COMP3161/COMP9164 Supplementary Lecture Notes

Subtyping

Gabriele Keller, Liam O'Connor

November 11, 2019

1 Subtyping

With type classes, the programmer can use the same overloaded function symbol both for addition of floating point values and integer values, and the compiler will figure out which 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 in 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

$$1 +_{Float} 1.75$$

as floating point addition wants to floating point values, but also accepts `Int`s, as they are a subtype of `Float`.

1.1 Coercion Interpretation

There are different ways to interpret the subtype relationship: one would be to define τ to be a subtype of σ if it is a actual subset. For example, in the mathematical sense, integer numbers are a subset of rational numbers, even integral numbers of integral numbers and so on. However, this interpretation is quite restrictive for a programming language: `Int` is not a subset of `Float`, as they have very different representations. However, there is an obvious *coercion* from `Int` to `Float`.

[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 $(1 :: \texttt{Int}) + (1.7 :: \texttt{Float})$.

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

1.2 Properties

For a subtyping relationship to be sound, it has to be reflexive, transitive, and antisymmetric (with respect to type isomorphism). This means it is a *partial order*. This is the case for both the subset as well as 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 $\tau \rightarrow \tau$. Transitivity holds since, given a coercion function from $f : \tau_1 \rightarrow \tau_2$ and $g : \tau_2 \rightarrow \tau_3$, the composition of f and g result in a coercion function from $\tau_1 \rightarrow \tau_3$.

In order to guarantee that subtyping is *antisymmetric* in the subtyping interpretation, this must mean that if we can coerce τ to ρ and ρ back to τ, this must mean $\tau \simeq \rho$. 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

The coercion of values should be coherent. This means that, if there are two ways to coerce a value to a value of a supertype, both coercions have to 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

$$
\begin{aligned}
&\mathit{intToFloat} :: \texttt{Int} \rightarrow \texttt{Float}\\
&\mathit{intToString} :: \texttt{Int} \rightarrow \texttt{String}\\
&\mathit{floatToString} :: \texttt{Float} \rightarrow \texttt{String}
\end{aligned}
$$

On first sight, this looks like a reasonable relationship. 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 exactly 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 `Int` $\leq$ `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:

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

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

[2] More about that later

$$
\begin{array}{lll}
s1 & :: & (\texttt{Int} + \texttt{Int}) \rightarrow (\texttt{Float} + \texttt{Float}) \\
s1\ x & = & \textbf{case}\ x\ \textbf{of} \\
 & & \quad \mathsf{InL}\ v \rightarrow \mathit{intToFloat}\ v \\
 & & \quad \mathsf{InR}\ v \rightarrow \mathit{intToFloat}\ v \\
\ldots & &
\end{array}
$$

This means that, if two types τ_1 and τ_2 are subtypes of σ_1 and σ_2, respectively, then pairs of τ_1 and τ_2 are also subtypes of pairs of σ_1 and σ_2 and 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:

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

Given that the pair as well as the sum type constructor interacts with subtyping in such an obvious way, it is easy to be tricked into thinking this applied to all type constructors. Unfortunately, this is not the case.

Consider function types: is `Int` $\rightarrow$ `Int` as 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:

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

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`, it is not a problem, since we can easily convert that `Int` to a `Float`. Similarly, if we need a function which returns a graph, and we get a tree, it is ok as a tree is a special case of a graph and can be converted to the graph representation:

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

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 $\tau \leq \rho$):

$$
\begin{tikzcd}
\texttt{Int} \to \texttt{Int} \arrow[r, Mapsto] & \texttt{Int} \to \texttt{Float} \\
\texttt{Float} \to \texttt{Int} \arrow[r, Mapsto] \arrow[u, Rightarrow] & \texttt{Float} \to \texttt{Float} \arrow[u, Rightarrow]
\end{tikzcd}
$$

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:

$\mathit{newIORef}$	$:: a \to \mathsf{IO}\,(\texttt{IORef}\ a)$	— Returns an initialised reference
$\mathit{writeIORef}$	$:: a \to \texttt{IORef}\ a \to \mathsf{IO}\,()$	— Updates the value of a reference
$\mathit{readIORef}$	$:: \texttt{IORef}\ a \to \mathsf{IO}\ a$	— 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`, let us have a look at 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`, run into trouble with *readIORef*. If *readIORef* requires an `IORef Int`, because the it should return an `Int` value as result, and we apply it to an `IORef Float` instead, *readIORef* will return a floating point value which we cannot convert into an `Int`. In this case, the other direction would be fine: if it expects 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 `IORef Int` at all: when a reference of a certain type is required, we cannot substitute the reference for a sub- or supertype.

We encounter exactly the same situation with updatable arrays. In fact, in Java, the language allows subtyping for arrays, at the cost of dynamic checks, as 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*, and type constructors like `IORef`, which do not imply a subtype relationship at all are called *invariant*.