Lecture Notes on Bidirectional Type Checking

15-312: Foundations of Programming Languages
Frank Pfenning

Lecture 17
October 21, 2004

At the beginning of this class we were quite careful to guarantee that every well-typed expression has a unique type. We relaxed our vigilance a bit when we came to constructs such as universal types, existential types, and recursive types, essentially because the question of unique typing became less obvious or, as in the case of existential types, impossible without excessive annotations.

In this lecture we first recall the notion of modes and mode correctness that allow us to interpret inference rules as an algorithm. We then apply this idea to develop an algorithm that propagates type information through an abstract syntax tree in two directions, allowing for a more natural type-checking algorithm we call *bidirectional*.

In either case, it is convenient to think of type checking as the process of bottom-up construction of a typing derivation. In that way, we can interpret a set of typing rules as describing an algorithm, although some restriction on the rules will be necessary (not every set of rules naturally describes an algorithm).

The idea behind modes is to label the constituents of a judgment as either *input* or *output*. For example, the typing judgment $\Gamma \vdash e : \tau$ should be such that Γ and e are input and τ is output (if it exists). We then have to check each rule to see if the annotations as input and output are consistent with a bottom-up reading of the rule. This proceeds as follows, assuming at first a single-premise inference rule. We refer to constituents of a judgment as either *known* or *free* during a particular stage of proof construction.

1. **Assume** each input constituent of the *conclusion* is known.

2. **Show** that each input constituent of the *premise* is known, and each output constituent of the premise is still free (unknown).

3. **Assume** that each output constituent of the *premise* is known.

4. **Show** that each output constituent of the *conclusion* is known.

Given the intuitive interpretation of an algorithm as proceeding by bottom-up proof construction, this method of checking should make some sense intuitively. As an example, consider the rule for functions.

$$\frac{\Gamma, x{:}\tau_1 \vdash e : \tau_2}{\Gamma \vdash \mathtt{fn}(\tau_1, x.e) : \tau_1 \to \tau_2}\ \mathit{FnTyp}$$

with the mode

$$\Gamma^{+} \vdash e^{+} : \tau^{-}$$

where we have marked inputs with + and outputs with -.

1. We **assume** that Γ, τ_1, and $x.e$ are known.

2. We **show** that $\Gamma, x{:}\tau_1$ and e are known and τ_2 is free, all of which follow from assumptions made in step 1.

3. We **assume** that τ_2 is also known.

4. We **show** that τ_1 and τ_2 are known, which follows from the assumptions made in steps 1 and 3.

Consequently our rule for function types is mode correct with respect to the given mode. If we had omitted the type τ_1 in the syntax for function abstraction, then the rule would not be mode correct: we would fail in step 2 because $\Gamma, x{:}\tau_1$ is not known because τ_1 is not known.

For inference rules with multiple premises we analyze the premises from left to right. For each premise we first show that all inputs are known and outputs are free, then assume all outputs are known before checking the next premise. After the last premise has been checked we still have to show that the outputs of the conclusion are all known by now. As an example, consider the rule for function application.

$$\frac{\Gamma \vdash e_1 : \tau_2 \to \tau \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash \mathtt{apply}(e_1, e_2) : \tau}\ \mathit{AppTyp}$$

Applying our technique, checking actually fails:

1. We **assume** that Γ, e_1 and e_2 are known.
2. We **show** that Γ and e_1 are known and τ_2 and τ are free, all which holds.
3. We **assume** that τ_2 and τ are known.
4. We **show** that Γ and e_2 are known and τ_2 is free. This latter check fails, because τ_2 is known at this point.

Consequently have to rewrite the rule slightly. This rewrite should be obvious if you have implemented this rule in ML: we actually first generate a type τ_2' for e_2 and then compare it to the domain type τ_2 of e_1.

$$\frac{\Gamma \vdash e_1 : \tau_2 \to \tau \quad \Gamma \vdash e_2 : \tau_2' \quad \tau_2' = \tau_2}{\Gamma \vdash \mathtt{apply}(e_1, e_2) : \tau}\ \mathit{AppTyp}$$

We consider all constitutents of the equality check to be input ($\tau^+ = \sigma^+$). This now checks correctly as follows:

1. We **assume** that Γ, e_1 and e_2 are known.
2. We **show** that Γ and e_1 are known and τ_2 and τ is free, all which holds.
3. We **assume** that τ_2 and τ are known.
4. We **show** that Γ and e_2 are known and τ_2' is free, all which holds.
5. We **assume** that τ_2' is known.
6. We **show** that τ_2 and τ_2' are known, which is true.
7. We **assume** the outputs of the equality to be known, but there are no output so there are no new assumption.
8. We **show** that τ (output in the conclusion) is known, which is true.

Now we can examine other language constructs and typing rules from the same perspective to arrive at a bottom-up inference system for type checking. We forego this exercise here, and instead consider what can be gained by introducing two mutually recursive judgments: one for expressions that have enough information to synthesize a type, and one for situations where we know what type to expect so we propagate it downward in the tree.

$\Gamma^+ \vdash e^+ \uparrow \tau^-$	e synthesizes τ
$\Gamma^+ \vdash e^+ \downarrow \tau^+$	e checks against τ
$\tau^+ \sqsubseteq \sigma^+$	τ is a subtype of σ

The subtype judgment is the same as $\tau \leq \sigma$, except that we omit the rule of transitivity which is not mode correct; 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.

