CPSC 311: Definition of Programming Languages: Subtyping
22-subtyping

Joshua Dunfield
University of British Columbia

November 19, 2015

1 Review: Typing

2 Subtyping

In Typed Fun, every expression either has no type (the typing judgment $\Gamma \vdash e : \ldots$ cannot be derived; equivalently, `typeof` returns `#false`) or has a unique type, which is the A such that $\Gamma \vdash e : A$, or equivalently, the type A returned by `typeof`. Thus, types are *non-overlapping*: if $A \neq B$, then the set of expressions that have type A are disjoint from the expressions that have type B.

In everyday life (and everyday mathematics), we classify things rather more elaborately than Typed Fun: an entity or person may belong to several, overlapping categories. Overlapping categories are beyond our concern today; instead, we'll consider categories that are entirely contained within each other.

2.1 Our first subtyping system

Mathematicians would generally agree that the number represented by writing 2 is

- a positive integer (an n such that $n \geq 0$);
- an integer;
- a rational number (it can be written as a ratio $\frac{2}{1}$);
- a real number; and
- a complex number (whose imaginary part is 0).

Mathematically, the positive integers are a subset of the integers, which are a subset of the rationals, and so on.

Adding a category such as "even integer" would spoil this arrangement: some numbers are even but not positive, and some numbers are positive but not even. Subtyping *can* capture such relationships, but we'll save those for another time.

§1 Review: Typing

The above *inclusion relationships* are mathematically appealing, but some of them require caution in a programming language. In particular, the leap from rationals to reals is dangerous: computers represent "real" numbers as floating-point numbers, which are very strange approximations of real numbers. Converting from a rational to a float is liable to result in a number that is close to the rational, but not close enough. To avoid this problem, we won't attempt to claim that rational numbers (as stored in a computer) are a subset of floating-point numbers.

Fortunately, the first two inclusion relationships (positive integers $\subseteq$ integers, and integers $\subseteq$ rationals) are unproblematic. We currently don't have any of these types, however—only Num, which includes everything up to and including complex numbers.

In the past, I've glossed over exactly what counts as a Num in Fun by waving my hands in the general direction of Racket's notion of a number. I'll be slightly more rigorous now: I'll restrict Fun to rational numbers, steering clear of the problematic leap from rationals to floats (which are fake "reals"), and then wave my hands at the mathematical notion of a rational number (which I hope is the same as Racket's).

$$\begin{array}{llll} \text{Types} & A, B & ::= \mathsf{Bool} & \text{booleans} \\ & & \mid A \rightarrow B & \text{functions from } A \text{ to } B \\ & & \mid A * B & \text{products (pairs)} \\ & & \mid \mathsf{Pos} & \text{integers} \geq 0 \\ & & \mid \mathsf{Int} & \text{integers} \\ & & \mid \mathsf{Rat} & \text{rationals} \end{array}$$

Why would we want to do this? Well, we might want to know that the absolute value of an integer is always positive. Maybe the result of calling an absolute value function is used as an index to a string (see previous lecture notes), and we want to avoid having to check that the index is non-negative. Merely knowing that the index is an integer, rather than an arbitrary rational number, would eliminate an additional check.

Or, if you prefer, think of these three types as representing a simple class hierarchy in an object-oriented language. Some aspects of OO inheritance are already present in this context, so we can use this simpler setting to build up your intuition for how to define classes and inheritance.

(I kind of wanted to jump straight into OO-style subtyping, but instead we'll approach that "sideways". Most OO languages combine what I think of as several different features—records (things that have fields/instance variables/methods), inheritance (subtyping), mutability, self-reference—into one, "objects". But these features don't have to appear together, and I believe they can often be better understood separately.)

Adding a type to the grammar isn't useful unless we can give expressions that type. So let's add three typing rules (the third effectively replaces the rule we used to have for Num):

$$\frac{n \in \mathbb{Z} \qquad n \geq 0}{\Gamma \vdash (\mathsf{num}\ n) : \mathsf{Pos}}\ \text{Type-pos} \qquad \frac{n \in \mathbb{Z}}{\Gamma \vdash (\mathsf{num}\ n) : \mathsf{Int}}\ \text{Type-int} \qquad \frac{n \in \mathbb{Q}}{\Gamma \vdash (\mathsf{num}\ n) : \mathsf{Rat}}\ \text{Type-rat}$$

Considering just one binary operator, =, will illustrate several aspects of subtyping. Suppose we have a specialized version of Type-binop, just for =:

$$\frac{\Gamma \vdash e1 : \mathsf{Rat} \qquad \Gamma \vdash e2 : \mathsf{Rat}}{\Gamma \vdash (\mathsf{binop}\ (\mathsf{equalsop})\ e1\ e2) : \mathsf{Bool}}\ \text{Type-binop-eq}$$

