Lecture Notes on Subtyping

15-814: Types and Programming Languages
Frank Pfenning

Lecture 12
Thu Oct 2, 2025

1 Introduction

Subtyping is a general technique in order to allow more natural expression of certain kinds of program. One can also think of it as checking more properties of programs that can already be written. There are different notions of subtyping. The first, more general one is *coercive subtyping* where we allow $\tau \leq \sigma$ (read τ is a subtype of σ) if there is a prescribed way to convert a value of type τ to a value of type σ. A commonly-used example is $\mathsf{int} \leq \mathsf{float}$ that allows us to convert an integer into a floating point number. This is also an indication of some of the dangers of coercive subtyping, as not all int's may have an exact representation as floating point numbers. Moreover, since subtyping is generally implicit, it may not be clear when and how coercions are applied, which can lead to subtle bugs. For more on this point of view, see Breazu-Tannen et al. [1991].

Another point of view is the *subset interpretation* that says that $\tau \leq \sigma$ may hold if every value of type τ also has type σ, without the need to apply a conversion. It might seem that in our strict type-based development of the language, this should rarely if ever be the case. A key ingredient is to generalize the binary $(\tau + \sigma)$ and nullary (0) sums to be general *labeled sums*. We similarly generalize lazy records. That is, for finite index sets I we have

$$\text{Types} \quad \tau \quad ::= \quad \ldots \mid \textstyle\sum_{i \in I}(i : \tau_i) \mid \mathbin{\&}_{i \in I}(i : \tau_i)$$

Then we recover the binary and nullary cases as follows:

$$\begin{array}{lcl} \tau + \sigma & \triangleq & (\mathbf{l} : \tau) + (\mathbf{r} : \sigma) \\ 0 & \triangleq & \textstyle\sum_{i \in \{\}}() \\ \tau \mathbin{\&} \sigma & \triangleq & (\pi_1 : \tau) \mathbin{\&} (\pi_2 : \sigma) \\ \top & \triangleq & \mathbin{\&}_{\in \in \{\}}() \end{array}$$

This apparently small change has a major impact regarding subtyping. Consider in LAMBDA syntax:

```
type bool = ('true : 1) + ('false : 1)
type true = ('true : 1) + ()
type false =          () + ('false : 1)
```

Here, the empty parenthesis `()` are there just to disambiguate the unary sum from a unary lazy record formed with `&`. Here, it should be the case that $\mathsf{true} \leq \mathsf{bool}$ and $\mathsf{false} \leq \mathsf{bool}$ since there is only one value of type true, namely `'true ()`, while bool is inhabited by `'true ()` and `'false ()`.

Here is an example from lazy record:

```
type finstream = $stream. ('empty : 1) & ('next : nat * stream)
type infstream = $stream.           () & ('next : nat * stream)
```

The type finstream could be finite, while infstream is always infinite.

In this lecture we explore just the subtyping that arises from sums. There is extensive literature; you might find Lakhani et al. [2022] a useful entry point with many pointers to further readings.

2 Labeled Sums

The rules for finite labeled sums $\sum_{i \in I}(i : \tau_i)$ are a straightforward generalization of binary and nullary sums. We show here only the rules for the statics and dynamics, but we will have occasion to reconsider one of them, marked with an asterisk.

$$\frac{\Gamma \vdash \tau_i \; \mathit{type} \quad (\text{for all } i \in I)}{\Gamma \vdash \sum_{i \in I}(i : \tau_i) \; \mathit{type}} \; \mathsf{tp/sum}$$

$$\frac{(k \in I) \quad \Gamma \vdash e_k : \tau_k \quad \Gamma \vdash \sum_{i \in I}(i : \tau_i) \; \mathit{type}}{\Gamma \vdash k \cdot e_k : \sum_{i \in I}(i : \tau_i)} \; \mathsf{tp/tag}$$

$$\frac{\Gamma \vdash e : \sum_{i \in I}(i : \tau_i) \quad \Gamma, x_i : \tau_i \vdash e_i : \sigma \quad (\text{for all } i \in I)}{\Gamma \vdash \mathsf{case}\ e\ (i \cdot x_i \Rightarrow e_i)_{i \in I} : \sigma} \; \mathsf{tp/cases}^*$$

