sorted case. Our particular unification problem fits this framework, because we can consider that we have only one sort, `Type`.

Let Σ be a signature over only one sort, i.e., a set of operations like in Maude, let X be a finite set of variables, and let

$$\mathcal{E} = \{t_1 = t'_1, ..., t_n = t'_n \mid t_1, t'_1, ..., t_n, t'_n \in T_\Sigma(X)\}$$

be a finite set of pairs of Σ-terms over variables in X, which from now on we call *equations*. Notice, however, that these are different from the standard equations in equational logic, which are quantified universally.

A map, or a substitution, $\theta : X \to T_\Sigma(X)$ is called a *unifier* of $\mathcal{E}$ if and only if $\theta^\star(t_i) = \theta^\star(t'_i)$ for all $1 \leq i \leq n$, where $\theta^\star : T_\Sigma(X) \to T_\Sigma(X)$ is the natural extension of $\theta : X \to T_\Sigma(X)$ to entire terms, by substituting each variable x by its corresponding term, $\theta(x)$.

A unifier $\theta : X \to T_\Sigma(X)$ is *more general* than $\varphi : X \to T_\Sigma(X)$ when φ can be obtained from θ by further substitutions; formally, when there is some other map, say $\rho : X \to T_\Sigma(X)$, such that $\varphi = \theta; \rho^\star$, where the composition of function was written sequentially. Let us consider again our special case signature, that of types, and the equations $\mathcal{E}$:

```
t(0) = t(1) -> t(2),
t(0) -> t(1) -> t(2) = (t(3) -> t(3)) -> (t(4) -> t(4)) -> t(5).
```

Here `t(0)`, ..., `t(5)` are seen as variables in X.

One unifier for these equations, say φ, takes `t(0)` to

$$\texttt{(int -> int) -> (int -> int)},$$

`t(1)`, `t(2)` and `t(3)` to `int -> int`, and `t(4)` to `int`. Another unifier, say θ, takes `t(0)` to

$$\texttt{(t(4) -> t(4)) -> (t(4) -> t(4))},$$

and `t(1)`, `t(2)` and `t(3)` to `t(4) -> t(4)`. Then it is clear that θ is *more general* than φ, because $\varphi = \theta; \rho^\star$, where ρ simply takes `t(4)` to `int`.

If a set of equations $\mathcal{E}$ admits a unifier, they are called *unifiable*. Note that there can be situations, like the ones we have already seen for our special case of signature of types, when a set of equations is not unifiable.

Finding the Most General Unifier

When a set of equations $\mathcal{E}$ is unifiable, its *most general unifier*, written $mgu(\mathcal{E})$ is one which is more general than any other unifier of $\mathcal{E}$. Note that typically there are more than one mgu.

We will next discuss a general procedure for finding an mgu for a set of equations $\mathcal{E}$ over an arbitrary mono-sorted signature Σ. Since we actually need to apply that mgu to the type calculated by the `preType` operation for the expression to type, we will provide a technique that directly calculates $\mathcal{E}[t]$ for any term t, which calculates the term after applying the substitution $mgu(\mathcal{E})$ to t, that is, $(mgu(\mathcal{E}))^\star(t)$.

The technique is in fact quite simple. It can be defined entirely equationally, but in order to make the desired direction clear we write them as rewriting rules (going from left to right). It consists

of iteratively applying the following two steps to $\mathcal{E}[t]$:

1. $(x = u, \mathcal{E})[t] \rightarrow subst(x, u, \mathcal{E})[subst(x, u, t)]$, if x is some variable in X, u is a term in $T_\Sigma(X)$ which does *not* contain any occurrence of x, and $subst(x, u, \mathcal{E})$ and $subst(x, u, t)$ substitute each occurrence of x in $\mathcal{E}$ and t, respectively, by u;

2. $(\sigma(t_1, ..., t_k) = \sigma(t'_1, ..., t'_k), \mathcal{E})[t] \rightarrow (t_1 = t'_1, ..., t_k = t'_k, \mathcal{E})[t]$, if $\sigma \in \Sigma$ is some operation of k arguments.

Theorem. *When none of the steps above can be applied anymore, we obtain a potentially modified final set of equations and a final term, say $\mathcal{E}_f[t_f]$. If $\mathcal{E}_f$ is empty then t_f is $(mgu(\mathcal{E}))^\star(t)$, so it can be returned as the type of the original expression. If $\mathcal{E}_f$ is not empty then $\mathcal{E}$ is not unifiable, so the original expression cannot be typed.*

Let Polymorphism

Unfortunately, our current polymorphic type inference technique is not as general as it could be. Suppose, for example, that one wants to define a function with the purpose of using it on expressions of various types, such as an identity function. Since the identity function types to a polymorphic type, `t(0) -> t(0)` or some similar type, one would naturally want to use it in different contexts. After all, that's the purpose of polymorphism. Unfortunately, our current type inferencer *cannot* type the following expression, even though it correctly evaluates to `int(1)`:

```
let f = fun x -> x
in if f true then f 1 else f 2
```

The reason is that `f true` will already enforce, via the type constraints, the type of `f` to be `true -> true`; then `f 1` and `f 2`

will enforce the type of `f` to be `int -> int`, thus leading to a contradiction. Indeed, our type inferencer will reduce $\mathcal{E}[t]$, where $\mathcal{E}$ is the set of constraints and t is the pre-type of this expression, to $\mathcal{E}_f[t_f]$, where $\mathcal{E}_f$ is the equation `bool=int` and t_f is `int`; since $\mathcal{E}_f$ is not empty, the original expression cannot be typed.