This is very different from Typed Fun: a single expression, like $(\mathsf{num}\ 5)$, can have more than one type. (In fact, $(\mathsf{num}\ 5)$ has three different types.) If our entire program is $(\mathsf{num}\ n)$ for some n, this isn't a problem. But for more realistic programs, we've painted ourselves into a corner. Consider this silly function:

$$(\mathsf{lam}\ x\ \mathsf{Pos}\ (\mathsf{binop}\ (\mathsf{equalsop})\ (\mathsf{id}\ x)\ (\mathsf{id}\ x)))$$

Ignore how silly this function is. It may be silly, and applying it will always evaluate to (btrue), but it's still a function that should typecheck. We won't be able to, however. In its derivation we will assume $x : \mathsf{Pos}$, but the premises of Type-binop-eq require (reasonably enough) that the expressions have type Rat.

But in fact, every *closed value* (that is, every expression that (1) has no free variables and (2) is a value) that has type Pos also has type Rat (and Int as well): The only closed values of type Pos have the form $(\mathsf{num}\ n)$ where $n \in \mathbb{Z}$, and $\mathbb{Z} \subseteq \mathbb{Q}$, so $n \in \mathbb{Q}$, so by rule Type-rat, $\emptyset \vdash (\mathsf{num}\ n) : \mathsf{Rat}$.

Our next steps are:

- Design *subtyping rules* that define when one type is a subtype of (included in) another type.
- Update our typing rules to make use of the subtyping rules.

Based on the set inclusions $\{n \in \mathbb{Z} \mid n \geq 0\} \subseteq \mathbb{Z}$ and $\mathbb{Z} \subseteq \mathbb{Q}$, we can write our first subtyping rules:

$$\frac{}{\mathsf{Pos} <: \mathsf{Int}}\ \text{Sub-pos-int} \qquad \frac{}{\mathsf{Int} <: \mathsf{Rat}}\ \text{Sub-int-rat}$$

In set theory, we know that the subset relation is reflexive (every set is a subset of itself) and transitive (if $S_1 \subseteq S_2$ and $S_2 \subseteq S_3$, then $S_1 \subseteq S_3$). Every subtyping judgment should have these same properties. The easiest way to ensure this (at least "on paper") is to add two more rules:

$$\frac{}{A <: A}\ \text{Sub-refl} \qquad \frac{A1 <: A2 \qquad A2 <: A3}{A1 <: A3}\ \text{Sub-trans}$$

For now, the only useful application of Sub-trans is to derive $\mathsf{Pos} <: \mathsf{Rat}$:

$$\frac{\dfrac{}{\mathsf{Pos} <: \mathsf{Int}}\ \text{Sub-pos-int} \qquad \dfrac{}{\mathsf{Int} <: \mathsf{Rat}}\ \text{Sub-int-rat}}{\mathsf{Pos} <: \mathsf{Rat}}\ \text{Sub-trans}$$

These four rules (the general rules Sub-refl and Sub-trans, and the rules specific to our numeric types, Sub-pos-int and Sub-int-rat) constitute a pretty good, or at least non-broken, subtyping system. So we can move on to update our typing rules.

2.2 Soundness of subtyping

How do we know that a set of subtyping rules makes sense? For typing rules, we talked about *type safety*: if the typing rules say e has type A, and evaluating e produces a value v, that value v *also* has type A. Otherwise, the static and dynamic semantics don't match.

For subtyping, we can define *subtype soundness*:

Definition 1. Subtype soundness holds if, for all v, A, B such that $\emptyset \vdash v : A$ *without* using Type-sub, and $A <: B$, then $\emptyset \vdash v : B$ *without* using Type-sub.

Type-sub is a rule we'll develop below, but since the definition doesn't let you use that rule within itself, it's okay that we haven't developed it yet!

So, for example, a rule

$$\frac{}{\mathsf{Rat} <: (\mathsf{Rat} \rightarrow \mathsf{Rat})}\;\text{??Sub-rat-arr}$$

violates subtype soundness, because there exists a v (actually, a whole lot of vs) such that

$$\emptyset \vdash v : \mathsf{Rat}$$

but *not*

$$\emptyset \vdash v : (\mathsf{Rat} \rightarrow \mathsf{Rat})$$

In fact, *every* value of type Rat is a valid counterexample.

2.3 Adding subtyping to the type system

Adding subtyping is easy; adding subtyping that can be easily implemented takes some work.

