Lecture Notes on Type Inference

15-312: Foundations of Programming Languages
Frank Pfenning

Lecture 19
October 30, 2003

In the previous lecture on type checking, our goal was first to ensure that every expression synthesized a unique type and later, that every expression synthesized a principal type. We accomplished this by a mode analysis of the typing rules, which allowed us to read a set of typing rules as describing an algorithm.

The principal judgments (here indicated with their modes, where $()^+$ means *input* and $()^-$ means *output*) are:

$\Gamma^+ \vdash e^+ \uparrow \tau^-$	e synthesizes principal type τ
$\tau^+ \sqsubseteq \sigma^+$	τ is a subtype of σ
$\tau^+ \sqcup \sigma^+ \Rightarrow \rho^-$	ρ is the least upper bound of τ and σ
$\tau^+ \sqcap \sigma^+ \Rightarrow \rho^-$	ρ is the greatest lower bound of τ and σ

Any of these judgments might fail when executed with the given input constituents, which means that type synthesis fails. Overall, we use this in a theorem which guarantees that $\Gamma \vdash e : \tau$ if and only if $\Gamma \vdash e \uparrow \sigma$ and $\sigma \sqsubseteq \tau$.

However, programs in this language remain excessively verbose, especially when advanced type constructs are involved. Consider the example of recursive types. Using mode analysis we obtain the following rules.

$$\frac{\Gamma \vdash e \uparrow \{\mu t.\sigma/t\}\sigma}{\Gamma \vdash \mathtt{roll}(t.\sigma, e) \uparrow \mu t.\sigma} \qquad \frac{\Gamma \vdash e \uparrow \mu t.\sigma}{\Gamma \vdash \mathtt{unroll}(e) \uparrow \{\mu t.\sigma/t\}\sigma}$$

The unfortunate property here is that we need to endow the roll construct with a type in order to guarantee principal types. It is possible to consider roll and unroll as coercions, but various problems such as deciding subtyping and existence or lack of principal becomes very difficult and delicate, so we will not pursue this here. The example below shows some redundant information, necessitated by the decision to always propagate types upward when type-checking.

$$\begin{array}{lcl}
\mathsf{nat} & = & \mu t.1{+}t \\
\mathsf{zero} & = & \mathtt{roll}(\mathsf{nat}, \mathtt{inl}(\mathsf{nat}, \mathtt{unitel})) \\
\mathsf{succ} & = & \mathtt{fn}(\mathsf{nat}, x.\mathtt{roll}(\mathsf{nat}, \mathtt{inr}(1, x)))
\end{array}$$

The question is how to improve on the situation. A simple mechanism for improvement is to split the one judgment for synthesis into two, mutually dependent judgments. One synthesizes a type from an expression, the other analyzes an expression against a type. The idea is by propagating information in two directions, we can save significantly in the verbosity of the type system. This is called a *bidirectional system*.

We would like to preserve the determinism and syntax-directed nature of type-checking. We therefore consider for each construct whether it should synthesize or analyze. It turns out that judgments of least upper bound and greatest lower bound are also no longer needed, providing another benefit of the typing rules.

$$\begin{array}{ll}
\Gamma^+ \vdash e^+ \uparrow \tau^- & e \text{ synthesizes } \tau \\
\Gamma^+ \vdash e^+ \downarrow \tau^+ & e \text{ checks against } \tau \\
\tau^+ \sqsubseteq \sigma^+ & \tau \text{ is a subtype of } \sigma
\end{array}$$

The subtype judgment is the same as before; the other two look significantly different from a pure synthesis judgment.

Generally, for *constructors* of a type we can propagate the type information *downward* into the term, which means it should be used in the analysis judgment $e^+ \downarrow \tau^+$. Conversely, the *destructors* generate a result of a smaller type from a constituent of larger type and can therefore be used for synthesis, propagating information *upward*.

