

Comp 411 Notes - Fri, Feb 28 2005

More About Subtyping

Scribe: Seth James Nielson

Previously, we introduced a new relation into our type system called *subtyping* denoted by $<$. Additionally, we added a new typing rule to make use of subtyping in type-checking. Unfortunately, this system is impossible to implement as currently defined.

In today's class we will introduce *algorithmic subtyping* and *algorithmic typing* as a solution to this problem.

In the book, subtyping and typing are denoted as:

$$\begin{array}{l} S <: T \\ \Gamma t : T \end{array}$$

and algorithmic subtyping and algorithmic typing as:

$$\begin{array}{l} \mapsto S <: T \\ \Gamma \mapsto t : T \end{array}$$

In class, we denoted subtyping with $<$ and algorithmic subtyping with $<:.$ Similarly, we denoted typing with \vdash and algorithmic typing \vdash_A . We also used $<:_A$ to represent the type-system when both algorithmic subtyping and algorithmic typing are combined and $<$ when neither are present.

In today's class we show that $<$ and $<:_A$ are equivalent but that $<:_A$ is formed in a way that is more convenient for implementation.

1 Algorithmic Typing

The following statement is true:

$$\Gamma \vdash_{<_{App}} e : t' \iff \Gamma \vdash_{<} e : t'$$

The only part of typing that needs any work at all is the application rule which has become a macro for combining three steps into one.

$$\frac{\Gamma \vdash_A t_1 : T_1 \quad T_1 = T_1 1 \rightarrow T_1 2 \quad \Gamma \vdash_A t_2 : T_2 \quad T_2 <: T_1 1}{\Gamma \vdash_A t_1 t_2 : T_1 2} \quad \mathbf{TA-APP}$$

2 Algorithmic Subtyping

Current (declarative) subtyping rules are:

$$\frac{t_2 < s_2 \quad s_i < t_i}{t_1 \rightarrow t_2 < s_1 \rightarrow s_2} \quad \mathbf{S-ARROW}$$

$$\frac{t_1 < t_3 \quad t_3 < t_2}{t_1 < t_2} \quad \mathbf{S-TRANS}$$

$$\frac{\{t_i < s_i\}}{\{l_i : t_i\} < \{l_i : s_i\}} \quad \mathbf{S-RCDDEPTH}$$

$$\frac{}{\{l_i : t_i, l_j : t_j\} < \{l_i : t_i\}} \quad \mathbf{S-RCDWIDTH}$$

We can combine the depth and width rules, subsequently eliminate transitivity and introduce a new reflexivity. These constitute the algorithmic subtyping rules.

$$\frac{t_2 <: s_2 \quad s_i <: t_i}{t_1 \rightarrow t_2 <: s_1 \rightarrow s_2} \quad \mathbf{SA-ARROW}$$

$$\frac{\{t_i <: s_i\}^i}{\{l_i : t_i, l_j : t_j\} <: \{l_i : s_i\}} \quad \mathbf{SA-RCDs}$$

If $S < T$ from the subtyping rules including S-RCDDEPTH and S-RCDWIDTH then it can be derived using SA-RCD (the book states that the proof is “straightforward induction on derivations”).

By eliminating the depth and width rules, we can also eliminate the S-TRANS rule (this rule was used to “paste” together subtyping derivations using combinations of these rules). The proof that S-TRANS is no longer needed is homework problem 16.1.2 part 2.

We can also prove that $S < S$ can be derived for every type S without using S-REFL. This is homework problem 16.1. part 1.

We have shown that Algorithmic subtyping is equivalent to subtyping. That is,

$$t_1 < t_2 \iff t_1 <: t_2$$

3 Post-Class Discussion

From Dan Smith:

Ignoring the distinction between $<_{App}$ and $<:_{App}$ (it’s not important here), you say

$$P \rightarrow Q, \text{ but not } Q \rightarrow P$$

and then

$$\sim P \rightarrow \sim Q, \text{ but not } \sim Q \rightarrow \sim P$$

As I understand it, the only difference between the tuples in the typing relation $<:_{App}$ and the tuples in the typing relation $<$ is this: whenever (Γ, e, T) is in $<:_{App}$, (Γ, e, T) is in $<$, but for every T' which is a supertype of T , (Γ, e, T') is also in $<$. The important point of the whole discussion is that an expression has some type under $<:_{App}$ iff it has a type under $<$. So you can write a type checker that works based on $<:_{App}$ and it will recognize as well-typed exactly those expressions that are well-typed under $<$. It may not come up with *all* the types that the expression could have under $<$, but it will come up with *something*.

And from Professor Taha:

Dan makes a really good point here that was, technically, not made in class. In class we focused on showing that one type system allows you to type a term iff and only if the other type system allows you to assign the term a *related* type. But this statement implies something more abstract, which is that one type system types a term iff the other type system also types the term for *some* type. The first statement clearly gives us more information (specifically, about the relation of the types in both systems), but the second, even though it is more abstract, says that both type systems type *exactly* the same set of terms. If we view “a type system” as a set of terms (paired with an environment for their free variables), then this second correspondance means that both systems characterize exactly the same type system.

4 Next Class Clarification

$$\begin{aligned} t_1 < t_2 &\iff t1 <: t2 \\ (\exists t'. \Gamma \vdash e : t' &\iff (\exists t''. \Gamma \vdash_A e : t'') \\ (\exists t'. 0 \vdash e : t' &\iff (\exists t''. 0 \vdash_A e : t'') \end{aligned}$$

A type system characterizes a set of expressions (closed, or pairs of open expressions and environments). The two type systems $<$ and $<:_{App}$ have the same set of terms.