The easy way is to add a single rule, called the *subsumption rule*:

$$\frac{\Gamma \vdash e : A \qquad A <: B}{\Gamma \vdash e : B}\;\text{Type-sub}$$

Rule Type-sub says that if we determine that e has type A, and A is a subtype of B, then e has type B.

Using Type-sub, we can type the function that gave us trouble before:

$$\dfrac{\dfrac{\dfrac{\dfrac{(x : \mathsf{Pos}) \in (x : \mathsf{Pos})}{x : \mathsf{Pos} \vdash (\mathsf{id}\ x) : \mathsf{Pos}}\;\text{Type-id} \qquad \overset{\checkmark}{\mathsf{Pos} <: \mathsf{Rat}}}{x : \mathsf{Pos} \vdash (\mathsf{id}\ x) : \mathsf{Rat}}\;\text{Type-sub} \qquad \overset{\checkmark}{x : \mathsf{Pos} \vdash (\mathsf{id}\ x) : \mathsf{Rat}}}{x : \mathsf{Pos} \vdash (\mathsf{binop}\ (\mathsf{equalsop})\ (\mathsf{id}\ x)\ (\mathsf{id}\ x)) : \mathsf{Bool}}\;\text{Type-binop-eq}}{\emptyset \vdash \big(\mathsf{lam}\ x\ \mathsf{Pos}\ (\mathsf{binop}\ (\mathsf{equalsop})\ (\mathsf{id}\ x)\ (\mathsf{id}\ x))\big) : \mathsf{Pos} \rightarrow \mathsf{Bool}}\;\text{Type-lam}$$

Unfortunately, Type-sub has cheerfully destroyed a useful property of the typing rules: they are no longer *syntax-directed*.

Definition 2. A set of typing rules is *syntax-directed* if, for each syntactic form (variant of the abstract syntax), only one rule has a conclusion that potentially matches that form.

(Warning: the term "syntax-directed" is sometimes used loosely, or with a slightly different meaning—but in all the usages I can recall, our typing rules *were* syntax-directed before we added Type-sub, and they are now *not* syntax-directed.)

We exploited this property to write `typeof`. We also exploited a rather similar property to write `interp`: for each variant of the abstract syntax, either one (usually) or *two* rules have a suitable conclusion (the variants with two rules being ite and, recently, lazy-ptr). For the variants with two rules, we could figure out which rule to try by evaluating an expression (ite) or inspecting the store (lazy-ptr).

Type-sub has broken this property, because Type-sub's conclusion works for *any* expression! No matter what e is, it is *possible* that we will need to use Type-sub. Even more thrillingly, instead of making a recursive call to `typeof` on a smaller expression, Type-sub has us making a recursive call on *the same expression*! Thus, if we implement our typing rules including Type-sub, we must be careful not to try to derive

$$\dfrac{\dfrac{\begin{array}{c}\vdots\\ \Gamma \vdash e : \_\_\_\end{array} \qquad \_\_\_ <: \_\_\_}{\Gamma \vdash e : \_\_\_}\,\text{Type-sub} \qquad \_\_\_ <: \_\_\_}{\Gamma \vdash e : \_\_\_}\,\text{Type-sub}$$

Fortunately, we never need to apply Type-sub twice in a row, because subtyping is transitive. So if $e = (\mathsf{num}\ 1)$ and we derived

$$\dfrac{\dfrac{\dfrac{}{\Gamma \vdash (\mathsf{num}\ 1) : \mathsf{Pos}}\,\text{Type-pos} \qquad \dfrac{}{\mathsf{Pos} <: \mathsf{Int}}\,\text{Sub-pos-int}}{\Gamma \vdash (\mathsf{num}\ 1) : \mathsf{Int}}\,\text{Type-sub} \qquad \dfrac{}{\mathsf{Int} <: \mathsf{Rat}}\,\text{Sub-int-rat}}{\Gamma \vdash (\mathsf{num}\ 1) : \mathsf{Rat}}\,\text{Type-sub}$$

we could instead have derived

$$\dfrac{\dfrac{}{\Gamma \vdash (\mathsf{num}\ 1) : \mathsf{Pos}}\,\text{Type-pos} \qquad \overset{\checkmark}{\mathsf{Pos} <: \mathsf{Rat}}}{\Gamma \vdash (\mathsf{num}\ 1) : \mathsf{Rat}}\,\text{Type-sub}$$

Transitivity took care of that problem, but we still need to know when to try to apply Type-sub. Let's try to figure that out.

What does that mean? Well, some of the rules, like Type-pair, just don't care:

$$\dfrac{\Gamma \vdash e1 : A1 \qquad \Gamma \vdash e2 : A2}{\Gamma \vdash (\mathsf{pair}\ e1\ e2) : A1 * A2}\,\text{Type-pair}$$

This rule makes no demands on $A1$ and $A2$. We never need to use Type-sub as the last (bottom-most) step of deriving $\Gamma \vdash e1 : A1$ or $\Gamma \vdash e1 : A2$. (Note that we might need Type-sub *somewhere* inside the derivations of these premises, but not as the *last* step.)