We consider some examples. First, functions. A function constructor will be checked, and application synthesizes, in accordance with the reasoning above.

$$\frac{\Gamma, x{:}\tau_1 \vdash e \downarrow \tau_2}{\Gamma \vdash \mathtt{fn}(\tau_1, x.e) \downarrow \tau_1 \to \tau_2} \qquad \frac{\Gamma \vdash e_1 \uparrow \tau_2 \to \tau_1 \quad \Gamma \vdash e_2 \downarrow \tau_2}{\Gamma \vdash \mathtt{apply}(e_1, e_2) \uparrow \tau_1}$$

Careful checking against the desired modes is required. In particular, the order of the premises in the rule for application is critical so that τ_2 is available to check e_2. Note that unlike in the case of pure synthesis, no subtype checking is required at the application rule. Instead, this must be handled implicitly in the definition of $\Gamma \vdash e_2 \downarrow \tau_2$. In fact, we will need a general rule that mediates between the two directions. This rule replaces subsumption in the general system.

$$\frac{\Gamma \vdash e \uparrow \tau \quad \tau \sqsubseteq \sigma}{\Gamma \vdash e \downarrow \sigma}$$

Note that the modes are correct: Γ, e, and σ are known as inputs in the conclusion. This means that Γ and e are known and τ is free, so the first premise is mode-correct. This yields a τ as output (if successful). This means we can now check if $\tau \sqsubseteq \sigma$, since both τ and σ are known.

For sums, the situation is slightly trickier, but not much. Again, the constructors are checked against a given type.

$$\frac{\Gamma \vdash e \downarrow \tau_1}{\Gamma \vdash \texttt{inl}(e) \downarrow \tau_1\texttt{+}\tau_2} \qquad \frac{\Gamma \vdash e \downarrow \tau_2}{\Gamma \vdash \texttt{inr}(e) \downarrow \tau_1\texttt{+}\tau_2}$$

For the destructor, we go from $e \uparrow \tau_1\texttt{+}\tau_2$ to the two assumptions $x_1{:}\tau_1$ and $x_2{:}\tau_2$ in the two branches. These assumptions should be seen as synthesis, variable synthesize their type from the declarations in Γ (which are given).

$$\frac{}{\Gamma_1, x{:}\tau, \Gamma_2 \vdash x \uparrow \tau}$$

$$\frac{\Gamma \vdash e \uparrow \tau_1\texttt{+}\tau_2 \quad \Gamma, x{:}\tau_1 \vdash e_1 \downarrow \sigma \quad \Gamma, x{:}\tau_2 \vdash e_2 \downarrow \sigma}{\Gamma \vdash \texttt{case}(e, x_1.e_1, x_2.e_2) \downarrow \sigma}$$

Here, both branches are checked against the same type σ. This avoids the need for computing the least upper bound, because one branch might synthesize σ_1, the other σ_2, but they are checked separately against σ. So σ must be an upper bound, but since we don't have to synthesize a principal type we never need to compute the least upper bound.

Finally, we consider recursive types. The simple idea that constructors (here: `roll`) should be checked against a type and destructors (here: `unroll`) should synthesize a type avoids any annotation on the type.

$$\frac{\Gamma \vdash e \downarrow \{\mu t.\sigma/t\}\sigma}{\Gamma \vdash \mathsf{roll}(e) \downarrow \mu t.\sigma} \qquad \frac{\Gamma \vdash e \uparrow \mu t.\sigma}{\Gamma \vdash \mathsf{unroll}(e) \uparrow \{\mu t.\sigma/t\}\sigma}$$

This seems too good to be true, because so far we have not needed *any* type information in the terms! However, there are still a multitude of situations where we need a type, namely where an expression requires a type to be checked, but we are in synthesis mode. Because of our general philosophy, this happens precisely where a destructor is meets a constructors, that is, where we can apply reduction in the operational semantics! For example, in the expression

```
(fn x => x) 3
```

the function part of the application is required to synthesize, but `fn x => x` can only be checked.