$$\frac{e \; \mathit{value}}{k \cdot e \; \mathit{value}} \; \mathsf{val/tag} \qquad \frac{v_k \; \mathit{value}}{\mathsf{case}\ k \cdot v_k\ (i \cdot x_i \Rightarrow e_i)_{i \in I} \mapsto [v_k/x_k]e_k} \; \mathsf{step/cases/tag}$$

$$\frac{e_1 \mapsto e_1'}{k \cdot e_1 \mapsto k \cdot e_1'} \; \mathsf{step/tag} \qquad \frac{e_0 \mapsto e_0'}{\mathsf{case}\ e_0\ (i \cdot x_i \Rightarrow e_i)_{i \in I} \mapsto \mathsf{case}\ e_0'\ (i \cdot x_i \Rightarrow e_i)_{i \in I}} \; \mathsf{step/cases}_0$$

3 Properties of Subtyping

When designing rules for subtyping $\tau \leq \sigma$ we need to make sure we understand the properties we want this relationship to satisfy. The first is somewhat straightforward: every value of type τ should also be a value of type σ.

Value inclusion: If $\cdot \vdash v : \tau$ and $\tau \leq \sigma$ then $\cdot \vdash v : \sigma$

As we will see, this by itself isn't sufficient to characterize subtyping. We recap the earlier example:

```
type bool = ('true : 1) + ('false : 1)
type true = ('true : 1) + ()
type false =          () + ('false : 1)
```

This suggests the following rule:

$$\frac{I \subseteq K}{\sum_{i \in I}(i : \tau_i) \leq \sum_{k \in k}(k : \tau_k)} \; \mathsf{sub/sum/width}$$

This is called *width subtyping* because we just take a subset of the labels. We can still satisfy the value inclusion principle if we allow the components that are present to be subtypes rather than be identity.

$$\frac{I \subseteq K \quad \tau_i \leq \sigma_i \quad (\text{for all } i \in I)}{\sum_{i \in I}(i : \tau_i) \leq \sum_{k \in k}(k : \sigma_k)}\ \mathsf{sub/sum}$$

This is often referred to as *depth subtyping* because we compare the components, not just the set of indices.

If we consider the interpretation of a type as a set of values, we could take one further step: the τ_i that are empty types do not contribute to the set of values. So we could have

$$\frac{\tau_j \textit{ empty} \quad (\text{for all } j \in J \subseteq I) \quad \tau_i \leq \sigma_i \quad (\text{for all } i \in I \backslash J)}{\sum_{i \in I}(i : \tau_i) \leq \sum_{k \in k}(k : \sigma_k)}\ \mathsf{sub/sum/empty}$$

We then also have to supply rules for the τ *empty* judgment. While sound, taking emptiness into account while subtyping can also lead to unexpected results by letting questionable code pass. For example, we would have $0 \times \mathsf{bool} \leq \mathsf{nat}$ but programs are a priori unlikely to want to take advantage of that. Also, whether types are empty is a rather delicate property. For example, in Haskell all data types are inhabited by $\bot$ (representing nontermination), while in the ML family of languages there are empty types such as `datatype empty = Void of empty`.

So for this lecture, we stick with the more robust middle ground: we have width subtyping and also depth subtyping, but we don't explicitly take emptiness into account. Still, it is the case that

$$0 \leq (\mathbf{true} : 1) + (\mathbf{false} : 1) = \mathsf{bool}$$

because it follows from the general rules for subtyping between sums because $\{\,\} \subseteq \{\mathbf{true}, \mathbf{false}\}$.

We consider the other types whose values are observable.

$$\frac{}{1 \leq 1}\ \mathsf{sub/one} \qquad \frac{\tau_1 \leq \sigma_1 \quad \tau_2 \leq \sigma_2}{\tau_1 \times \tau_2 \leq \sigma_1 \times \sigma_2}\ \mathsf{sub/times}$$

Now we could prove the value inclusion property as a theorem, by straightforward rule induction. Instead, we will move on to function types, which stand in for general behavioral types.