Other rules do require something about the types. For example, Type-pair-case requires that the type of the scrutinee e be a product type $A1 * A2$. The rule itself doesn't care what $A1$ and $A2$ are, but e has to be some kind of product and not, say, Rat or Pos $\rightarrow$ Pos.

$$\dfrac{\Gamma \vdash e : (A1 * A2) \qquad x1 : A1, x2 : A2, \Gamma \vdash eBody : B}{\Gamma \vdash (\mathsf{pair\text{-}case}\ e\ x1\ x2\ eBody) : B}\,\text{Type-pair-case}$$

However, here we *also* don't need to use Type-sub, because (for the moment) we don't have any subtyping for products—except reflexivity: $(A1 * A2) <: (A1 * A2)$—so it won't do us any good. If we *did* have subtyping for product types, we *still* wouldn't want to use it!

Suppose we added some rules so that $(\mathsf{Pos} * \mathsf{Int}) <: (\mathsf{Int} * \mathsf{Int})$, and then tried to derive

$$\dfrac{\dfrac{\Gamma \vdash e : (\mathsf{Pos} * \mathsf{Int})}{\Gamma \vdash e : (\mathsf{Int} * \mathsf{Int})}\text{Type-sub} \qquad x1 : \mathsf{Int}, x2 : \mathsf{Int}, \Gamma \vdash eBody : B}{\Gamma \vdash (\mathsf{pair\text{-}case}\ e\ x1\ x2\ \underbrace{(\mathsf{app}\ (\mathsf{id}\ pow2)\ (\mathsf{id}\ x1))}_{eBody}) : B}\text{Type-pair-case}$$

where $\Gamma = (pow2 : \mathsf{Pos} \rightarrow \mathsf{Pos})$.

The idea is that $pow2$ is a Fun function such that $(\mathsf{app}\ (\mathsf{id}\ pow2)\ (\mathsf{num}\ k))$ returns the kth power of 2, for integers $k \geq 0$. This function only works for nonnegative integer powers (otherwise it would have to deal with cases like raising 2 to the power $1/3$), so its type is $\mathsf{Pos} \rightarrow \cdots$.

(The result of raising 2 to such a power is always a nonnegative integer, so its result type is also Pos. But it's the domain of $pow2$ that matters in this example.)

I haven't chosen an e yet; I can use a pair whose first component is a positive integer, and whose second component is an integer:

$$e = (\mathsf{pair}\ (\mathsf{num}\ 3)\ (\mathsf{num}\ 4))$$

I'm missing a derivation for $\Gamma \vdash e : (\mathsf{Pos} * \mathsf{Int})$; I'll leave that as an exercise:

Exercise 3. Complete the derivation tree:

$$\dfrac{\qquad\qquad\qquad\qquad}{\Gamma \vdash (\mathsf{pair}\ (\mathsf{num}\ 3)\ (\mathsf{num}\ 4)) : (\mathsf{Pos} * \mathsf{Int})}\ \text{------------}$$

(If you used Type-sub to do this, try it again without using Type-sub.)

Returning to the above example, and assuming you did the exercise, the derivation tree we have so far is

$$\dfrac{\dfrac{\overset{\checkmark}{\Gamma \vdash e : (\mathsf{Pos} * \mathsf{Int})}}{\Gamma \vdash e : (\mathsf{Int} * \mathsf{Int})}\text{Type-sub} \qquad x1 : \mathsf{Int}, x2 : \mathsf{Int}, \Gamma \vdash eBody : B}{\Gamma \vdash (\mathsf{pair\text{-}case}\ e\ x1\ x2\ \underbrace{(\mathsf{app}\ (\mathsf{id}\ pow2)\ (\mathsf{id}\ x1))}_{eBody}) : B}\text{Type-pair-case}$$

Now we want to derive

$$x1 : \mathsf{Int}, x2 : \mathsf{Int}, \Gamma \vdash \underbrace{(\mathsf{app}\ (\mathsf{id}\ pow2)\ (\mathsf{id}\ x1))}_{eBody} : B$$

Trying Type-app, we get

$$\frac{x1 : \mathsf{Int}, x2 : \mathsf{Int}, \Gamma \vdash (\mathsf{id}\ pow2) : (\mathsf{Pos} \to \mathsf{Pos}) \qquad x1 : \mathsf{Int}, x2 : \mathsf{Int}, \Gamma \vdash (\mathsf{id}\ x1) : \mathsf{Pos}}{x1 : \mathsf{Int}, x2 : \mathsf{Int}, \Gamma \vdash \underbrace{(\mathsf{app}\ (\mathsf{id}\ pow2)\ (\mathsf{id}\ x1))}_{eBody} : \mathsf{Pos}}\ \text{Type-app}$$

The first premise can be derived with Type-id, recalling that $\Gamma = (pow2 : \mathsf{Pos} \to \mathsf{Pos})$.

