Lecture Notes on
Type Checking

15-312: Foundations of Programming Languages
Frank Pfenning

Lecture 17
October 23, 2003

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. In this lecture we first consider how to systematically design the language so that every expression has a unique type, and how this statement has to be modified when we consider subtyping. This kind of language will turn out to be impractical, so we consider a more relaxed notion of type checking, which is nonetheless quite a bit removed from the type inference offered by ML (which is left for another lecture).

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). This harkens back to an earlier lecture where we considered parsing as the bottom-up construction of a derivation. The requirement we put on the rules is that they be *mode correct*. We do not fully formalize this notion here, but only give a detailed description.

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.

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

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.

If we want to be pedantic, we can define the type equality judgment $\tau = \sigma$ by a single rule of reflexivity.

$$\frac{}{\tau = \tau} \; \mathit{Refl}$$

This is clearly mode correct for the mode $\tau^+ = \sigma^+$.

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. To be more precise, we refer to the mode $\Gamma^+ \vdash e^+ : \tau^-$ as *type synthesis*, because a given expression in a given context generates a type (if it has one). The stronger property we want to enforce (for now) is that of *unique type synthesis*, that is, each well-typed expression has a unique type. The proof of uniqueness if left as an exercise.

We now show a few rules, where each expression construct is annotated with enough types to guarantee mode correctness, but no more.

$$\frac{\Gamma \vdash e_1 : \tau_1}{\Gamma \vdash \mathtt{inl}(\tau_2, e_1) : \tau_1 + \tau_2} \quad \frac{\Gamma \vdash e_2 : \tau_2}{\Gamma \vdash \mathtt{inr}(\tau_1, e_2) : \tau_1 + \tau_2}$$

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

Note that we check that both branches of a `case`-expression synthesize the same type, and how the left and right injection need to include precisely the information that is not available from the expression we inject.

One can see from the sum types, that guaranteeing unique type synthesis could lead to quite verbose programs. The trickiest cases have to do with constructs that bind types. We consider existential types, but related observations apply to universal and recursive types. Recall the constructor rule from an earlier lecture on data abstraction:

$$\frac{\Gamma \vdash \sigma \ \mathsf{type} \quad \Gamma \vdash e : \{\sigma/t\}\tau}{\Gamma \vdash \mathtt{pack}(\sigma, e) : \exists t.\tau}$$

We apply our mode checking algorithm to see if we can read this rule as part of an algorithm. For this we assign the mode $\Gamma^+ \vdash \sigma^+ \ \mathsf{type}$ to the verification that types are well-formed.

1. **Assume** that Γ, e and σ are known.
2. **Show** that Γ and σ are known, which is true.
3. **Show** that Γ and e are known, and $\{\sigma/t\}\tau$ is free. The former holds, but the latter does not.

So we rewrite the rule:

$$\frac{\Gamma \vdash \sigma \ \mathsf{type} \quad \Gamma \vdash e : \tau' \quad \tau' = \{\sigma/t\}\tau}{\Gamma \vdash \mathtt{pack}(\sigma, e) : \exists t.\tau}$$

Now we can proceed one step further, but we still don't know τ, and we cannot determine it from the constraints given here. For example $\mathtt{pack}(\mathsf{int}, 3)$:

$\exists t.t$ but also $\texttt{pack}(\mathsf{int}, 3) : \exists t.\mathsf{int}$. Concretely, $t.\tau$ is unknown when we reach the equality test.

In the end this means we need to put not only σ but also $t.\tau$ into the syntax of a `pack` expression.

$$\frac{\Gamma \vdash \sigma\ \mathsf{type} \quad \Gamma \vdash e : \tau' \quad \tau' = \{\sigma/t\}\tau}{\Gamma \vdash \texttt{pack}(\sigma, t.\tau, e) : \exists t.\tau}$$

At this point everything is well-moded, because $t.\tau$ and σ determine $\{\sigma/t\}\tau$. But it has become quite verbose because $t.\tau$ can be large. If we have nested existentials of the form $\exists t.\exists t'.\tau$, this is particularly troublesome because τ must be repeated twice: once in the type of outer `pack` expression, then in the inner `pack` expression.

Before describing a general solution to this problem, we consider how to add subtyping. Assume we have $\mathsf{int} \leq \mathsf{float}$, reflexivity, transitivity, and the usual co- and contravariant rules for type constructors. Recall also the rule of subsumption:

$$\frac{\Gamma \vdash e : \tau \quad \tau \leq \sigma}{\Gamma \vdash e : \sigma}$$

It should be immediately clear that an expression cannot possibly synthesize a unique type, because $3 : \mathsf{int}$ but also $3 : \mathsf{float}$. Instead, we have to design a system where an expression e in a context Γ synthesizes a *principal type*.