4 Subtyping Functions

Let's say we have a function, like *not* : $\mathsf{bool} \to \mathsf{bool}$ where $\mathsf{bool} = (\mathbf{true} : 1) + (\mathbf{false} : 1)$. Clearly, we can apply the function to a value of type $\mathsf{true} = (\mathbf{true} : 1) + ()$ because $\mathsf{true} \leq \mathsf{bool}$. So every value of type true also has type bool

Even though we haven't yet considered have recursive types, let's consider an example of them where we try to capture positive natural numbers.

```
type nat = $nat. ('zero : 1) + ('succ : nat)
type pos = () + ('succ : nat)
```

A quick observation: we have

```
fold 'zero () : nat
fold 'succ fold 'zero () : nat
'succ fold 'zero () : pos
```

so it is not quite true that every value of type `pos` also has type `nat`. So we need to add $\mu\alpha$ even if α does not occur, that is, the type `pos` isn't really recursive.

```
type nat = $nat. ('zero : 1) + ('succ : nat)
type pos = $a. () + ('succ : nat)
```

Now we have, as anticipated:

```
fold 'zero () : nat
fold 'succ fold 'zero () : nat
fold 'succ fold 'zero () : pos
```

So it should hold (and will, see Section 6) that pos $\leq$ nat. Then we would further expect that

```
decl succ : nat -> pos
defn succ = \x. fold 'succ x
```

We can confirm this by running it through the LAMBDA implementation, using the `--subtyping=true` flag.

Because every value of type pos also has type nat, we should also get `succ : nat -> nat`. In other words, we should have nat $\to$ pos $\leq$ nat $\to$ nat.

Conversely, since have checked that `succ : nat -> pos` it should also be the case that `succ : pos -> pos`. We have more information on the input to the function, but since it returns a positive number no matter which number is it given, this will be particularly be the case if know the input is positive. In summary: nat $\to$ pos $\leq$ pos $\to$ pos.

These considerations lead us to the following subtyping rule for functions.

$$\frac{\sigma_1 \leq \tau_1 \quad \tau_2 \leq \sigma_2}{\tau_1 \to \tau_2 \leq \sigma_1 \to \sigma_2}\ \mathsf{sub/arrow}$$

We say that the function type is *covariant* in its result and *contravariant* in its argument. In contrast, product types are covariant in both components.

To justify this rule we need to extend our principle of value inclusion to general expressions (because that's what the bodies of λ-abstraction are) and also account for nonempty contexts.

Right subsumption: If $\Gamma \vdash e : \tau$ and $\tau \leq \sigma$ then $\Gamma \vdash e : \sigma$

Left subsumption: If $\Gamma, x : \sigma \vdash e : \rho$ and $\tau \leq \sigma$ then $\Gamma, x : \tau \vdash e : \rho$.

It turns out, that in the bidirectional system for typechecking we only need to generalize the rule for changing direction.

$$\frac{\Gamma \vdash e \Rightarrow \tau \quad \tau \leq \sigma}{\Gamma \vdash e \Leftarrow \sigma}\ \Rightarrow/\Leftarrow$$

If we were considering universal or existential types (which are beyond the scope of this lecture), we would carry the context Γ into the subtyping judgment.

This kind of subtyping for functions is quite powerful. Consider the four possible types for the successor function.

```
type nat = $nat. ('zero : 1) + ('succ : nat)
type pos = $a. () + ('succ : nat)
% so pos <: nat

decl succ : nat -> pos
defn succ = \x. fold 'succ x
```

```
decl succ : nat -> nat  % since nat -> pos <: nat -> nat
decl succ : pos -> pos  % since nat -> pos <: pos -> pos
decl succ : pos -> nat  % since nat -> pos <: pos -> nat
```

We see that three are redundant once we have first type. It is not always the case that there is a most informative type, and we'll see some examples later.

