COMP3161/COMP9164 Supplementary Lecture Notes

Subtyping

Gabriele Keller Liam O'Connor Johannes Åman Pohjola

November 13, 2025

1 Subtyping

With type classes, the programmer can use the same overloaded function symbol + for addition of both floating point values and integer values, and the compiler or runtime will figure out which one to use. However, the following expression would still be rejected by the MinHs compiler:

$$1 + 1.75$$

This is because addition can be applied to two integers or two floats, but not a combination of both.[1] We explicitly have to convert the `Int` value to `Float` to add the two values.

C solves this "problem" using something called *integer promotion*: the basic types are ordered and if operations like `+` or `==` are applied to mixed operands, the one which is the lowest in the hierarchy is automatically cast to the higher type. This is quite convenient, but can easily lead to unexpected behaviour and subtle bugs, in particular with respect to signed/unsigned types.

The idea behind subtyping is similar to the approach in C. The key technical device is that that types can be partially ordered in a subtype relationship

$$\tau \leq \sigma$$

such that, whenever a value of some type σ is required, it is also fine to provide a value of type τ, as long as τ is a subtype of σ. For example, we could have the following subtype relationship:

$$\texttt{Int} \leq \texttt{Float} \leq \texttt{Double}$$

With subtyping, it would then be ok to have

$$1 +_{Float} 1.75$$

because even though the LHS is an `Int`, any subtype of `Float` is acceptable.

1.1 Coercion Interpretation

There are different ways to interpret the subtype relationship. One is to define τ to be a subtype of σ if $\tau \subseteq \sigma$. After all, in mathematics, the integers are a subset of the rationals. However, this interpretation is quite restrictive for a programming language: `Int` is *not* a subset of `Float` since they have very different representations. However, there is an obvious *coercion* from `Int` to `Float`. For our study of subtyping, we will focus on this so-called *coercion interpretation* of subtyping: τ is a subtype of σ, if there is a *sound* [2] coercion from values of type τ to values of type σ.

As another example, consider a `Graph` and a `Tree`. Since trees are a special case of graphs, trees can be converted into graphs, and we can view the tree type as subtype of the graph type in the coercion interpretation.

[1] In Haskell, this expression by itself would be fine, as constants are also overloaded and 1 has type $Num\ a \rightarrow a$. However, the compiler would also reject the addition of integer and float values, for example $(1 :: \texttt{Int}) + (1.7 :: \texttt{Float})$.

[2] More about that later

1.2 Properties

For a subtyping relationship to be sound, it has to be reflexive, transitive, and antisymmetric (with respect to type isomorphism). In other words $\leq$ is a *partial order*. This holds both for the subset interpretation and the coercion interpretation. For the subset interpretation, all three properties follow directly from the properties of the subset relation. In the coercion interpretation, reflexivity holds because the identity function is a coercion from $\tau \to \tau$. Transitivity holds since, given a coercion function from $f : \tau_1 \to \tau_2$ and $g : \tau_2 \to \tau_3$, the composition of f and g result in a coercion function from $\tau_1 \to \tau_3$.

In the coercion relation, the antisymmetry of subtyping means that if we can coerce τ to ρ and ρ back to τ, then it must be the case that $\tau \simeq \rho$. This is only true if the coercion functions are *injective* — that is, we can map each element of the *domain* (input) of the function to a *unique* element of the *codomain* (output).

1.3 Coherency of Coercion

Coercion should be *coherent*. That is, if there are two ways to coerce a value, both coercions should yield the same result.

For example, let us assume we define *Int* to be a subtype of *Float*, and both to be subtypes of *String*, with coercion functions

$$\mathit{intToFloat} :: \texttt{Int} \to \texttt{Float}$$

$$\mathit{intToString} :: \texttt{Int} \to \texttt{String}$$

$$\mathit{floatToString} :: \texttt{Float} \to \texttt{String}$$

On first sight, this looks reasonable. It is not coherent, however, because there are two coercion function from *Int* to *String*: the provided function *intToString*, but also *intToFloat* composed with *floatToString*. Unfortunately, applied to the number 3, for example, one would result in the string "3", the other in "3.0".