But the second premise can't be derived! We know that $x1$ is an integer, but we don't know that it's a positive integer. We knew that the scrutinee had type $\mathsf{Pos} * \mathsf{Int}$, but we forgot that information when we used Type-sub.

The lesson here is not to use Type-sub unless you really need to. One place where we do need to use Type-sub is Type-binop-eq, which requires that the expressions being compared have type Rat.

3 Developing subtyping

Once we decide that the values of a type can also be values of another, larger type, subtyping becomes another (nested) step in the recipe of adding a feature to the language:

1. Extend the concrete syntax.
2. Extend the abstract syntax.
3. Extend the dynamic semantics (e.g. environment-based evaluation rules).
4. For a typed language, extend the static semantics (e.g. typing rules):
   (a) For a language with subtyping, extend the subtyping rules.
5. (Not in CPSC 311.) Prove desirable properties of the language (e.g. type safety).

(Steps 3–4 need not be done in that order.)

We're adding subtyping rather late in the game, but we can go back through the features we've built up, adding subtyping to them.

3.1 Product types (pair types)

For products, the subtyping rule works "pairwise":

$$\frac{A1 <: B1 \qquad A2 <: B2}{(A1 * A2) <: (B1 * B2)}\ \text{Sub-product}$$

For example, every pair of an Int and a Bool is also a pair of a Rat and a Bool:

$$\frac{\dfrac{}{\mathsf{Int} <: \mathsf{Rat}}\ \text{Sub-int-rat} \qquad \dfrac{}{\mathsf{Bool} <: \mathsf{Bool}}\ \text{Sub-refl}}{(\underbrace{\mathsf{Int}}_{A1} * \underbrace{\mathsf{Bool}}_{A2}) <: (\underbrace{\mathsf{Rat}}_{B1} * \underbrace{\mathsf{Bool}}_{B2})}\ \text{Sub-product}$$

3.2 Lists

For lists, we can follow the pattern of pairs (for this purpose, the Lispish notion that a list is "really" a pair is not wrong):

$$\frac{A <: B}{(\mathsf{List}\ A) <: (\mathsf{List}\ B)}\ \text{Sub-list}$$

For example, every list of positive integers is also a list of integers.

Notice that for both products and lists, the subtyping in the premise(s) "goes the same way" as the subtyping in the conclusion: In Sub-list, A appears on the left of $<:$ in the conclusion, and in the premise. In Sub-product, $A1$ appears on the left of $<:$ in the conclusion, and also on the left of $<:$ in a premise; $A2$ works similarly.

Because the subtyping goes the same way, Sub-product and Sub-list are said to be *covariant*.

3.3 Functions

A function type $A1 \rightarrow A2$ has two types inside it, a domain of inputs $A1$ and a range of outputs (or "codomain") $A2$. Following the pattern of Sub-product, we get

$$\frac{A1 <: B1 \qquad A2 <: B2}{(A1 \rightarrow A2) <: (B1 \rightarrow B2)}\ \text{??Sub-arr}$$

However, to quote John Reynolds, "As usual, something funny happens at the left of the arrow." (This is one of the enduring truths of programming languages.) Using ??Sub-arr, we can derive

$$\frac{\dfrac{}{\mathsf{Int} <: \mathsf{Rat}}\ \text{Sub-int-rat} \qquad \dfrac{}{\mathsf{Bool} <: \mathsf{Bool}}\ \text{Sub-refl}}{(\mathsf{Int} \rightarrow \mathsf{Bool}) <: (\mathsf{Rat} \rightarrow \mathsf{Bool})}\ \text{??Sub-arr}$$

This should mean that, if we expect to be given a function of type $(\mathsf{Rat} \rightarrow \mathsf{Bool})$, we should be happy with a function of type $(\mathsf{Int} \rightarrow \mathsf{Bool})$. But a function of type $(\mathsf{Int} \rightarrow \mathsf{Bool})$ is only half as good as one of type $(\mathsf{Rat} \rightarrow \mathsf{Bool})$, because a function whose domain is Rat can be applied to any rational number, while a function whose domain is Int can only be applied to integers.

Informally, ??Sub-arr is validating false advertising: a function that only handles integers should not be able to pass itself off as a function that handles all rational numbers.