> **Principal Type.** We say τ is the *principal type* of e in context Γ if $\Gamma \vdash e : \tau$ and for every type σ such that $\Gamma \vdash e : \sigma$ we have $\tau \leq \sigma$.

The important property of a principal type τ of an expression e is that we can recover all other types of e as supertypes of τ.

Some languages have the property that they satisfy this principle: every expression does indeed have a principal type. If that is the case, the goal is to find a formulation of the typing rules such that every expression synthesizes its principal type, or fails if no type exists. If we have, in addition, a means for checking the subtype relation $\tau \leq \sigma$, then we can effectively test if $\Gamma \vdash e : \sigma$ for any given Γ, e and σ: we compute the principal type τ for e in Γ and then check if $\tau \leq \sigma$. An alternative will be discussed in a future lecture.

Let us first tackle the problem of deciding of $\tau \leq \sigma$, assuming both τ and σ are inputs. Unfortunately, the rule of transitivity

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

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. We will not write the coercions explicitly here for the 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 $\tau \sqsubseteq \sigma$.

Now we can write out the rules that synthesize principal types. We write this new judgment as $\Gamma \vdash e \uparrow \tau$ with mode $\Gamma^+ \vdash e^+ \uparrow \tau^-$. The idea is to eliminate the rule of subsumption entirely. To compensate, we replace uses of the equality judgment $\tau = \sigma$ by appropriate uses of algorithmic subtyping. It is not obvious at this point that this should work, but we will see in Theorem 7 that it does. We only show a selection of the rules here.

$$\frac{x{:}\tau \text{ in } \Gamma}{\Gamma \vdash x \uparrow \tau}\ \textit{Var}$$

$$\frac{\Gamma, x{:}\tau_1 \vdash e \uparrow \tau_2}{\Gamma \vdash \texttt{fn}(\tau_1, x.e) \uparrow \tau_1 \to \tau_2}\ \textit{FnTyp}$$

$$\frac{\Gamma \vdash e_1 \uparrow \tau_2 \to \tau \quad \Gamma \vdash e_2 \uparrow \tau_2' \quad \tau_2' \sqsubseteq \tau_2}{\Gamma \vdash \texttt{apply}(e_1, e_2) \uparrow \tau}\ \textit{AppTyp}$$

$$\frac{\Gamma \vdash e_1 \uparrow \tau_1 \quad \Gamma \vdash e_2 \uparrow \tau_2}{\Gamma \vdash \texttt{pair}(e_1, e_2) \uparrow \tau_1 \times \tau_2}$$

$$\frac{\Gamma \vdash e \uparrow \tau_1 \times \tau_2}{\Gamma \vdash \texttt{fst}(e) \uparrow \tau_1} \qquad \frac{\Gamma \vdash e \uparrow \tau_1 \times \tau_2}{\Gamma \vdash \texttt{snd}(e) \uparrow \tau_2}$$

For sums, we have no problem with the injections, because they carry some type information.

$$\frac{\Gamma \vdash e_1 \uparrow \tau_1}{\Gamma \vdash \texttt{inl}(\tau_2, e_1) \uparrow \tau_1 + \tau_2} \qquad \frac{\Gamma \vdash e_2 \uparrow \tau_2}{\Gamma \vdash \texttt{inr}(\tau_1, e_2) \uparrow \tau_1 + \tau_2}$$

However, the case constructs creates a problem, because the two branches may synthesize principal types σ_1 and σ_2, but they may not be the same or even comparable (neither $\sigma_1 \leq \sigma_2$ or $\sigma_2 \leq \sigma_1$ may hold).

$$\frac{\Gamma \vdash e \uparrow \tau_1 + \tau_2 \quad \Gamma, x_1{:}\tau_1 \vdash e_1 \uparrow \sigma_1 \quad \Gamma, x_2{:}\tau_2 \vdash e_2 \uparrow \sigma_2}{\Gamma \vdash \texttt{case}(e, x_1.e_1, x_2.e_2) \uparrow \sigma}\ \textbf{?}$$

So the question is how to compute σ from σ_1 or σ_2 or fail. What we need is the smallest upper bound of σ_1 and σ_2. In other words, we need a type σ such that $\sigma_1 \leq \sigma$, $\sigma_2 \leq \sigma$, and for any other ρ such such $\sigma_1 \leq \rho$ and $\sigma_2 \leq \rho$ we have $\rho \leq \sigma$.