One reason why type promotion in C can be so tricky is precisely that it is not coherent in this way.

1.4 Variance

If we add subtyping to MinHS, one question that arises is how the subtyping relationship interacts with our type constructors. For example, if $\texttt{Int} \leq \texttt{Float}$, what about pairs, sums and function over these types? How do they relate to each other?

For pairs and sums, the answer is quite straight forward. Obviously, given a coercion function *intToFloat*, we can easily define coercion functions on pairs and sums:

$$\begin{aligned} &p1 :: (\texttt{Int} \times \texttt{Int}) \to (\texttt{Float} \times \texttt{Float}) \\ &p1\,(x,\ y) = (\mathit{intToFloat}\ x,\ \mathit{intToFloat}\ y) \end{aligned}$$

$$\begin{aligned} &p2 :: (\texttt{Int} \times \texttt{Float}) \to (\texttt{Float} \times \texttt{Float}) \\ &p2\,(x,\ y) = (\mathit{intToFloat}\ x,\ y) \\ &\ldots \end{aligned}$$

$$\begin{aligned} &s1 \quad :: (\texttt{Int} + \texttt{Int}) \to (\texttt{Float} + \texttt{Float}) \\ &s1\ x = \textbf{case}\ x\ \textbf{of} \\ &\qquad\qquad \mathsf{InL}\ v \to \mathit{intToFloat}\ v \end{aligned}$$

$$\mathsf{InR}\ v \ \rightarrow\ \mathit{intToFloat}\ v$$

...

This means that, if two types τ_1 and τ_2 are subtypes of σ_1 and σ_2, respectively, then the product type $\tau_1 \times \tau_2$ is also a subtype of $\sigma_1 \times \sigma_2$. The same is true for sums. More formally, we have:

$$\frac{\tau_1 \leq \rho_1 \qquad \tau_2 \leq \rho_2}{(\tau_1 \times \tau_2) \leq (\rho_1 \times \rho_2)}$$

$$\frac{\tau_1 \leq \rho_1 \qquad \tau_2 \leq \rho_2}{(\tau_1 + \tau_2) \leq (\rho_1 + \rho_2)}$$

The following diagram also shows the subtype relationship between pair types, for `Int` and `Float`:

$$\begin{array}{ccc} \texttt{Int} \times \texttt{Int} & \Longmapsto & \texttt{Int} \times \texttt{Float} \\ \Downarrow & & \Downarrow \\ \texttt{Float} \times \texttt{Int} & \Longmapsto & \texttt{Float} \times \texttt{Float} \end{array}$$

Since product and sum types interact with subtyping in such a natural way, it is tempting to expect the same to work for all type constructors. Resist this temptation.

Consider function types. Is `Int` $\rightarrow$ `Int` a subtype of `Float` $\rightarrow$ `Int`? That is, if a function of type `Float` $\rightarrow$ `Int` is required, would it be ok to provide a function of type `Int` $\rightarrow$ `Int` instead? Considering that the type `Int` is more restricted than the type `Float`, this means that a function which only works on the "smaller" type `Int` is also, in some sense, less powerful. Or, coming back to our second example, if we need a function to process any graph, then a function which only works on trees (and maybe relies on the fact that there are no cycles in a tree) is clearly not sufficient. We are also not able to define a coercion function in terms of our coercion function *intToFloat*:

$$c\ ::\ (\texttt{Int}\ \rightarrow\ \texttt{Float})\ \rightarrow\ (\texttt{Float}\ \rightarrow\ \texttt{Float})$$

The other direction, however, is actually quite easy:

$$\begin{array}{l} c'\ \ ::(\texttt{Float}\ \rightarrow\ \texttt{Float})\ \rightarrow\ (\texttt{Int}\ \rightarrow\ \texttt{Float}) \\ c'\ f = \textbf{let}\ g\ x\ =\ f\ (\mathit{intToFloat}\ x) \\ \qquad\quad \textbf{in}\ g \end{array}$$

Therefore, somewhat surprisingly, we have $(\texttt{Float}\ \rightarrow\ \texttt{Float})\ \leq\ (\texttt{Int}\ \rightarrow \texttt{Float})$.

