Subtyping in the Lambda-Calculus

Karl Schnaitter

April 28, 2005

1 Introduction

Subtyping is a common feature in modern programming languages. Type systems will expect values of certain types in expressions, function arguments, or storage. There are common situations where a type `T` is expected, but a value of some other type `S` would also make sense. It is adventageous to allow the flexibility to accept expressions of type `S` in such situations. This allows for more expressiveness in the programming language. Functions can be written more generally to accept more than one type as an argument. In functions that accept records as input, this also provides some abstraction because a function does not need to know every field of the record. The function only needs to specify the fields it needs in the input record, and any additional fields may be present when the function is called. Consider the following example of a function that selects a field `a` from its input.

λ`r:{a:Nat}.r.a`

Without subtyping, this function can only operate on arguments of type {`a:Nat`}. However, the function is well-behaved on inputs with type {`a:Nat, b:Bool`}, {`a:Nat, x:Nat, y:Nat`}, etc. With subtyping, we are allowed to pass these types of values to the function. Thus one function may be applied to many types.

2 Subtyping Rules

Informally, `S` is a subtype of `T` if, for all contexts that expect a value of type `T`, a value of type `S` may be supplied instead. We use `S <: T` to denote that `S` is a subtype of `T`. Two special terms will be added, called `Top` and `Bot`. The `Top` term is a supertype of all types, and `Bot` is a subtype of all types. Now we can define some of the inference rules used to find subtyping relationships.

$$\frac{}{\texttt{S <: S}}\ \text{(S-Refl)} \qquad \frac{\texttt{S <: U} \quad \texttt{U <: T}}{\texttt{S <: T}}\ \text{(S-Trans)}$$

$$\frac{}{\texttt{S <: Top}}\ \text{(S-Top)} \qquad \frac{}{\texttt{Bot <: T}}\ \text{(S-Bot)}$$

These rules are straightforward and intuitive. It would not cause any serious problems if these were our only subtyping rules, because the type system would still be sound. However, these rules do not offer much power yet. We need rules for functions, records, and references.

2.1 Function Rules

Suppose we have two functions, s and t, with types $\mathtt{S}_1 \to \mathtt{S}_2$ and $\mathtt{T}_1 \to \mathtt{T}_2$, respectively. When is it safe to substitute t with s? First, s must accept *at least* all the values that t accepts. This intuitively means that $\mathtt{T}_1 \ \mathtt{<:}\ \mathtt{S}_1$. Second, s cannot return values that are not contained in the return type of t. This requirement is stated as $\mathtt{S}_2 \ \mathtt{<:}\ \mathtt{T}_2$. We can now define the subtyping rule for functions.

$$\frac{\mathtt{T}_1 \ \mathtt{<:}\ \mathtt{S}_1 \qquad \mathtt{S}_2 \ \mathtt{<:}\ \mathtt{T}_2}{\mathtt{S}_1 \to \mathtt{S}_2 \ \mathtt{<:}\ \mathtt{T}_1 \to \mathtt{T}_2} \text{ (S-Arrow)}$$

It may be surprising that the first antecedent requires that $\mathtt{T}_1$ is a subtype of $\mathtt{S}_1$, because this relationship is in the reverse direction of the final result. This kind of rule is called *contravariant*. Since s is an acceptor of $\mathtt{S}_1$ values, we might call it a *sink* for $\mathtt{S}_1$. Contravariant rules normally work with sink objects. Conversely, s produces $\mathtt{S}_2$ values, so it is also a *source* for $\mathtt{S}_2$. The direction of the antecedent $\mathtt{S}_2 \ \mathtt{<:}\ \mathtt{T}_2$ is in the same direction as the conclusion, so we say this rule is *covariant*. Covariant rules normally work with source objects.

If we consider a type language with only `Top`, `Bot`, and arrow types, then we can think of a lattice that illustrates the subtyping relation as a partial order. In our class discussion, we were essentially able to describe an infinite decreasing chain, as follows:

$$\begin{array}{l}
\mathtt{Top}, \\
\mathtt{Bot} \to \mathtt{Top},\ \mathtt{Bot} \to (\mathtt{Bot} \to \mathtt{Top}),\ \mathtt{Bot} \to (\mathtt{Bot} \to (\mathtt{Bot} \to \mathtt{Top})),\ \ldots \\
\mathtt{Top} \to \mathtt{Top} \\
\ldots,\ \mathtt{Top} \to (\mathtt{Top} \to (\mathtt{Top} \to \mathtt{Bot})),\ \mathtt{Top} \to (\mathtt{Top} \to \mathtt{Bot}),\ \mathtt{Top} \to \mathtt{Bot}, \\
\mathtt{Bot}
\end{array}$$

The second and fourth rows are infinite sequences, and the type $\mathtt{Top} \to \mathtt{Top}$ on the third row is between the two infinite sequences. We could alternatively have $\mathtt{Bot} \to \mathtt{Bot}$ on the third row, but both types cannot be in the chain since $\mathtt{Top} \to \mathtt{Top}$ and $\mathtt{Bot} \to \mathtt{Bot}$ are not comparable.