More formally, rule ??Sub-arr violates the Liskov[–Wing] (1994) "Subtype Requirement" (often called the "Liskov substitution principle"):

> Let $\varphi(x)$ be a property provable about objects x of type T.
> Then $\varphi(y)$ should be true for objects y of type S where S is a subtype of T.

As our property Φ we can essentially use type safety: a property of functions f of type $\mathsf{Rat} \rightarrow \mathsf{Bool}$ is that, when applied to any value of type Rat, certain errors will not occur.

However, this property is *not* true of all functions g of type $\mathsf{Int} \to \mathsf{Bool}$: type safety tells us that, for any function $g : (\mathsf{Int} \to \mathsf{Bool})$, if we apply g to any value of type Int, certain errors will not occur.

But that doesn't tell us that those errors will not occur *for arguments that are not integers*. Here, it's useful to recall something from our treatment of strings in Typed Fun (`15-strings.pdf`). We added an expression nth that returned the nth character in a string:

$$\frac{eS \Downarrow (\mathsf{str}\ s1) \qquad eIdx \Downarrow (\mathsf{num}\ n) \qquad n \in \mathbb{Z} \qquad n \geq 0 \qquad n < \mathit{len}(s1)}{(\mathsf{nth}\ eS\ eIdx) \Downarrow (\mathsf{str}\ s1_n)}\ \text{Eval-nth}$$

$$\frac{\Gamma \vdash eS : A1 \qquad A1 = \mathsf{String} \qquad \Gamma \vdash eIdx : A2 \qquad A2 = \cancel{\mathsf{Num}}\ \mathsf{Rat}}{\Gamma \vdash (\mathsf{nth}\ eS\ eIdx) : \mathsf{String}}\ \text{Type-nth}$$

Since we only had a generic Num type, whenever we evaluated nth we had to check that the index evaluated to a number that was (1) an integer, (2) ≥ 0, and (3) less than the length of the string.

Now that we have a type Int, we can remove the check for n being an integer from Eval-nth, provided Type-nth checks that $eIdx$ is an Int and not merely a Rat:

$$\frac{eS \Downarrow (\mathsf{str}\ s1) \qquad eIdx \Downarrow (\mathsf{num}\ n) \qquad \cancel{n \in \mathbb{Z}} \qquad n \geq 0 \qquad n < \mathit{len}(s1)}{(\mathsf{nth}\ eS\ eIdx) \Downarrow (\mathsf{str}\ s1_n)}\ \text{Eval-nth}$$

$$\frac{\Gamma \vdash eS : A1 \qquad A1 = \mathsf{String} \qquad \Gamma \vdash eIdx : A2 \qquad A2 = \cancel{\mathsf{Num}}\ \cancel{\mathsf{Rat}}\ \mathsf{Int}}{\Gamma \vdash (\mathsf{nth}\ eS\ eIdx) : \mathsf{String}}\ \text{Type-nth}$$

Let g be the function

$$(\mathsf{lam}\ x\ \mathsf{Int}\ (\mathsf{nth}\ (\mathsf{str}\ \texttt{"hello"})\ (\mathsf{id}\ x)))$$

which has type $\mathsf{Int} \to \mathsf{Bool}$. Applying g to a Rat, say 2.5, will lead to an error that should be impossible: taking the 2.5th character of a string.

So rule ??Sub-arr doesn't work. (Some people call the relation described by ??Sub-arr "naïve subtyping". I do not approve: subtyping that uses ??Sub-arr is not a form of subtyping that is naïve, it's *not subtyping at all!* If you want even more disapproval of this term, consult Ron Garcia.)

A rule that does work is this one, which is *contravariant* in the domain, meaning the subtyping "goes the other way" in the premise for the function domains $A1$ and $B1$:

$$\frac{B1 <: A1 \qquad A2 <: B2}{(A1 \to A2) <: (B1 \to B2)}\ \text{Sub-arr}$$

3.4 Refs

$\boxed{\Gamma \vdash e : A}$ Under assumptions Γ, expression e has type A

$$\frac{\Gamma \vdash e : A \qquad A <: B}{\Gamma \vdash e : B}\ \text{Type-sub} \qquad \frac{(x : A) \in \Gamma}{\Gamma \vdash (\mathsf{id}\ x) : A}\ \text{Type-var}$$

$$\frac{}{\Gamma \vdash (\mathsf{num}\ n) : \mathsf{Num}}\ \text{Type-num} \qquad \frac{op : A1 * A2 \to B \qquad \Gamma \vdash e1 : A1 \qquad \Gamma \vdash e2 : A2}{\Gamma \vdash (\mathsf{binop}\ op\ e1\ e2) : B}\ \text{Type-binop}$$

