Chapter 12

Subtyping - An introduction

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:

let `area r = 3.14 *. r *. r`

The type of `area` will be `float -> float`. 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 `float` 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. OCaml does not offer subtyping.

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

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

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 `float` because 4 has type `int` and `int`

is a subtype of the type `float`. 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

> Every value described by S 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 `float` 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.

$$\frac{}{\texttt{int} \leq \texttt{float}} \qquad \frac{}{\texttt{pos} \leq \texttt{int}}$$

Clearly, we should be able to provide a positive integer whenever a `float` 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.

$\texttt{int} * \texttt{int}$	$\leq$	$\texttt{float} * \texttt{float}$	yes
$\texttt{int} * \texttt{int}$	$\leq$	$\texttt{int} * \texttt{float}$	yes
$\texttt{int} * \texttt{int}$	$\leq$	$\texttt{float} * \texttt{int}$	yes
$\texttt{int} * \texttt{float}$	$\leq$	$\texttt{int} * \texttt{float}$	yes
$\texttt{int} * \texttt{float}$	$\leq$	$\texttt{float} * \texttt{int}$	NO!

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:

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

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 `fun r:x:int -> r.x` which extracts the x-component of a record `r` is applied to a record which contains an x-component and a y-component.

```
(fun r:{x:int} -> r.x) {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.

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

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.

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

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.

```
  (* areaSqr: float -> float *)
let areaSqr (r:float) = r *. r in
  (* fakeArea: float -> int *)
let fakeArea (r:float) = 3 in
  areaSqr 2.2 + 4.2
```

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

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

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

```
  (* areaSq: int -> int *)
let areaSq (r:int) = r *. r in
  (* fakeArea: float -> int *)
let fakeArea (r:float) = 3 in
  areaSq 2 + 2
```

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 `float -> int`. This seems to suggest that we can provide a function of type `float -> int` whenever a function of type `int -> int` is required. This seems to suggest the following rule for functions.

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

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.

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

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 float`, then we know that whenever we read from `x` we will have `!x:float`.

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

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

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

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

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

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 `float` 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 `float ref`. So in other words this suggest the following rule:

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

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.

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

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

The importance of being coherent In OCaml, it would be convenient if int $\leq$ string and float $\leq$ string: then, instead of constantly writing `Int.toString` and `Float.toString`, as in `"i = "^Int.toString i ^"; x = "^Float.toString x`, we could write

```
"i = " ^i ^"; x = " ^x
```

This gives a chain of subtypings:

$$\mathsf{int} \leq \mathsf{float} \leq \mathsf{string}$$

where at each link in the chain, hidden coercions convert values of the subtype into values of the supertype.

But it's not quite that simple. The first coercion from int to float is `float` , and the second coercion from float to string is `string_of_float`. So, inserting coercions in the obvious way into the expression above gives

```
"i = " ^(string_of_float (float i)) ^"; x = " ^(string_of_float x)
```

which doesn't give the same result as the boring OCaml expression: instead of `string_of_float` we have `string_of_float (float i))`. So, if the original OCaml expression evaluated to `"i = 77; x = 2.5"`, the coerced expression would evaluate to `"i = 77.0; x = 2.5"`.

We could solve this particular problem by adding *another* coercion `string_of_int` directly from int to string. But we would have to make sure the compiler follows a convention that it use the shortest sequence of coercions, or special-case a preference for `string_of_int` over `string_of_float o float`. Now the programmer might have to reason nontrivially about which coercions are applied. Whenever there is more than one sequence of coercions to get from a subtype to a supertype, we say that subtyping is *incoherent*.

A nastier example of incoherence is a well-intentioned attempt to make programming more convenient by having, besides int $\leq$ float, the converse float $\leq$ int. Yes, above we treated this as being false or nonsensical, but why not just have a coercion `Float.toInt IEEEFloat.TO_NEAREST`? Well, now we could, logically, coerce float to int and then back to float, silently turning 2.3 into 2.0. So again we need to invoke a shortest-sequence convention, or a "no cycles" convention.

A few words on subtyping in Java Subclassing, a central feature of object-oriented languages, is a form of subtyping. In principle, this is perfectly reasonable, sound, and efficient. In practice, many object-oriented languages handle subtyping very imperfectly. The root of the problem is the combination of mutable state—arrays and objects with mutable fields (or *members*)—with subtyping.

When we add subtyping to an ML-like language, we resign ourselves to having no useful subtyping on references: S ref $\leq$ T ref if and only if $S \leq T$ and $T \leq S$,

that is, if S and T are equivalent types with the same set of values. This is the *only* statically sound formulation of the rule![1] This is highly restrictive, but the restriction is tolerable: While ML is an "impure" functional language—it *does* have mutable references, uncontrolled I/O, and other "imperative" features—it is still a functional language. There is often no need whatsoever to use those impure features, and when they are used, they often don't call for subtyping. (Think of using a reference as a counter, or a flag to control debugging.)

By contrast, in Java mutable state is used everywhere, so using a statically sound rule isn't practical. Instead, Java resorts to run-time checks to keep objects of the wrong type from being used.

An array is a collection of individually mutable elements. Logically, it is equivalent to a collection of mutable references. If Direct_Itinerary is a subtype (subclass) of Itinerary, then:

- code that expects an array of Itinerary can always *get* elements from a Direct_Itinerary array, because Direct_Itinerary is a subtype of Itinerary, but
- the same code cannot necessarily *put* elements into the Direct_Itinerary array, because it might try to write an Itinerary that is not a Direct_Itinerary. It can only reliably put an element that is a subtype of Direct_Itinerary.

The statically sound subtyping rule for arrays is, therefore, the same as for references:

$$\frac{S \leq T \qquad T \leq S}{(S \text{ array}) \leq (T \text{ array})}$$

However, this would prevent us from, for example, passing an array of Direct_Itinerary to a function that prints an Itinerary array, even if that function only reads from the array. So instead of the statically sound covariant-contravariant (or "bivariant" or "invariant") rule above, Java uses a covariant rule:

$$\frac{S \leq T}{(S \text{ array}) \leq (T \text{ array})}$$

This rule is statically unsound, but Java is supposed to be (unlike, say, C++) a *safe* language: Java programs should never crash in a completely uncontrolled way. So, whenever a Java program writes to an array of objects, it must compare the "tag" of the object being put in the array to a tag on the array itself, to make sure they are compatible. If we write an Itinerary to an array of Direct_Itinerary, the check that the class represented by the tag Itinerary is a subclass of the Direct_Itinerary will fail, and Java will raise an exception (which is bad, but better than segfaulting).

[1] Besides the even more restrictive rule $S \text{ ref} \leq S \text{ ref}$.

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 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.

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

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.

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

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:

```
fun (x:float) -> (int)x + 1
```

This is in general dangerous, because not all values of type `float` 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.