2.2 Rules for Records and References

Here are the subtyping rules for records:

$$\frac{}{\{\mathtt{l}_i\mathtt{:}\ \mathtt{T}_i{}^{i\in 1..n+k}\} \ \mathtt{<:}\ \{\mathtt{l}_i\mathtt{:}\ \mathtt{T}_i{}^{i\in 1..n}\}} \text{ (S-RcdWidth)} \qquad \frac{\text{for each } i,\ \ \mathtt{S}_i \ \mathtt{<:}\ \mathtt{T}_i}{\{\mathtt{l}_i\mathtt{:}\ \mathtt{S}_i{}^{i\in 1..n}\} \ \mathtt{<:}\ \{\mathtt{l}_i\mathtt{:}\ \mathtt{T}_i{}^{i\in 1..n}\}} \text{ (S-RcdDepth)}$$

$$\frac{\{\mathtt{k}_j\mathtt{:}\ \mathtt{S}_j{}^{j\in 1..n}\} \text{ is a permutation of } \{\mathtt{l}_i\mathtt{:}\ \mathtt{T}_i{}^{i\in 1..n}\}}{\{\mathtt{k}_j\mathtt{:}\ \mathtt{S}_j{}^{j\in 1..n}\} \ \mathtt{<:}\ \{\mathtt{l}_i\mathtt{:}\ \mathtt{T}_i{}^{i\in 1..n}\}} \text{ (S-RcdPerm)}$$

There are a few interesting things about these rules. The S-RcdPerm rule shows that

$$\begin{array}{rl}
 & \{\mathtt{a} : \mathtt{Nat}, \mathtt{b} : \mathtt{Bool}\} \ \mathtt{<:}\ \{\mathtt{b} : \mathtt{Bool}, \mathtt{a} : \mathtt{Nat}\} \\
\text{and} & \{\mathtt{b} : \mathtt{Bool}, \mathtt{a} : \mathtt{Nat}\} \ \mathtt{<:}\ \{\mathtt{a} : \mathtt{Nat}, \mathtt{b} : \mathtt{Bool}\}
\end{array}$$

This lack of antisymmetry shows that the subtyping relation is *not* a partial order, unless we formally consider these types to be equal. Also note that S-RcdDepth does not have a contravariant rule. This is because records are immutable. On the other hand, references can change their stored values, so references are both sources and sinks. The rule for references follows:

$$\frac{\mathtt{S} \ \mathtt{<:}\ \mathtt{T} \qquad \mathtt{T} \ \mathtt{<:}\ \mathtt{S}}{\mathtt{Ref\ S} \ \mathtt{<:}\ \mathtt{Ref\ T}} \text{ (S-Ref)}$$

2.3 Bringing it together

The subtyping relation has been defined. There needs to be a way of integrating the subtyping rules into the type system. Essentially, we need to allow a typed term to typecheck as any supertype. This idea is called *subsumption*. The following rule is added to the type rules:

$$\frac{\Gamma \vdash \texttt{t} : \texttt{S} \qquad \texttt{S} <: \texttt{T}}{\Gamma \vdash \texttt{t} : \texttt{T}} \text{ (T-Sub)}$$

This is how the type system is officially extended. When the type system is made more flexible by allowing more terms to be well-typed, there is an inherent danger. If we allow a bad program to be well-typed, it might get stuck. In other words, the "progress" property of our language could be lost. In order to check for mistakes in the subtyping rules, the progress theorem needs to be proven again.

3 The `Top` and `Bot` Types

The `Top` type is straightforward to implement in a type checker, and it is easy to understand as a set containing all values. The `Bot` type is a little more interesting. First, we need to realize that the `Bot` type does not have any values in it. If a value of type `Bot` exists, then it could be used as a function or a record, so progress would be lost. However, there can be terms with type `Bot`. This is useful for errors. Since errors do not have values, they should be allowed in any context. This is possible if errors typecheck as `Bot`.

Suppose we know that a term $\texttt{t}_1 \to \texttt{t}_2$ is well-typed. If there is no `Bot` type, then we can be sure that $\texttt{t}_1$ has an arrow type. This is often convenient in algorithms and proofs. With the introduction of `Bot`, we would need to consider the case that $\texttt{t}_1$ has type `Bot`. This adds some complexity when reasoning about the type system.

4 Ascription and Casting

In Chapter 11 of the textbook, we were introduced to ascription. In the simply typed λ-calculus, ascription does not have an effect on how the program runs. It is a way to double check that a term has the type that the programmer expects. It is useful because ascription can lead to better documentation, and type errors can be easier to understand. With the introduction of subtyping, now ascription can become more powerful.