So, what about the result type of a function: is `Int` $\rightarrow$ `Int` as subtype of `Int` $\rightarrow$ `Float`, vice versa, or are these types not in a subtype relationship at all? If we need a function which returns a `Float` and get one that returns an `Int`, that's ok since `Int`s can be coerced to `Float`s. Similarly, if we need a function which returns a graph, and we get a tree, that's ok since a tree is a special case of a graph.

$$\begin{array}{l} c''\ \ ::(\texttt{Int}\ \rightarrow\ \texttt{Int})\ \rightarrow\ (\texttt{Int}\ \rightarrow\ \texttt{Float}) \\ c''\ f = \textbf{let}\ g\ x\ =\ \mathit{intToFloat}\ (f\ x)\ \textbf{in}\ g \end{array}$$

To summarise, the subtype relationship on functions over `Int` and `Float` is as follows (of course, you can substitute any type τ for `Int`, ρ for `Float` here, as long as $\tau \leq \rho$):

$$\begin{array}{ccc} \texttt{Int} \rightarrow \texttt{Int} & \Longmapsto & \texttt{Int} \rightarrow \texttt{Float} \\ \Uparrow & & \Uparrow \\ \texttt{Float} \rightarrow \texttt{Int} & \Longmapsto & \texttt{Float} \rightarrow \texttt{Float} \end{array}$$

The subtype propagation rule for function types expresses exactly the same relationship:

$$\frac{\tau_1 \leq \rho_1 \qquad \tau_2 \leq \rho_2}{(\rho_1 \to \tau_2) \leq (\tau_1 \to \rho_2)}$$

Another example of a type which interacts with subtyping in a non-obvious manner are updateable arrays and reference types. To understand what is happening, let us have a look at Haskell-style updatable references. We have the following basic operations on this type:

newIORef :: $a \to$ IO (`IORef` a)	— Returns an initialised reference
writeIORef :: $a \to$ `IORef` $a \to$ IO ()	— Updates the value of a reference
readIORef :: `IORef` $a \to$ IO a	— Returns the current value

All other operations can be expressed in terms of these three operations.

The question now is: if $\tau \leq \sigma$, what is the subtype relationship between `IORef` τ and `IORef` σ? To check whether, for example, `IORef Int` $\leq$ `IORef Float`, think about what happens if we apply *writeIORef* (1.5 :: `Float`) to an `IORef Int` instead of an `IORef Float`. Clearly, this would not work, as the floating point value can't be stored in an `Int` reference. It would be okay the other way around: if we *writeIORef* (1 :: `Int`) apply to an `IORef Float`, it would be fine, since we could first coerce the value to a `Float` and store the result in the `Float` reference. This seems to suggest that `IORef Float` $\leq$ `IORef Int`.

However, if we assume that `IORef Float` $\leq$ `IORef Int`, we run into trouble with *readIORef*. If *readIORef* expects an `IORef Int`, and we give it an `IORef Float` instead, *readIORef* will return a floating point value which we cannot coerce to `Int`. In this case, the other direction would be fine: if we expect an `IORef Float`, we could apply it to an `IORef Int`, and then cast the resulting `Int` value to `Float`.

This means that $\tau \leq \sigma$ implies no subtype relationship between `IORef Float` and `IORefInt` at all: when a mutable reference of a certain type is required, we cannot substitute the reference for a sub- or supertype: we must give exactly that type.

We encounter exactly the same situation with updatable arrays. In Java, the language allows subtyping for arrays, at the cost of dynamic checks since this violates type safety.

Type constructors like product and sum, which leave the subtype relationship intact, as called *covariant*. type constructors which reverse the relationship, lie the function type in its first argument, are called *contravariant*. Type constructors like `IORef`, which do not imply a subtype relationship at all, are called *invariant*.