Subtyping: An introduction

Brigitte Pientka

March 30, 2007

We will briefly describe the basics about subtyping. Up to now, we have shown how to enrich a language by adding in new language constructs together with their type. As a very important property we maintained that every expression had a unique type. In contrast, we will now consider subtyping, a feature which cross-cuts across all other language construct and affects our basic principles. In particular, we will not be able to maintain that an expression will have a unique type – an expression can have more than one type.

The goal of subtyping is to refine the existing typing rules so we can accept more programs. Consider the following simple program:

```
fun area r = 3.14 * r * r
```

The type of `area` will be `real -> real`. Hence, we would raise a type error, if we apply this function to an integer. This seems quite annoying, since we should be able to provide an `integer` where a `real` is required.

So how can we enrich our type system to handle subtyping? What effect will this have? Will type checking still be decidable?

> *Remark*: This will be a purely theoretical discussion. SML does not offer subtyping.

To extend a type system with subtyping, we first add an inference rule, called *subsumption*:

$$\frac{\Gamma \vdash e : S \quad S \leq T}{\Gamma \vdash e : T} \; \mathsf{T - Sub}$$

The rule says that an expression e is of type T if the expression e has type S and S is a subtype of T. So for example, 4 is of type `real` because 4 has type `int` and `int` is a subtype of the type `real`. The intuition of subtyping can be made precise with the subtyping principle.

Basic subtyping principle: $S \leq T$

S is a subtype of T if we can provide a value of type S
whenever a value of type T is required.

or

$$\text{Every value described by } S \text{ is also described by } T.$$

We will now try to formalize this subtyping relation between S and T. We consider each form of type (base types, tuples, functions, references, etc.) separately, and for each one we will define formally when it is safe to allow elements of one type of this form to be used where another is expected.

First, some relations between base types. Let us assume that in addition to `real` and `int`, we also have a base type `pos`, describing positive integers. We can then define the basic subtyping relations between the base types as follows.

$$\overline{\texttt{int} \leq \texttt{real}} \qquad \overline{\texttt{pos} \leq \texttt{int}}$$

Clearly, we should be able to provide a positive integer whenever a `real` is required, but our inference rules do not yet allow us to conclude this. To reason about subtyping relations, we will stipulate that subtyping should be *reflexive* and *transitive*.

$$\frac{}{T \leq T} \; \mathsf{S-ref} \qquad \frac{S \leq T' \quad T' \leq T}{S \leq T} \; \mathsf{S-trans}$$

Product

First, we will consider pairs and cross-products. Let's consider some examples.

$$
\begin{array}{llll}
\texttt{int} * \texttt{int} & \leq & \texttt{real} * \texttt{real} & \text{yes} \\
\texttt{int} * \texttt{int} & \leq & \texttt{int} * \texttt{real} & \text{yes} \\
\texttt{int} * \texttt{int} & \leq & \texttt{real} * \texttt{int} & \text{yes} \\
\texttt{int} * \texttt{real} & \leq & \texttt{int} * \texttt{real} & \text{yes} \\
\texttt{int} * \texttt{real} & \leq & \texttt{real} * \texttt{int} & \text{NO!}
\end{array}
$$

From these examples, we can conclude that the following subtyping relation between products seems sensible.

$$\frac{S_1 \leq T_1 \quad S_2 \leq T_2}{S_1 * S_2 \leq T_1 * T_2} \; \mathsf{S-Prod}$$

The subtyping rule for products is *co-variant*.

Records

We have seen records, which essentially were n-ary labeled tuples. Hence we can generalize the rule for products to records as follows:

$$\frac{S_1 \leq T_1 \quad \ldots \quad S_n \leq T_n}{\{x_1 : S_1, \ldots, x_n : S_n\} \leq \{x_1 : T_1, \ldots x_n : T\}} \; \mathsf{S-RDepth}$$

2

Since we compare each element of the record, this is also called *depth subtyping*.

