15-312 Foundations of Programming Languages
Recitation 10: Type Checking and Type Inference

Daniel Spoonhower
spoons+@cs

November 5, 2003

1 Type Checking

Today, we will consider a number of examples related to type checking, mostly concerning those constructs for introducing and eliminating expressions of sum type. To warm up, review the cases for ordinary type synthesis:

$$\frac{\Gamma \vdash \mathtt{e}_1 : \tau_1}{\Gamma \vdash \mathtt{inl}(\tau_2, e_1) : \tau_1 + \tau_2}$$

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

What happens when we add subtyping to this system?

Speaking of subtyping, in the course of assigning types in the context of subtyping we came upon a problem, specifically, with the transitivity of subtyping.

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

What mode would we like to assign to the subtyping judgment? Why doesn't that work in this case?

Instead of continuing with our original formulation of subtyping, we instead move forward with *algorithmic subtyping*, a less ambitious variation. Recall that once we defined algorithmic subtyping, we proved the following lemma.

Lemma 1 (Transitivity of Algorithmic Subtyping). *If $\tau \sqsubseteq \sigma$ and $\sigma \sqsubseteq \rho$ then $\tau \sqsubseteq \rho$.*

What's the difference between this statement and the inference rule above? If we specifically avoided adding a rule for transitivity to algorithmic subtyping, why did we immediately prove that it is true?

Since we can prove that transitivity holds for algorithmic subtyping, adding such a rule would be redundant, but before we continue, it's worth taking a moment to understand *how* this rule would be redundant. In some cases, a rule

is redundant simply because it is the composition of existing rules; here we say that the rule is *derivable*. In other cases, there is no such derivation, but we can, nonetheless prove the following: that the conclusion is derivable under no premises whenever the premises are derivable under no premises. In this latter case, we say that a rule is *admissible*.

Here's another (albeit simple) example using a standard inductive definition of natural numbers.

$$\frac{n\ \mathsf{nat}}{\mathsf{succ}(\mathsf{succ}(n))\ \mathsf{nat}} \qquad \frac{\mathsf{succ}(n)\ \mathsf{nat}}{n\ \mathsf{nat}}$$

Notice that the statement on the left is derivable: it can be proved using two applications of the successor rule. The rule on the right, however, is not derivable. (Convince yourself that this is true.) It is, however, admissible: the bottom judgment is derivable under no premises whenever the top is derivable under no premises. (Prove this!)

2 Type Inference

Back in our discussion of sums, note that in the bidirectional case, the notion of a least upper bound (as used in type checking above) is unnecessary (since we must only *check* each expression against σ).

$$\frac{\Gamma \vdash \mathtt{e}_1 \uparrow \tau_1}{\Gamma \vdash \mathtt{inl}(\tau_2, e_1) \uparrow \tau_1 + \tau_2}$$

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

In bidirectional typing, however, we still had to include the type of the other half of the sum (e.g. $\mathtt{inl}(\tau_2, e_1)$). In full type inference this is not necessary. Recall our judgment for full type inference, shown here with modes attached.

$$\Gamma^+ \vdash e^+ \Longrightarrow \tau^- \mid C^-$$

In the full inference system, we can write the injection expression without any type.

$$\frac{\Gamma \vdash e \Longrightarrow \tau_1 \mid C \quad (\alpha_2 \text{ new})}{\Gamma \vdash \mathtt{inl}(e) \Longrightarrow \tau_1 + \alpha_2 \mid C}$$

$$\frac{\Gamma \vdash e \Longrightarrow \tau \mid C \quad (\alpha, \alpha_2 \text{ new}) \quad \Gamma, x_1 : \alpha_1 \vdash e_1 \Longrightarrow \sigma \mid C_1 \quad \Gamma, x_2 : \alpha_2 \vdash e_2 \Longrightarrow \sigma' \mid C_2}{\Gamma \vdash \mathtt{case}(e, x_1.e_1, x_2.e_2) \Longrightarrow \sigma \mid C, C_1, C_2, \tau \doteq \alpha_1 + \alpha_2, \sigma \doteq \sigma'}$$

In order to verify your understanding of type inference, show how our inference system derives a set of constraints and then solves them to find a type for the following expression:

```
case inl 7
  of inl(x) => 3 + x
   | inr(f) => f 4
```

2.1 The Value Restriction

In lecture, we noted some of the difficulties related to using `let` constructs in a language with full type inference. Does that discussion shed any light on the following type error?

```
- val empty = ref nil;
stdIn:22.2-22.21 Warning: type vars not generalized because of
   value restriction are instantiated to dummy types (X1,X2,...)
val empty = ref [] : ?.X1 list ref
```