The general solution is to allow a type annotation at the place where synthesis and analysis judgments meet in the opposite direction from the subsumption rule shown before. This means we require a new form of syntax, $e : \tau$, and this is the *only* place in an expression where a type needs to occur. Then the example above becomes

```
(fn x => x : int -> int) 3
```

From this example it should be clear that bidirectional checking is not necessarily advantageous over pure synthesis, at least with the simple strategy we have employed so far.

$$\frac{\Gamma \vdash e \downarrow \tau}{\Gamma \vdash (e : \tau) \uparrow \tau}$$

Looking back at our earlier example, we obtain:

$$\begin{array}{lcl} \mathsf{nat} & = & \mu t.1{+}t \\ \mathsf{zero} & = & \mathtt{roll}(\mathtt{inl}(\mathtt{unitel})) : \mathsf{nat} \\ \mathsf{succ} & = & \mathtt{fn}(x.\mathtt{roll}(\mathtt{inr}(x))) : \mathsf{nat} \rightarrow \mathsf{nat} \end{array}$$

One reason this seems to work reasonably well in practice that code rarely contains explicit redexes. Programmers instead tend to turn them into definitions, which then need to be annotated. So the rule of thumb is that in typical programs one needs to annotate the outermost functions and recursions, and the local functions and recursions, but not much else.

With these ideas in place, one can prove a general soundness and completeness theorem with respect to the original subtyping system. We will not do so yet, but move on to discuss full type inference.

The idea here is that for a fragment of the language (which does *not* include full recursive, existential, or universal types) we can in fact solve the *full inference* problem where we are given an expression without *any* type information and have to determine the set of all types, giving it some finitary description. Consider the simple example

```
    fn x => x
```

This expression clearly has type $\tau \to \tau$ for any type τ. We can express this is ML by saying

```
    fn x => x : 'a -> 'a
```

where `'a` is a type variable that is (implicitly) universally quantified. In MinML we might express this with

$$\cdot \vdash \texttt{fn}(x.x) : \forall t.\, t \to t$$

but we have to be careful to recognize that the universal type constructor $\forall t$ has a different interpretation than before. This is because the expression we assign such a type does not have explicit type abstractions and instantiations as in our previous discussion of parametric polymorphism.

Ignoring the nature of quantification for now, the question is how we actually carry out type inference (for the fragment where it is possible). We repeat a few rules for *type assignment*, which is the name for a system where we assign types to expressions containing no explicit types themselves.

We begin with functions and variables.

$$\frac{}{\Gamma_1, x{:}\tau, \Gamma_2 \vdash x : \tau} \quad \frac{\Gamma, x{:}\tau_1 \vdash e : \tau_2}{\Gamma \vdash \texttt{fn}(x.e) : \tau_1 \to \tau_2} \quad \frac{\Gamma \vdash e_1 : \tau_2 \to \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash \texttt{apply}(e_1, e_2) : \tau_1}$$

No matter how we try to assign modes to the $\Gamma \vdash e : \tau$ judgment, these three rules will not be well-moded at the same time. If we picture how type inference should work in practice, it makes most sense to assume that Γ and e are given, and we synthesize a type τ for e in context Γ. This works for the variable rule, but breaks down for functions and for applications.

In order to turn this into a judgment that can be executed, we introduce placeholders for types that we cannot determine at the time a rule is executed. We then collect constraints on these placeholders in the form of equations (or, more generally, inequalities). Any solution to the constraint equations should yield a valid typing derivation, and any valid typing derivation in the type assignment system should yield a system solution to the constraints. We call this *constraint-based type inference*.

We write this as a new judgment $\Gamma \vdash e \Longrightarrow \tau \mid C$, stating that in context Γ the type inference on e yields type τ under constraints C. We only need very simple constraints here since we do not consider subtyping or other difficult constructs ($\forall$, $\exists$, μ); much more complex system are imaginable and have in fact been designed and used in practice.

$$\text{Constraints} \quad C \quad ::= \quad \cdot \mid \tau \doteq \sigma, C$$

We write the placeholders as *existential variables* α, β, etc, although we will not make the quantification over them explicit for now. Since existential variables are just a new syntactic notation for types we consider them as *known* during the constraint generation process.