The example above may look artificial, because one does not have a need to define an identity function. Consider a length function instead, which is, naturally, intended to work on lists of any type:

```
fun x -> let c = 0
         in {
              while(not null?(x))
              {
                x := cdr(x) ;
                c := c + 1
              } ;
              c
            }
```

As expected, our type inferencer will type this expression to a polymorphic type, `list t(0) -> int`. However, if one uses it on a list of concrete type, then `t(0)` will be unified with that type, thus preventing one from using the length function on other lists of different types. For example,

```
let l = fun x -> let c = 0
                 in {
                      while(not null?(x))
                      {
                        x := cdr(x) ;
                        c := c + 1
                      } ;
                      c
                    }
in l [5 :: 3 :: 1] + l [true :: false :: true]
```

will not type, even though it correctly evaluates to `int(6)`.

This leads to the following natural question, that a programming language designer wishing a type system on top of his/her language must address: what should one do when valid programs are rejected by the type system? One unsatisfactory possibility is, of course, to require the programmers write their programs differently, so that they will be accepted by the type system. Another, better possibility is to improve the type system to accept a larger class of programs.

Since *reusing* is the main motivation for polymorphism, it would practically make no sense to disallow the use of polymorphic expressions in different contexts. Therefore, programming language scientists fixed the problem above by introducing the important notion of *let-polymorphism*.

The idea is to type the `let` language constructs differently. Currently, we first type all the binding expressions to some (pre-)types, then we bind those types to the corresponding names

into the type environment, and finally we type the body expression in the new environment. This technique is inherently problematic with respect to reuse, because a (polymorphic) binding expression is enforced to have two different types if used in two different contexts, which is not possible.

The idea of *let-polymorphism* is to first *substitute* all the free occurrences of the bound names into the body of the `let` expression by the corresponding binding expressions, and then to type directly the body expression. This way, each use of a bound name will be typed locally, thus solving our problem.

Exercise 1 *Modify our current type inferencer to include let-polymorphism. How about* `let rec`*?*

However, a complexity problem is introduced this way: the size of the code can grow exponentially and the binding expressions will be (pre-)typed over and over again, even though each time they

will be typed to "almost" the same type (their types will differ by only the names of the type variables). There is a partial solution to this problem, too, but that is a more advanced topic that will be covered in CS522 next semester.

Type Systems May Reject Correct Programs

While static type checkers/inferencers offer a great help to programmers to catch (trivial) errors at early stages in their programs and to programming language implementers to generate more efficient code (by assuming the untyped memory model), they have an inherent drawback: they *disallow correct programs which do not type check*. In the previous section we saw a fortunate case where a type system could be relatively easily extended (let-polymorphism), to allow polymorphic definitions to be used as desired. Unfortunately, due to undecidability reasons, for any given

(strong) type system, there will always be programs which can evaluate correctly but which will not type check.

In our context, for example, consider the following expression/program for calculating the length of a list *without* using `let rec` or loops:

```
let t f x = if null?(x) then 0 else 1 + f f cdr(x)
in let l x = t t x
   in l [4 :: 7 :: 2 :: 1 :: 9]
```

The idea in the program/expression above is to pass the function as an argument to itself; this way, it will be available, or "seen", in its own body, thus simulating the behavior of `let rec` without using it explicitly. That is how (functional) programmers wrote recursive programs in the early days of functional programming, when the importance of static scoping was relatively accepted but no clean support for recursion was available. It, indeed, evaluates to `int(5)` but does not type. Try it!

One may wrongly thing that the above does not type because our type system does not include let-polymorphism. To see that let-polymorphism is not the problem here, one can apply the substitutions by hand and thus eliminate the `let` constructs entirely:

```
(fun f x -> if null?(x) then 0 else 1 + f f cdr(x))
  (fun f x -> if null?(x) then 0 else 1 + f f cdr(x))
    [4 :: 7 :: 2 :: 1 :: 9]
```

The above will again evaluate to `int(5)`, but will not type. The set of constraints is reduced to `t(6) = t(6) -> list int -> int`, reflecting the fact that there is some expression whose type has a recursive behavior. Indeed, the type of `f` (any of them) cannot be inferred due to its recursive behavior.

In fact, the problematic expression is

```
fun f x -> if null?(x) then 0 else 1 + f f cdr(x)
```

which cannot be typed because of the circularity in the type of `f`. No typed functional languages can actually type this expression. In ML, for example, the same expression, which in ML syntax is

```
fn (f,x) => if null(x) then 0 else 1 + f f tl(x)
```

generates an error message of the form:

```
t.sml:16.36-16.49 Error: operator is not a function [circularity]
  operator: 'Z
  in expression:
    f f
```