$$\frac{}{\Gamma \vdash (\mathsf{bfalse}) : \mathsf{Bool}}\ \text{Type-false} \qquad \frac{}{\Gamma \vdash (\mathsf{btrue}) : \mathsf{Bool}}\ \text{Type-true}$$

$$\frac{\Gamma \vdash e : \mathsf{Bool} \qquad \Gamma \vdash eThen : A \qquad \Gamma \vdash eElse : A}{\Gamma \vdash (\mathsf{ite}\ e\ eThen\ eElse) : A}\ \text{Type-ite}$$

$$\frac{x : A, \Gamma \vdash eBody : B}{\Gamma \vdash (\mathsf{lam}\ x\ A\ eBody) : A \to B}\ \text{Type-lam} \qquad \frac{\Gamma \vdash e1 : A \to B \qquad \Gamma \vdash e2 : A}{\Gamma \vdash (\mathsf{app}\ e1\ e2) : B}\ \text{Type-app}$$

$$\frac{\Gamma \vdash e1 : A1 \qquad \Gamma \vdash e2 : A2}{\Gamma \vdash (\mathsf{pair}\ e1\ e2) : A1 * A2}\ \text{Type-pair} \qquad \frac{\Gamma \vdash e : A1 * A2 \qquad x1 : A1, x2 : A2, \Gamma \vdash eBody : B}{\Gamma \vdash (\mathsf{pair\text{-}case}\ e\ x1\ x2\ eBody) : B}\ \text{Type-pair-case}$$

$$\frac{\Gamma \vdash e : A \qquad x : A, \Gamma \vdash eBody : B}{\Gamma \vdash (\mathsf{with}\ x\ e\ eBody) : B}\ \text{Type-with} \qquad \frac{u : B, \Gamma \vdash e : B}{\Gamma \vdash (\mathsf{rec}\ u\ B\ e) : B}\ \text{Type-rec}$$

$$\frac{}{\Gamma \vdash (\mathsf{list\text{-}empty}\ A) : \mathsf{List}\ A}\ \text{Type-empty} \qquad \frac{\Gamma \vdash e1 : A \qquad \Gamma \vdash e2 : \mathsf{List}\ A}{\Gamma \vdash (\mathsf{list\text{-}cons}\ e1\ e2) : \mathsf{List}\ A}\ \text{Type-cons}$$

$$\frac{\Gamma \vdash e : \mathsf{List}\ A \qquad \Gamma \vdash eEmpty : B \qquad xh : A, xt : \mathsf{List}\ A, \Gamma \vdash eCons : B}{\Gamma \vdash (\mathsf{list\text{-}case}\ e\ eEmpty\ xh\ xt\ eCons) : B}\ \text{Type-list-case}$$

$$\frac{\Gamma \vdash e : A}{\Gamma \vdash (\mathsf{ref}\ e) : \mathsf{Ref}\ A}\ \text{Type-ref} \qquad \frac{\Gamma \vdash e : \mathsf{Ref}\ A}{\Gamma \vdash (\mathsf{deref}\ e) : A}\ \text{Type-deref} \qquad \frac{\Gamma \vdash e1 : \mathsf{Ref}\ A \qquad \Gamma \vdash e2 : A}{\Gamma \vdash (\mathsf{setref}\ e1\ e2) : A}\ \text{Type-setref}$$

Figure 1 **Typing rules for Typed Fun with pairs, lists, and refs**

3.4.1 Subtyping for refs

Following the pattern of List, we might write a covariant rule for references:

$$\frac{A <: B}{(\mathsf{Ref}\ A) <: (\mathsf{Ref}\ B)}\ \text{??Sub-ref}$$

By this rule, (Ref Int) <: (Ref Rat). However, if you expect something of type Ref Rat and I give you an expression of type (Ref Int), you can use setref to replace the reference's contents with 3.5 (because, to you, it is a Ref Rat and you can assign any Rat to it).

So we might try contravariance:

$$\frac{B <: A}{(\mathsf{Ref}\ A) <: (\mathsf{Ref}\ B)}\ \text{??Sub-ref-2}$$

Now, however, if you expect something of type (Ref Int) and deref it, expecting an Int, you may be disappointed: By ??Sub-ref-2, (Ref Rat) <: (Ref Int). But the contents of (Ref Rat) could be 3.5 or any rational number, not necessarily an integer.

The covariant rule ??Sub-ref works fine with deref, but not with setref; the contravariant rule ??Sub-ref-2 works fine with setref, but not with deref. So the covariant rule enforces a necessary condition for deref, and the contravariant rule enforces a necessary condition for setref. Therefore, a correct rule is:

$$\frac{A <: B \qquad B <: A}{(\mathsf{Ref}\ A) <: (\mathsf{Ref}\ B)}\ \text{Sub-ref}$$

which enforces *both* conditions.