However, we often would like to have a more powerful rule for records. Let us consider an example again, where we apply a function `fn r:{x:int} => #x r` which extracts the x-component of a record `r` is applied to a record which contains an x-component and a y-component.

```
(fn r:{x:int} => #x r) {x=0, y=1}
```

Clearly, this should be safe, since the function just requires that its argument is a record with a field `x` and its body never even access the y-component! In other words, it should always be safe to pass an argument `{x=0, y=1}` whenever a value of type `{x:int}` is required. Hence, it is always safe to forget some fields This seems to indicate the following subtyping rule for records.

$$\frac{n < k}{\{x_1 : T_1, \ldots, x_k : T_k\} \leq \{x_1 : T_1, \ldots, x_n : T_n\}} \; \text{S−Width}$$

It may be surprising that the subtype (the type considered "smaller") is actually the one with more fields. The easiest way to understand this is to view the record type `{x:int}` as describing the "set of all records with at least a field `x` of type `int`". Therefore, `{x=0}` is an element of this type, and so is for example `{x=0, y=1}` and `{x=0, y=true}`

A longer record constitutes a more informative and more demanding specification, and hence describes a smaller set of values.

This rule is also called *width subtyping*.

Finally, we do not care about the order of elements in a record, since every element has a unique label. Hence it should not matter whether we write `{x=0, y=1}` or `{y=1, x=0}`. This leads us to the third subtyping rule for records.

$$\frac{\text{where } \phi \text{ is a permutation}}{\{x_1 : T_1, \ldots, x_k : T_k\} \leq \{x_{\phi(1)} : T_{\phi(1)}, \ldots, x_{\phi(n)} : T_{\phi(n)}\}} \; \text{S−Perm}$$

Functions

Since we are working with functions, and functions can be passed as arguments and returned as results, it is natural to ask when a function of type $T \to S$ can be used in place of a function of type $T' \to S'$. Let's consider again some examples.

```
let
  (* areaSqr: real -> real *)
  fun areaSqr (r:real) = r * r
  (* fakeArea: int -> real *)
  fun fakeArea (r:real) = 3
in
  areaSqr 2.2
end
```

The expression `areaSqr 2.2` has type `real` and `areaSqr` has type `real -> real`. Clearly, we should be able to replace the function `areaSqr` with the function `fakeArea:real -> int`, since whenever a `real` is required it suffices to provide an `int`. This seems to suggest the following rule for functions:

$$\frac{S_2 \leq T_2}{S \rightarrow S_2 \leq S \rightarrow T_2} \; \mathsf{S-Fun-Try1}$$

Let's modify the program from above a little bit.

```
let
  (* areaSq: int -> int *)
  fun areaSq (r:int) = r * r
  (* fakeArea: real -> int *)
  fun fakeArea (r:real) = 3
in
  areaSq 2
end
```

The expression `areaSq 2` has type `int`and `areaSq` has type `int -> int`.

What happens if we replace `areaSq` with the function `fakeArea`? Clearly this is safe, because it is always safe to pass an integer to a function of type `real -> int`. This seems to suggest that we can provide a function of type `real -> int` whenever a function of type `int -> int` is required. This seems to suggest the following rule for functions.

$$\frac{T_1 \leq S_1}{S_1 \rightarrow S \leq T_1 \rightarrow S} \; \mathsf{S-Fun-Try2}$$

Notice that the sense of subtyping is reversed (contra-variant) for the argument type! Combining the two rules into one, we derive at our subtyping rule for functions.

$$\frac{T_1 \leq S_1 \quad S_2 \leq T_2}{S_1 \rightarrow S_2 \leq T_1 \rightarrow T_2} \; \mathsf{S-Fun}$$

Subtyping for functions is *contra-variant* in the input argument and *co-variant* in the result.

References

References are slightly tricky, since we can use a value of type `ref` T for reading and writing. Let us repeat the subtyping principle:

> Suppose the program context expects a value of type T, then we can provide a value of type S where $S \leq T$.