Suppose that $\Gamma \vdash \texttt{t} : \texttt{S}$ and $\texttt{S} <: \texttt{T}$. Then, without any new extensions, we get $\Gamma \vdash \texttt{t as T} : \texttt{T}$. This can be derived by applying the original ascription rule and subsumption. This is called *up-casting* because the type of a term is being reassigned to a supertype. This can be used to hide some functionality of the term. For example, the ascription could restrict access to fields of a record by ascribing a "narrower" record type with fewer fields. Similarly, ascription could be used on a function to disallow some argument types. Up-casting does not introduce any run-time problems.

On the other hand, if $\texttt{S} <: \texttt{T}$ and $\Gamma \vdash \texttt{t} : \texttt{T}$, then `t as S` is called a *down-cast*. If we wish to allow down-casts, we need a new rule:

$$\frac{\Gamma \vdash \texttt{t}_1 : \texttt{S}}{\Gamma \vdash \texttt{t}_1 \texttt{ as T} : \texttt{T}} \text{ (T-DownCast)}$$

This allows any well-typed term to be interpreted as any other type. Clearly, the progress property of the language is lost if we stop here. A term such as `(true as {Unit → Unit}) unit` will get

stuck in the normal form `true unit`. The type preservation property is also lost because a term of type `Unit` $\to$ `Unit` evaluated to a term of type `Bool`. We can make things better by checking casts of values at run-time. The original ascription evaluation rule should be replaced with this rule:

$$\frac{\vdash \mathtt{v}_1 : \mathtt{T}}{\mathtt{v}_1 \ \mathtt{as} \ \mathtt{T} \to \mathtt{v}_1} \ \text{(E-DownCast)}$$

This rule brings back type preservation, but progress still does not hold. The program mentioned earlier will get stuck without making any evaluation steps. We need a rule to use when the antecedent in E-DownCast fails. Assuming we have errors in the language, this is easy:

$$\frac{\nvdash \mathtt{v}_1 : \mathtt{T}}{\mathtt{v}_1 \ \mathtt{as} \ \mathtt{T} \to \mathtt{error}} \ \text{(E-CastError)}$$

In this way, progress and preservation can hold with down-casting.

5 Algorithmic Subtyping

The subtyping rules allow for a more flexible type system. The implementation of this new type system is not as straightforward as the simple type system. The first problem is that the type rules are not syntax-directed. The subsumption rule applies to every term. This means that every syntactic form can fit two type rules, so the structure of a term does not determine what rule should be applied.

The second problem is with the subtype relation. The rules to derive subtypes are not syntax-directed either. It is particularly hard to think of a way to algorithmically apply the S-Trans rule for subtype transitivity. If you are trying to prove that `S <: T`, then it is not clear whether there exists a third intermediate type `U` to use in the transitivity rule.

To solve these problems, we will consider a subset of the language that only uses the types `Top`, `Unit`, and arrow types. We can solve the first problem by making subsumption more restricted. The original rule allows subsumption to be applied freely to any term. We need to be more specific about where to apply subsumption. The rules can be made syntax-directed again, so a natural algorithm can be written. It turns out that the only time we need subsumption is in function applications where the argument may need to be assigned a supertype. The type rule for an application is replaced with the following rule:

$$\frac{\Gamma \vdash \mathtt{t}_1 : \mathtt{T} \to \mathtt{R} \qquad \Gamma \vdash \mathtt{t}_2 : \mathtt{S} \qquad \mathtt{S} <: \mathtt{T}}{\Gamma \vdash \mathtt{t}_1 \mathtt{t}_2 : \mathtt{R}}$$

It can be shown that a term is well-typed in the original type system if and only if the term is well-typed with this change. In addition, this rule allows only one unique type to be assigned to a well-typed term. This type is as specific as possible. In other words, this rule gives every well-typed term a *principle type*. In contrast, with the old system, we could have vague type judgements such as $\vdash$ `true` : `Top`. These judgements are not particularly useful, and they are eliminated when we have principle types.

The second problem deals with deciding whether `S <: T` holds. This can be solved without changing the type system. In order to solve the problem, we need to avoid using subtype transitivity. Then we can create an algorithm that has cases on the structure of `T`. The algorithm can be described with simple pseudocode:

```
subtype(S, T)
  match T with
    Top: return true
    Unit:
      match S with
        Unit: return true
        _ : return false
    T1 -> T2:
      match S with
        S1 -> S2: return [subtype(T1, S1) && subtype(S2, T2)]
        _ : return false
```

In order for this algorithm to make sense, we need to prove *completeness* and *soundness*. In order to show completeness, we must prove "If `S <: T`, then `subtype(S, T)` returns true." Soundness is just the converse of this statement: "If `subtype(S, T)` returns true, then `S <: T`."