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

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.

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

For the destructor, we go from $e \uparrow \tau_1{+}\tau_2$ 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).

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

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

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:

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

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 this here, but move on to discuss the form of subtyping that is amenable to an algorithmic interpretation. In other words, we want to write out a judgment $\tau \sqsubseteq \sigma$ which holds if and only if $\tau \leq \sigma$, but which is mode-correct when both τ and σ are given.

The difficulty in the ordinary subtyping rules is transitivity

$$\frac{\tau \leq \sigma \quad \sigma \leq \rho}{\tau \leq \rho}\ \mathit{Trans}$$

which is not well-moded: σ is an input in the premise, but unknown. So we have to design a set of rules that get by without the rule of transitivity. We write this new judgment as $\tau \sqsubseteq \sigma$. The idea is to eliminate transitivity and reflexivity and just have decomposition rules except for the primitive coercion from int to float.[1] We will not write the coercions explicitly for the

[1] In Assignment 6, a slightly different choice has been made to account for type variables which we ignore here.

sake of brevity.

$$\frac{}{\mathsf{int} \sqsubseteq \mathsf{float}}$$

$$\frac{}{\mathsf{int} \sqsubseteq \mathsf{int}} \qquad \frac{}{\mathsf{float} \sqsubseteq \mathsf{float}} \qquad \frac{}{\mathsf{bool} \sqsubseteq \mathsf{bool}}$$

$$\frac{\sigma_1 \sqsubseteq \tau_1 \quad \tau_2 \sqsubseteq \sigma_2}{\tau_1 \to \tau_2 \sqsubseteq \sigma_1 \to \sigma_2}$$

$$\frac{\tau_1 \sqsubseteq \sigma_1 \quad \tau_2 \sqsubseteq \sigma_2}{\tau_1 \times \tau_2 \sqsubseteq \sigma_1 \times \sigma_2} \qquad \frac{}{1 \sqsubseteq 1}$$

$$\frac{\tau_1 \sqsubseteq \sigma_1 \quad \tau_2 \sqsubseteq \sigma_2}{\tau_1 + \tau_2 \sqsubseteq \sigma_1 + \sigma_2} \qquad \frac{}{0 \sqsubseteq 0}$$

Note that these are well-moded with $\tau^+ \sqsubseteq \sigma^+$. We have ignored here universal, existential and recursive types: adding them requires some potentially difficult choices that we would like to avoid for now.

Now we need to show that the algorithmic formulation of subtyping ($\tau \sqsubseteq \sigma$) coincides with the original specification of subtyping ($\tau \leq \sigma$). We do this in several steps.

Lemma 1 (Soundness of algorithmic subtyping)
If $\tau \sqsubseteq \sigma$ then $\tau \leq \sigma$.

Proof: By straightforward induction on the structure of the given derivation. ■

Next we need two properties of algorithmic subtyping. Note that these arise from the attempt to prove the completeness of algorithmic subtyping, but must nonetheless be presented first.

Lemma 2 (Reflexivity and transitivity of algorithmic subtyping)

(i) $\tau \sqsubseteq \tau$ for any τ.

(ii) If $\tau \sqsubseteq \sigma$ and $\sigma \sqsubseteq \rho$ then $\tau \sqsubseteq \rho$.

Proof: For (i), by induction on the structure of τ.

For (ii), by simultaneous induction on the structure of the two given derivations $\mathcal{D}$ of $\tau \sqsubseteq \sigma$ and $\mathcal{E}$ of $\sigma \sqsubseteq \rho$. We show one representative cases; all others are similar or simpler.

Case: $\mathcal{D} = \dfrac{\sigma_1 \sqsubseteq \tau_1 \quad \tau_2 \sqsubseteq \sigma_2}{\tau_1 \to \tau_2 \sqsubseteq \sigma_1 \to \sigma_2}$ and $\mathcal{E} = \dfrac{\rho_1 \sqsubseteq \sigma_1 \quad \sigma_2 \sqsubseteq \rho_2}{\sigma_1 \to \sigma_2 \sqsubseteq \rho_1 \to \rho_2}$. Then

$\rho_1 \sqsubseteq \tau_1$	By i.h.
$\tau_2 \sqsubseteq \rho_2$	By i.h.
$\tau_1 \to \tau_2 \sqsubseteq \rho_1 \to \rho_2$	By rule

■

Now we are ready to prove the completeness of algorithmic subtyping.

Lemma 3 (Completeness of algorithmic subtyping)
If $\tau \leq \sigma$ then $\tau \sqsubseteq \sigma$.

Proof: By straightforward induction over the derivation of $\tau \leq \sigma$. For reflexivity, we apply Lemma 2, part (i). For transitivity we appeal to the induction hypothesis and apply Lemma 2, part (ii). In all other cases we just apply the induction hypothesis and then the corresponding algorithmic subtyping rule. ■

Summarizing the results above we obtain:

Theorem 4 (Correctness of algorithmic subtyping)
$\tau \leq \sigma$ if and only if $\tau \sqsubseteq \sigma$.