$\Gamma^+ \vdash e^+ \Longrightarrow \tau^- \mid C^-$ Generation on e wrt. Γ yields type τ under C

Revisiting the earlier typing rules, it is easy to see where constraints need to be generated. Note that constraint generation should *never* fail, even if the term is ill-typed. In that case, the constraints should just not have a solution.

$$\frac{}{\Gamma_1, x{:}\tau, \Gamma_2 \vdash x \Longrightarrow \tau \mid \cdot} \qquad \frac{\Gamma, x{:}\alpha_1 \vdash e \Longrightarrow \tau_2 \mid C \quad (\alpha_1 \text{ new})}{\Gamma \vdash \mathtt{fn}(x.e) \Longrightarrow \alpha_1 \to \tau_2 \mid C}$$

$$\frac{\Gamma \vdash e_1 \Longrightarrow \tau \mid C_1 \quad \Gamma \vdash e_2 \Longrightarrow \tau_2 \mid C_2 \quad (\alpha_1 \text{ new})}{\Gamma \vdash \mathtt{apply}(e_1, e_2) \Longrightarrow \alpha_1 \mid C_1, C_2, \tau \doteq \tau_2 \to \alpha_1}$$

As examples, we have the following judgments

$$\begin{array}{l}
\cdot \vdash \mathtt{fn}(x.x) \Longrightarrow \alpha \to \alpha \mid \cdot \\
\cdot \vdash \mathtt{apply}(\mathtt{fn}(x.x), \mathtt{num}(3)) \Longrightarrow \alpha_2 \mid \alpha_1 \to \alpha_1 \doteq \mathsf{int} \to \alpha_2 \\
\cdot \vdash \mathtt{apply}(\mathtt{num}(3), \mathtt{num}(3)) \Longrightarrow \alpha \mid \mathsf{int} \doteq \mathsf{int} \to \alpha \\
\cdot \vdash \mathtt{fn}(x.\mathtt{apply}(x, x)) \Longrightarrow \alpha_1 \to \alpha_2 \mid \alpha_1 \doteq \alpha_1 \to \alpha_2
\end{array}$$

Note that the first two are well-typed, while the latter two are not. In the earlier type system, this simply meant that no typing derivation existed. Here, the collected constraints are unsatisfiable. For example, there is no type α such that $\mathsf{int} \doteq \mathsf{int} \to \alpha$. Similarly, there can be no types α_1 and α_2 such that $\alpha_1 \doteq \alpha_1 \to \alpha_2$, because the right-hand side of the equation will always be larger than the left-hand side.

It is very important to recognize the difference between the judgment $\tau = \sigma$ (defined only by the reflexivity axiom $\tau = \tau$) and and equation $\tau \doteq \sigma$ which is a new syntactic construct. We now need a new judgment to determine if a set of contraints is satisfiable, that is, if there is a substitution for the existential variables that makes all the equations true.

The algorithm for solving the set of equations is written once again as an inference system. We define a judgment

$\tau^+ \mid C^+ \implies \tau'^-$ type τ under constraints C becomes τ'

where the output τ' is the type τ after the solution to the constraints has been applied. Note that if the constraints are inconsistent (that is, have no solution), then the process of computing τ' will fail.

The first rule states that the empty set of constraints has a solution, without any need for substitution for type variables.

$$\frac{}{\tau \mid \cdot \implies \tau}$$

The remaining rules break down the constraints into simpler constraint, when read in the bottom-up direction until we either fail or reach the case above. The first category of rules just eliminates basic types that are seen as equal to themselves.

$$\frac{\tau \mid C \implies \tau'}{\tau \mid \mathsf{int} \doteq \mathsf{int}, C \implies \tau'} \quad \frac{\tau \mid C \implies \tau'}{\tau \mid \mathsf{bool} \doteq \mathsf{bool}, C \implies \tau'} \quad \frac{\tau \mid C \implies \tau'}{\tau \mid \alpha \doteq \alpha, C \implies \tau'}$$