Fortunately, this is not difficult in our particular system. In real languages, however, this can be a real problem. For example, verification algorithms for Java bytecode have to deal with this problem, with a somewhat ad hoc solution.[1] The general solution is to introduce *intersection types*, something we may discuss in a future lecture.

[1] If I remember this correctly.

We define the computation of the least upper bound as a 3-place judgment, $\sigma_1 \sqcup \sigma_2 \Rightarrow \sigma$. It is defined by the following rules, which have mode $\sigma_1^+ \sqcup \sigma_2^+ \Rightarrow \sigma^-$. Unfortunately, the contra-variance of the function type requires us to also define the greatest lower bound of two types, written $\sigma_1 \sqcap \sigma_2 \Rightarrow \sigma$, with mode $\sigma_1^+ \sqcap \sigma_2^+ \Rightarrow \sigma^-$.

$$\frac{}{\mathsf{int} \sqcup \mathsf{int} \Rightarrow \mathsf{int}} \quad \frac{}{\mathsf{int} \sqcup \mathsf{float} \Rightarrow \mathsf{float}} \quad \frac{}{\mathsf{float} \sqcup \mathsf{int} \Rightarrow \mathsf{float}} \quad \frac{}{\mathsf{float} \sqcup \mathsf{float} \Rightarrow \mathsf{float}}$$

$$\frac{}{\mathsf{bool} \sqcup \mathsf{bool} \Rightarrow \mathsf{bool}} \quad \frac{}{1 \sqcup 1 \Rightarrow 1} \quad \frac{}{0 \sqcup 0 \Rightarrow 0}$$

$$\frac{\tau_1 \sqcap \sigma_1 \Rightarrow \rho_1 \quad \tau_2 \sqcup \sigma_2 \Rightarrow \rho_2}{\tau_1 \to \tau_2 \sqcup \sigma_1 \to \sigma_2 \Rightarrow \rho_1 \to \rho_2}$$

$$\frac{\tau_1 \sqcup \sigma_1 \Rightarrow \rho_1 \quad \tau_2 \sqcup \sigma_2 \Rightarrow \rho_2}{\tau_1 \times \tau_2 \sqcup \sigma_1 \times \sigma_2 \Rightarrow \rho_1 \times \rho_2}$$

$$\frac{\tau_1 \sqcup \sigma_1 \Rightarrow \rho_1 \quad \tau_2 \sqcup \sigma_2 \Rightarrow \rho_2}{\tau_1 + \tau_2 \sqcup \sigma_1 + \sigma_2 \Rightarrow \rho_1 + \rho_2}$$

$$\frac{}{\mathsf{int} \sqcap \mathsf{int} \Rightarrow \mathsf{int}} \quad \frac{}{\mathsf{int} \sqcap \mathsf{float} \Rightarrow \mathsf{int}} \quad \frac{}{\mathsf{float} \sqcap \mathsf{int} \Rightarrow \mathsf{int}} \quad \frac{}{\mathsf{float} \sqcap \mathsf{float} \Rightarrow \mathsf{float}}$$

$$\frac{}{\mathsf{bool} \sqcap \mathsf{bool} \Rightarrow \mathsf{bool}} \quad \frac{}{1 \sqcap 1 \Rightarrow 1} \quad \frac{}{0 \sqcap 0 \Rightarrow 0}$$

$$\frac{\tau_1 \sqcup \sigma_1 \Rightarrow \rho_1 \quad \tau_2 \sqcap \sigma_2 \Rightarrow \rho_2}{\tau_1 \to \tau_2 \sqcap \sigma_1 \to \sigma_2 \Rightarrow \rho_1 \to \rho_2}$$

$$\frac{\tau_1 \sqcap \sigma_1 \Rightarrow \rho_1 \quad \tau_2 \sqcap \sigma_2 \Rightarrow \rho_2}{\tau_1 \times \tau_2 \sqcap \sigma_1 \times \sigma_2 \Rightarrow \rho_1 \times \rho_2}$$

$$\frac{\tau_1 \sqcap \sigma_1 \Rightarrow \rho_1 \quad \tau_2 \sqcap \sigma_2 \Rightarrow \rho_2}{\tau_1 + \tau_2 \sqcap \sigma_1 + \sigma_2 \Rightarrow \rho_1 + \rho_2}$$