5 Summary So Far

The rules for subtyping:

$$\frac{I \subseteq K \quad \tau_i \leq \sigma_i \quad (\text{for all } i \in I)}{\sum_{i \in I}(i : \tau_i) \leq \sum_{k \in k}(k : \sigma_k)}\ \mathsf{sub/sum}$$

$$\frac{}{1 \leq 1}\ \mathsf{tp/one} \qquad \frac{\tau_1 \leq \sigma_1 \quad \tau_2 \leq \sigma_2}{\tau_1 \times \tau_2 \leq \sigma_1 \times \sigma_2}\ \mathsf{tp/times}$$

$$\frac{\sigma_1 \leq \tau_1 \quad \tau_2 \leq \sigma_2}{\tau_1 \to \tau_2 \leq \sigma_1 \to \sigma_2}\ \mathsf{sub/arrow}$$

The properties of subtyping:

Right subsumption: If $\Gamma \vdash e : \tau$ and $\tau \leq \sigma$ then $\Gamma \vdash e : \sigma$

Left subsumption: If $\Gamma, x : \sigma \vdash e : \rho$ and $\tau \leq \sigma$ then $\Gamma, x : \tau \vdash e : \rho$.

When trying to prove these we discover that the earlier rules for typing a case over values of sum type is incorrect! What's the problem? Please think about this before reading on: it is an excellent exercise in bringing inference rules and their properties together.

To see the problem, consider

```
decl pred : nat -> nat
defn pred = \x. case unfold x of
     ( 'zero () => fold 'zero ()
     | 'succ y => y )
```

By the left subsumption principle, we shold also have

```
pred : pos -> nat
```

because every positive natural number is also a natural number ($\mathsf{pos} \leq \mathsf{nat}$). However, the definition of `pred` does not check against this type! We have

$$\dfrac{\dfrac{\dfrac{}{x : \mathsf{pos} \vdash x : \mathsf{pos}}}{x : \mathsf{pos} \vdash \mathsf{unfold}\, x : (\,) + (\mathbf{succ} : \mathsf{nat})}\ \mathsf{tp/unfold} \qquad ??}{x : \mathsf{pos} \vdash \mathsf{case}\ \mathsf{unfold}\, x\ (\mathbf{zero}\ (\,) \Rightarrow \mathsf{fold}\ \mathbf{zero}\ (\,) \mid \mathbf{succ}\ y \Rightarrow y)}\ \mathsf{tp/cases}}{\cdot \vdash \lambda x.\, \mathsf{case}\ \mathsf{unfold}\, x\ (\mathbf{zero}\ (\,) \Rightarrow \mathsf{fold}\ \mathbf{zero}\ (\,) \mid \mathbf{succ}\ y \Rightarrow y) : \mathsf{pos} \to \mathsf{nat}}\ \mathsf{tp/lam}$$

The problem here is that the subject of the case has type $(\,) + (\mathbf{succ} : \mathsf{nat})$ but the case expression has two branches: one for label **zero** and one for label **succ**. So far this expression to type-check we need to ignore the extraneous branch for label **zero**!

So we get the new rule

$$\frac{\Gamma \vdash e : \sum_{i \in I}(i : \tau_i) \quad I \subseteq J \quad \Gamma, x_i : \tau_i \vdash e_i : \sigma \quad (\text{for all } i \in I)}{\Gamma \vdash \mathsf{case}\ e\ (j \cdot x_j \Rightarrow e_j)_{j \in J} : \sigma}\ \mathsf{tp/cases}$$

From the perspective of a programmer, this may be unexpected but it is difficult to avoid and maintain the desirable properties of subtyping. A practical solution might be to track all the branches that are used and issue warnings for unreachable branches. Fortunately, all the properties of our language we have discussed like preservation, progress, canonical forms, sequentiality, finality of values, etc. remain intact if we allow unreachable branches.

We did not discuss it explicitly, but we also have reflexivity and transitive as properties of subtyping.

Reflexivity: $\tau \leq \tau$ for all types τ.

Transitivity: If $\tau \leq \sigma$ and $\sigma \leq \rho$ then $\tau \leq \rho$.

Both of these can be proved for the types so far (sums, products, units, and functions) by simple inductions. However, we pointedly omitted recursive types, which require a significant shift of viewpoint. Under this new viewpoint these properties still hold, but they can no longer be proved by induction.