There are similar rules to break down constructors. We only show the case of function types and disjoint sums.

$$\frac{\tau \mid \sigma_1 \doteq \rho_1, \sigma_2 \doteq \rho_2, C \implies \tau'}{\tau \mid \sigma_1 \to \sigma_2 \doteq \rho_1 \to \rho_2, C \implies \tau'} \qquad \frac{\tau \mid \sigma_1 \doteq \rho_1, \sigma_2 \doteq \rho_2, C \implies \tau'}{\tau \mid \sigma_1 + \sigma_2 \doteq \rho_1 + \rho_2, C \implies \tau'}$$

In both of these cases, one equation is replaced by two smaller ones, eliminating the top-level type constructor. Similar rules apply to other type constructors. We do not need any rules for mismatches, for example $\sigma_1 \to \sigma_2 \doteq \rho_1 + \rho_2$. Such equations have no solution, which is modeled by simply having no rules for them.

This reduces us to the case where at least one side of the first equation in C is a type variable. In that case we know the variable must be equal to the other side, and we can simply substitute. However, there is one subtlety to consider. Consider the earlier example of `fn x => x x`. In the generated constraints, we have an equation $\alpha_1 \doteq \alpha_1 \to \alpha_2$ which has no solution, because no matter what we substitute for α_1, the right-hand side will always be larger than the left-hand side. We therefore need a condition: when processing an equation $\alpha \doteq \sigma$ we can subsitute σ for α, but only if α does not occur in σ.

$$\frac{\{\sigma/\alpha\}\tau \mid \{\sigma/\alpha\}C \implies \tau' \quad (\alpha \text{ not in } \sigma)}{\tau \mid \alpha \doteq \sigma, C \implies \tau'}$$

$$\frac{\{\sigma/\alpha\}\tau \mid \{\sigma/\alpha\}C \implies \tau' \quad (\alpha \text{ not in } \sigma) \quad (\sigma \text{ not a type variable})}{\tau \mid \sigma \doteq \alpha, C \implies \tau'}$$

Note that we do not substitute in the output τ': when this is finally generated at the axiom $\tau \mid \cdot \implies \tau$, all substitutions have already taken place in τ. Therefore, all the rules are well-moded. If we have an equation $\alpha_1 \doteq \alpha_2$, both rules may apply. In this case it does not really matter which one we use, so we arbitrarily restrict the second rule so only the first one applies. In order to understand these rules best, the reader should trace their behavior in the examples above.

We say a substitution θ for some free type variables *unifies* $\sigma \doteq \rho$ if $\{\theta\}\sigma = \{\theta\}\rho$. Note the use of the (ordinary!) type equality judgment in this definition. We say a substitution θ unifies a collection of equations C if it unifies every equation in C.

The following theorem states some properties of the algorithm above.

Theorem 1 (Correctness of Unification)

(i) If $\tau \mid C \implies \tau'$ then there is a substitution for θ that unifies C and, moreover, $\{\theta\}\tau = \tau'$.

(ii) If there is a substitution θ that unifies C, then for any τ there is a τ' such that $\tau \mid C \implies \tau'$.

The proofs are not difficult, but they are tricky in the details. It remains to relate constraint synthesis and solving to type assignment.

Theorem 2 (Correctness of Constraint-Based Inference)

(i) For every Γ and e such that all free variables of e are declared in Γ, there exist a τ and C such that $\Gamma \vdash e \Longrightarrow \tau \mid C$.

(ii) If $\Gamma \vdash e \Longrightarrow \tau \mid C$ and $\tau \mid C \implies \tau'$, then $\Gamma \vdash e : \tau'$.

(iii) If $\Gamma \vdash e : \sigma$ and $\Gamma \vdash e \Longrightarrow \tau \mid C$ then $\tau \mid C \implies \tau'$ and there is a substitution θ such that $\{\theta\}\tau' = \sigma$.