It is straightforward, but tedious to verify that these judgments do indeed define the least upper bound and greatest lower bound of two types and fail if no bound exists. Then the rule for `case`-expressions becomes:

$$\frac{\Gamma \vdash e \uparrow \tau_1 + \tau_2 \quad \Gamma, x_1{:}\tau_1 \vdash e_1 \uparrow \sigma_1 \quad \Gamma, x_2{:}\tau_2 \vdash e_2 \uparrow \sigma_2 \quad \sigma_1 \sqcup \sigma_2 \Rightarrow \sigma}{\Gamma \vdash \mathtt{case}(e, x_1.e_1, x_2.e_2) \uparrow \sigma}$$

Now we can formulate the soundness and completeness theorem for the synthesis of principal types.

Lemma 5 (Soundess of principal type synthesis)
If $\Gamma \vdash e \uparrow \tau$ *then* $\Gamma \vdash e : \tau$

Proof: By straightforward induction on the given derivation, using the soundness of algorithmic subtyping and the fact that if $\sigma_1 \sqcap \sigma_2 \Rightarrow \sigma$ then $\sigma_1 \leq \sigma$ and $\sigma_2 \leq \sigma$. ■

The completeness is more difficult to prove.

Lemma 6 (Completeness of principal type synthesis)
If $\Gamma \vdash e : \tau$ *then there exists a* σ *such that* $\sigma \sqsubseteq \tau$ *and* $\Gamma \vdash e \uparrow \sigma$*.*

Proof: By induction on the derivation of $\Gamma \vdash e : \tau$, using previous lemmas and inversion on algorithmic subtyping. We show only three cases.

Case: $\dfrac{\Gamma \vdash e : \tau' \quad \tau' \leq \tau}{\Gamma \vdash e : \tau}$.

$\Gamma \vdash e \uparrow \sigma'$ and $\sigma' \sqsubseteq \tau'$ for some σ'	By i.h.
$\tau' \sqsubseteq \tau$	By completeness of $\sqsubseteq$ (Lemma 3)
$\sigma' \sqsubseteq \tau$	By transitivity of $\sqsubseteq$ (Lemma 2(ii))

Case: $\dfrac{\Gamma \vdash e_1 : \tau_2 \to \tau \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash \mathtt{apply}(e_1, e_2) : \tau}$.

$\Gamma \vdash e_1 \uparrow \sigma_1$ and $\sigma_1 \sqsubseteq \tau_2 \to \tau$ for some σ_1	By i.h.
$\sigma_1 = \sigma_2' \to \sigma$ for some σ_2' and σ where $\tau_2 \leq \sigma_2'$ and $\sigma \leq \tau$	By inversion
$\Gamma \vdash e_1 \uparrow \sigma_2' \to \sigma$	By equality
$\Gamma \vdash e_2 \uparrow \sigma_2$ and $\sigma_2 \sqsubseteq \tau_2$	By i.h.
$\sigma_2 \sqsubseteq \sigma_2'$	By transitivity of $\sqsubseteq$ (Lemma 2(ii))
$\Gamma \vdash \mathtt{apply}(e_1, e_2) \uparrow \sigma$	By rule
$\sigma \leq \tau$	Copied from above

Case: $\dfrac{\Gamma, x{:}\tau_1 \vdash e_1 : \tau_2}{\Gamma \vdash \mathtt{fn}(\tau_1, x.e_2) : \tau_1 \to \tau_2}$.

$\Gamma, x{:}\tau_1 \vdash e_1 \uparrow \sigma_2$ and $\sigma_2 \sqsubseteq \tau_2$ for some σ_2	By i.h.
$\Gamma \vdash \mathtt{fn}(\tau_1, x.e_2) \uparrow \tau_1 \to \sigma_2$	By rule
$\tau_1 \sqsubseteq \tau_1$	By reflexivity of $\sqsubseteq$ (Lemma 2(i))
$\tau_1 \to \sigma_2 \sqsubseteq \tau_1 \to \tau_2$	By rule

■

Now we can put theses together into a correctness theorem for principal type synthesis.

Theorem 7 (Correctness of principal type synthesis)

(i) *If* $\Gamma \vdash e \uparrow \tau$ *then* $\Gamma \vdash e : \tau$.

(ii) *If* $\Gamma \vdash e : \tau$ *then* $\Gamma \vdash e \uparrow \sigma$ *for some* σ *with* $\sigma \leq \tau$.

Proof: From the previous two lemmas, using soundess of algorithmic subtyping in part (ii). ■