6 Subtyping Recursive Types

As a more interesting example for recursive types, let's consider even and odd numbers. We'd like to satisfy

$$\begin{array}{lcl} \mathsf{nat} & \cong & (\mathbf{zero} : 1) + (\mathbf{succ} : \mathsf{nat}) \\ \mathsf{even} & \cong & (\mathbf{zero} : 1) + (\mathbf{succ} : \mathsf{odd}) \\ \mathsf{odd} & \cong & (\,) + (\mathbf{succ} : \mathsf{even}) \end{array}$$

The last two are mutually recursive, so we have to expand the second one in place to obtain explicit definitions.

$$\begin{array}{lcl} \mathsf{nat} & = & \mu\mathsf{nat}.\,(\mathbf{zero} : 1) + (\mathbf{succ} : \mathsf{nat}) \\ \mathsf{even} & = & \mu\mathsf{even}.\,(\mathbf{zero} : 1) + (\mathbf{succ} : \mu\mathsf{odd}.\,() + (\mathbf{succ} : \mathsf{even})) \\ \mathsf{odd} & = & \mu\mathsf{odd}.\,() + (\mathbf{succ} : \mathsf{even}) \end{array}$$

We would like to have that $\mathsf{even} \leq \mathsf{nat}$ and $\mathsf{odd} \leq \mathsf{nat}$, but neither $\mathsf{even} \leq \mathsf{odd}$ nor $\mathsf{odd} \leq \mathsf{even}$.

We can try the plausible rule

$$\frac{[\mu\alpha.\,\tau/\alpha]\tau \leq [\mu\beta.\,\sigma/\beta]\sigma}{\mu\alpha.\,\tau \leq \mu\beta.\,\sigma}\ \mathsf{sub/rec}$$

and start to construct a derivation.

$$\dfrac{\dfrac{\dfrac{}{1 \leq 1}\ \mathsf{sub/one} \qquad \begin{array}{c}\vdots\\ \mathsf{odd} \leq \mathsf{nat}\end{array}}{(\mathbf{zero} : 1) + (\mathbf{succ} : \mathsf{odd}) \leq (\mathbf{zero} : 1) + (\mathbf{succ} : \mathsf{nat})}\ \mathsf{sub/sum}}{\mathsf{even} \leq \mathsf{nat}}\ \mathsf{sub/rec}$$

Here the two premises of sub/sub account for the labels **zero** and **succ**. In the second premise, we once again unfold the recursive types.

$$\dfrac{\dfrac{\dfrac{}{1 \leq 1}\ \mathsf{sub/one} \qquad \dfrac{\begin{array}{c}\vdots\\ \mathsf{even} \leq \mathsf{nat}\end{array}}{() + (\mathbf{succ} : \mathsf{even}) \leq (\mathbf{zero} : 1) + (\mathbf{succ} : \mathsf{nat})}\ \mathsf{sub/sum}}{\mathsf{odd} \leq \mathsf{nat}}\ \mathsf{sub/rec}}{(\mathbf{zero} : 1) + (\mathbf{succ} : \mathsf{odd}) \leq (\mathbf{zero} : 1) + (\mathbf{succ} : \mathsf{nat})}\ \mathsf{sub/sum}}{\mathsf{even} \leq \mathsf{nat}}\ \mathsf{sub/rec}$$

At this point we seem to be stuck because the original goal of proving $\mathsf{even} \leq \mathsf{nat}$ has been reduced to itself! On the other hand, we haven't found a "counterexample", that is, a value that would be in the type on the smaller type but not the larger one. If such an example existed, we would have already encountered it because from now on, any attempt at creating a derivation of $\mathsf{even} \leq \mathsf{nat}$ will just repeat the same fragment of a derivation we wrote out so far.

This suggests that we can build a *cyclic derivation* as evidence that the subtyping judgment holds because the smallest counterexample cannot be above the node we use for the cycle.