Again, the proofs are not difficult on the language fragment we have considered here, but they are tricky since they involve substitutions and freshness conditions. In particular, we must choose different fresh variables in different branches of a derivation that generates constraints, a condition which is difficult to capture formally with the mechanisms presented here.

The main aspect of ML that the above analysis does not cover is the behavior of `let` and top-level definitions. For example, the following is well-typed:

```
let f = (fn x => x)
in
  (f 3, f true)
end
```

The reason it is well-typed is after we infer a type $\alpha \rightarrow \alpha$ for the identity function, we assign f a type which is equivalent to $\forall t.\, t \rightarrow t$. It implicitly instantiates t to int at the first occurrence and bool at the second occurrence. In order to model this more formally, we need to introduce the notion of a *type scheme*, which is a (non-polymorphic) type with a sequence of leading universal quantifiers.

We define $\mathsf{Gen}(\tau)$ for the result of quantifying over all free (existential) type variables in τ and $\mathsf{Inst}(\sigma)$ for the result of instantiating all quantified types in the scheme σ with some new (existential) type variables. For example, $\mathsf{Gen}(\alpha_1 \rightarrow \alpha_2 \rightarrow \alpha_1) = \forall t_1.\, \forall t_2.\, t_1 \rightarrow t_2 \rightarrow t_1$ and $\mathsf{Inst}(\forall t_1.\, \forall t_2.\, t_1 \rightarrow t_2 \rightarrow t_1) = \alpha_1 \rightarrow \alpha_2 \rightarrow \alpha_1$.

When we see a top-level definition $x = e$ when processing in the interactive loop or reading a file, we can then proceed as follows:

1. Compute $\Gamma \vdash e \Longrightarrow \tau \mid C$. This must always succeed.
2. Solve $\tau \mid C \Longrightarrow \tau'$. This may fail, in which case we report a type error.
3. Otherwise we can now (compile and) evaluate e yielding v.
4. Generalize $\sigma = \mathsf{Gen}(\tau')$
5. Add $x{:}\sigma$ to the pervasive context for type-checking and $x = v$ to the pervasive environment for evaluation.

The constraint generation rule for variables now has to be modified to allow instantiation of the type schemes that are assigned to variables in the pervasive context. It is important that all generated existential type variables are new and do not already occur in the context.

$$\frac{\mathsf{Inst}(\sigma) = \tau}{\Gamma_1, x{:}\sigma, \Gamma_2 \vdash x \Longrightarrow \tau \mid \cdot}$$

Handling local let-definitions in a similar manner is more difficult to describe, because it forces us to interleave unification (that is, constraint solving) with type inference and generalization (the Gen operation), so we will not formally describe it here.

A less realistic alternative that is easy to describe is to transform embedded occurrences of $\texttt{let}(e_1, x.e_2)$ to $\{e_1/x\}e_2$ for the purposes of type-checking. This means for every occurrence of x in e_2 we have a separate copy of e_1 that can be type-checked differently. If some provision is made so that at least one copy of e_1 survives, this is a sound and complete method in the absence of effects. When we add effects, several copies of e_1 will behave differently than a single copy, so we need to restrict e_1 to be a value; if e_1 is not a value we leave the `let` and infer a single type for e_1.

Unfortunately, constrained type inference of the style of this lecture (and therefore in the style of Standard ML) breaks down under various useful extensions of the type system, such as fully general universal, existential, and recursive types. Interestingly, subtyping by itself still works, although the constraint solving algorithm is more complex. The Standard ML language (which does not support subtyping) adopts various solutions to retain (almost) full type inference. Universal types are restricted to occur in the prefix of a type, as described in this lecture. Existential types are modeled only at the level of modules where signatures are available to check the implementation against. This is unavoidable, since existential types provide an mechanism for data abstraction that is incompatible with full inference: it could not possibly guess what to hold abstract and what not. Recursive types can be defined only via the `datatype` construct so that they remain abstract and don't cause any deep problems in type reconstruction. Furthermore, datatype constructors and pattern matching expressions are compiled into the proper roll and unroll expressions after type inference.