Example 1 If we have a variable x of type `ref real`, then we know that whenever we read from x we will have `!x:real`.

```
let
  val x = ref 2.0
  val y = ref 2
in
 (!x) * 3.14
end
```

But intuitively, we it should be safe to replace `!x` with `(!y):int`, since whenever a value of type `real` is required it suffices to provide a value of type `int`. This seems to suggest that whenever we need a value of type `real ref` it suffices to provide a value of type `int ref`.

$$\frac{S \leq T}{S \text{ ref} \leq T \text{ ref}} \; \mathsf{S-Ref-Try1}$$

On the other hand, we can not only use references for reading but also for writing.

```
let
  val x = ref 2.0
  val y = ref 2
in
  y := 4
end
```

Intuitively, it should also be safe to replace the assignment `y := 4` with the assignment `x := 4`, since x is a reference cell containing values of type `real` it should be safe to store a value of type `int`. But this suggests that when we require a value of type `int ref`, it suffices to provide a value of type `real ref`. So in other words this suggest the following rule:

$$\frac{T \leq S}{S \text{ ref} \leq T \text{ ref}} \; \mathsf{S-Ref-Try2}$$

However, now we have a problem. The two suggested rules are (almost) contradictory! To achieve a sound rule for subtyping in the presence of references, we cannot allow any subtyping to happen.

$$\frac{T \leq S \quad S \leq T}{S \text{ ref} \leq T \text{ ref}} \; \mathsf{S-Ref}$$

This means references are co-variant *and* contra-variant, in other words they are *invariant*. S and T must really be the same type.

It is worth pointing out that the analysis of reading and writing applies to any mutable data-structure. Therefore, it also applies for example to arrays.

$$\frac{T \leq S \quad S \leq T}{S \text{array} \leq T \text{array}} \; \mathsf{S-Array}$$

Unfortunate fact: In Java we omit the first premise during type checking. This is generally considered a flaw. Java fixes this flaw by inserting a runtime check for every assignment to an array thereby achieving type safety. This leads to a serious performance penalty.

Final remarks

Subtyping is characteristically found in *object-oriented* languages and is often considered essential. Generally, we find *nominal* subtyping in object-oriented systems, which is in contrast to *structural* subtyping, the subtyping we have considered here. In *nominal* subtyping, the user specifies what type (class) extends another type (class).

In Java, we often abuse subtyping. For example, elements in lists belong to the maximal type object in order to provide operations on for all lists, no matter what their elements are. If we add an element `x:int` to a list, we promote it to type object (upcast). If we take it out again, we unfortunately also only know it is an object. To do reasonable things with it, we must downcast it to the appropriate type. This however is an abuse of subtyping and often considered a "poor man's polymorphism". Newer versions of Java include some form of polymorphism, named generics, to avoid this kind of situation.

Up and down casts

Earlier we saw the subsumption rule.

$$\frac{\Gamma \vdash e : S \quad S \leq T}{\Gamma \vdash e : T} \; \mathsf{T - Sub}$$

This rule is completely unguided, and will always be applicable if we are trying to check or infer a type for an expression e. Moreover, it is not obvious that the subtyping relation itself is decidable. One easy way to achieve decidability, is to annotate the subsumption rule with a type. Typically, we have upcasts and downcasts.

$$\frac{\Gamma \vdash e : S \quad S \leq T}{\Gamma \vdash (T)e : T} \; \mathsf{T-Up} \qquad \frac{\Gamma \vdash e : S}{\Gamma \vdash (T)e : T} \; \mathsf{T-Down}$$

In an upcast, we ascribe a supertype of the type the type-reconstruction algorithm would find. This is always safe. It is useful if we want to hide some information in a record for example. In a downcast, we have no demand on the relationship between S and T. For example, we can write the following code:

```
fn (x:real) => (int)x + 1
```

This is in general dangerous, because not all values of type `real` can be safely cast to an integer. In effect, the programmer says, "I know why this can't happen – trust me!". To remain safe, a compiler must verify the typing constraint during run-time.