Actually, any finite (closed off by axioms) or infinite derivation is acceptable as evidence that there is no counterexample. Unfortunately, infinite derivations are difficult to write out. Instead, we can view a finite but circular derivation as a finite representation of its infinite unfoldings. This is somewhat like $\mu\alpha.\,\tau$ serving as a representation of its infinite unfolding.

Can this work algorithmically as a decision procedure for subtyping? Yes! The key insight is that in a typing derivation of $\tau \leq \sigma$ there will be at most n^2 different questions we can ask when constructing a derivation bottom-up, where n is the number of type constructors in τ and σ (which we might write as $n = |\tau| + |\sigma|$. Any subexpression of τ might be compared to any subexpression of σ, but no more: all the rules except sub/rec access components of the types, and sub/rec just unfolds the type without creating any new subexpressions. This means that along any path in the

derivation we are trying to construct we must eventually find a counterexample (that is, no rule is applicable) or reach a cycle.

To illustrate this further, let's try to prove $\mathsf{odd} \leq \mathsf{even}$, which clearly should fail.

$$\dfrac{\dfrac{\dfrac{\text{no rule applies}}{(\mathbf{zero} : 1) + (\mathbf{succ} : \mathsf{odd}) \leq () + (\mathbf{succ} : \mathsf{even})}\ \mathsf{sub/rec}}{\mathsf{even} \leq \mathsf{odd}}\ \mathsf{sub/sum}}{\dfrac{() + (\mathbf{succ} : \mathsf{even}) \leq (\mathbf{zero} : 1) + (\mathbf{succ} : \mathsf{odd})}{\mathsf{odd} \leq \mathsf{even}}\ \mathsf{sub/rec}}$$

At the leaf, no rules applies because the label **zero** exists on the left-hand side but no on the right-hand side.

From this failed derivation we can construct a counterexample by using a corresponding constructor. As long as the rule applies, the constructors (and labels) will match, but the place where no rule applies, one side will allow a constructor the other one doesn't.

$$\begin{array}{ll} \vdash & \mathsf{fold}\ \mathbf{succ}\ \mathsf{fold}\ \mathbf{zero}\ (\,) : \mathsf{odd} \\ \nvdash & \mathsf{fold}\ \mathbf{succ}\ \mathsf{fold}\ \mathbf{zero}\ (\,) : \mathsf{nat} \end{array}$$

As mentioned above, there won't always be a (closed) counterexample unless we account for empty types. You can read about how to account for these and also construct counterexamples in Lakhani et al. [2022] and Ligatti et al. [2017]. The pioneering work on subtyping recursive types was by Amadio and Cardelli [1993] and Brandt and Henglein [1998].

7 Coinductively Interpreted Rules

Unfortunately, there is no universally accepted notation for inference rules to be interpreted coinductively, that is, allow infinite proofs. We use a dashed rule to indicate such rules.

$$\dfrac{I \subseteq K \quad \tau_i \leq \sigma_i \quad (\text{for all } i \in I)}{\sum_{i \in I}(i : \tau_i) \leq \sum_{k \in k}(k : \sigma_k)}\ \mathsf{sub/sum}$$

$$\dfrac{}{1 \leq 1}\ \mathsf{tp/one} \qquad \dfrac{\tau_1 \leq \sigma_1 \quad \tau_2 \leq \sigma_2}{\tau_1 \times \tau_2 \leq \sigma_1 \times \sigma_2}\ \mathsf{tp/times}$$

$$\dfrac{\sigma_1 \leq \tau_1 \quad \tau_2 \leq \sigma_2}{\tau_1 \to \tau_2 \leq \sigma_1 \to \sigma_2}\ \mathsf{sub/arrow}$$

$$\dfrac{[\mu\alpha.\,\tau/\alpha]\tau \leq [\mu\beta.\,\sigma/\beta]\sigma}{\mu\alpha.\,\tau \leq \mu\beta.\,\sigma}\ \mathsf{sub/rec}$$

In case a derivation is circular, we label a judgment with a number in parentheses and then use

this number to indicate a backlink.