(We might try to "optimize" this rule by replacing the premises with $A = B$. This is probably okay for this system, but doesn't work for all type systems, so I'd rather leave it as is.)

The following may be a useful additional explanation, particularly if you understand contravariant subtyping for function types $A1 \rightarrow A2$. We can think of a reference as an object with two methods, called deref and setref:

- The deref "method" has no arguments (we are thinking of this, for the moment, as a class method, so the reference to "self" or "this" is implicit), and returns (for a reference of type (Ref A)) a value of type A.

  So we can think of the type of deref as $() \rightarrow A$, where $()$ represents taking zero arguments.

- The setref "method" takes one argument, of type A (assuming the reference has type (Ref A)). It also returns the value of the argument. So we can think of the type of setref as $A \rightarrow A$.

Thus, the deref "method" has type $() \rightarrow A$ and setref has type $A \rightarrow A$. According to the contravariant rule for functions, Sub-arr, we can compare the types of the deref method of a reference of type (Ref A) and the deref method of a reference of type (Ref B) as follows:

$$\frac{() <: () \qquad A <: B}{(() \rightarrow A) <: (() \rightarrow B)}\ \text{Sub-arr}$$

The second premise here matches the covariant premise of Sub-ref. (Regardless of whatever () is, exactly, the first premise is derivable using Sub-refl.)

For setref, we get

$$\frac{B <: A \qquad A <: B}{(A \to A) <: (B \to B)}\ \text{Sub-arr}$$

The second premise here is something of an accident: we happened to decide that setref should return the new contents just written to the reference. If we said, instead, that setref returned "nothing", which we seem to be writing as $()$, then we would have

$$\frac{B <: A \qquad () <: ()}{(A \to ()) <: (B \to ())}\ \text{Sub-arr}$$

3.5 Upper bounds

Something I hadn't thought of by Monday's lecture: there are a few more places where we need to use Type-sub. We need to use it in Type-ite; otherwise, `typeof` will return `false` for the expression

$$(\text{ite}\ (\text{btrue})\ (\text{num}\ 1)\ (\text{num}\ {-1}))$$

This is because $(\text{num}\ 1)$ has type Pos, and $(\text{num}\ {-1})$ has type Int, but $\text{Pos} \neq \text{Int}$. So when we implement Type-ite, we need to find the *upper bound* of the types of the $eThen$ and $eElse$ branches:

$$\frac{\Gamma \vdash e : \text{Bool} \qquad \Gamma \vdash eThen : A \qquad \Gamma \vdash eElse : A}{\Gamma \vdash (\text{ite}\ e\ eThen\ eElse) : A}\ \text{Type-ite}$$

$$\frac{\Gamma \vdash e : B \qquad B = \text{Bool} \qquad \Gamma \vdash eThen : A1 \qquad \Gamma \vdash eElse : A2 \qquad A1 = A2}{\Gamma \vdash (\text{ite}\ e\ eThen\ eElse) : A1}\ \text{Type-ite}$$

$$\frac{\Gamma \vdash e : B \qquad B <: \text{Bool} \qquad \Gamma \vdash eThen : A1 \qquad A1 <: A \qquad \Gamma \vdash eElse : A2 \qquad A2 <: A}{\Gamma \vdash (\text{ite}\ e\ eThen\ eElse) : A}\ \text{Type-ite*}$$

This last version of Type-ite, marked *, is really just the original Type-ite with three uses of Type-sub:

$$\frac{\dfrac{\Gamma \vdash e : B \qquad B <: \text{Bool}}{\Gamma \vdash e : \text{Bool}}\ \text{Type-sub} \qquad \dfrac{\Gamma \vdash eThen : A1 \qquad A1 <: A}{\Gamma \vdash eThen : A}\ \text{Type-sub} \qquad \dfrac{\Gamma \vdash eElse : A2 \qquad A2 <: A}{\Gamma \vdash eElse : A}\ \text{Type-sub}}{\Gamma \vdash (\text{ite}\ e\ eThen\ eElse) : A}\ \text{Type-ite}$$

That is, Type-ite* is an easier rule to implement, but Type-ite* isn't adding any power to the type system. (It's harder, actually, to prove that Type-ite* isn't *taking anything away* from the type system. But I'm pretty sure it isn't.)

We also need to do this in some other rules, such as Type-list-case, so I wrote a function `upper-bound` that takes two types A and B, and returns A if $B <: A$, and B if $A <: B$. See the updated version of `subtyping.rkt`.