$$\dfrac{\dfrac{}{1 \leq 1}\ \text{sub/one} \qquad \dfrac{\dfrac{\dfrac{[1]}{\mathsf{even} \leq \mathsf{nat}}}{() + (\mathbf{succ} : \mathsf{even}) \leq (\mathbf{zero} : 1) + (\mathbf{succ} : \mathsf{nat})}\ \text{sub/sum}}{\mathsf{odd} \leq \mathsf{nat}}\ \text{sub/rec}}{(\mathbf{zero} : 1) + (\mathbf{succ} : \mathsf{odd}) \leq (\mathbf{zero} : 1) + (\mathbf{succ} : \mathsf{nat})}\ \text{sub/sum}$$
$$\dfrac{}{\mathsf{even} \leq \mathsf{nat}}\ \text{sub/rec}$$
$$[1] \quad \mathsf{even} \leq \mathsf{nat}$$

The central properties of reflexivity and transitivity still hold for this coinductive judgment (as do the subsumption properties), but their proofs are no longer by induction but by coinduction, which is beyond the scope of this lecture.

There are techniques to transform infinitary systems into finitary ones, that work especially well if we know that derivations should eventually be finite or circular. For example, we can add a context of subtyping judgments we have seen and then succeed when encounter in the context. In this particular example this would look like this:

$$\dfrac{\dfrac{}{\mathsf{even} \leq \mathsf{nat} \vdash 1 \leq 1}\ \text{sub/one} \qquad \dfrac{\dfrac{\dfrac{}{\mathsf{even} \leq \mathsf{nat}, \mathsf{odd} \leq \mathsf{nat} \vdash \mathsf{even} \leq \mathsf{nat}}\ \text{hyp}}{\mathsf{even} \leq \mathsf{nat}, \mathsf{odd} \leq \mathsf{nat} \vdash () + (\mathbf{succ} : \mathsf{even}) \leq (\mathbf{zero} : 1) + (\mathbf{succ} : \mathsf{nat})}\ \text{sub/sum}}{\mathsf{even} \leq \mathsf{nat} \vdash \mathsf{odd} \leq \mathsf{nat}}\ \text{sub/rec}}{\mathsf{even} \leq \mathsf{nat} \vdash (\mathbf{zero} : 1) + (\mathbf{succ} : \mathsf{odd}) \leq (\mathbf{zero} : 1) + (\mathbf{succ} : \mathsf{nat})}\ \text{sub/sum}$$
$$\dfrac{}{\cdot \vdash \mathsf{even} \leq \mathsf{nat}}\ \text{sub/rec}$$

It should be easy to see that the two sets of inference rules coincide, and that derivations in the latter are the usual, inductive kind: something is derivable if there is a *finite* derivation of the judgment.

8 Subtyping in LAMBDA

You can experiment with subtyping in LAMBDA, using the `--subtyping=true` command line switch. You can find code with the examples from this lecture in lec12.cbv.

We have there one additional example we wrote in lecture, namely a version of the interpreter in continuation-passing style that distinguishes object-level values as a subtype of object-level expressions. Typing the guarantees that evaluation returns a value and not an arbitrary expression. See cps.cbv.

References

Roberto M. Amadio and Luca Cardelli. Subtyping recursive types. *ACM Transactions on Programming Languages and Systems*, 15(4):575–631, 1993.

Michael Brandt and Fritz Henglein. Coinductive axiomatization of recursive type equality and subtyping. *Fundamenta Informaticae*, 33(4):309–338, 1998.

Val Breazu-Tannen, Thierry Coquand, Carl Gunter, and Andre Scedrov. Inheritance as implicit coercion. *Information and Computation*, 93:172–221, 1991.

Zeeshan Lakhani, Ankush Das, Henry DeYoung, Andreia Mordido, and Frank Pfenning. Polarized subtyping. In Ilya Sergey, editor, *31st European Symposium on Programming (ESOP 2022)*, pages 431–461, Munich, Germany, April 2022. Springer LNCS 13240.

Jay Ligatti, Jeremy Blackburn, and Michael Nachtigal. On subtyping-relation completeness, with an application to iso-recursive types. *ACM Transactions on Programming Languages and Systems*, 39(4):4:1–4:36, March 2017.