

# CPSC 311: Introduction to Subtyping (DRAFT)

## (“lec-subtyping”)

Joshua Dunfield  
University of British Columbia

October 21, 2016

### 1 Subtyping

In Typed Fun, every expression either has no type (the typing judgment  $\Gamma \vdash e : \dots$  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 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, like the diagrams I showed on Monday.

#### 1.1 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.

The above *inclusion relationships* are fine mathematically, 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.

## §1 Subtyping

---

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).

Types	A, B ::=	bool	booleans
		A → B	functions from A to B
		A * B	products (pairs)
		pos	integers ≥ 0
		int	integers
		rat	rationals

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. Also, knowing that the index is an integer, rather than an arbitrary rational number, would eliminate an additional check.

(That leaves the last check: that the index is less than the length of the string. Some type systems will get rid of that check too. . .)

Or, if you prefer, think of these three types as representing a small 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 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 "side-ways". 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):

$$\frac{n \in \mathbb{Z} \quad n \geq 0}{\Gamma \vdash (\text{Num } n) : \text{pos}} \text{Type-pos} \quad \frac{n \in \mathbb{Z}}{\Gamma \vdash (\text{Num } n) : \text{int}} \text{Type-int} \quad \frac{n \in \mathbb{Q}}{\Gamma \vdash (\text{Num } n) : \text{rat}} \text{Type-rat}$$

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

$$\frac{\Gamma \vdash e1 : \text{rat} \quad \Gamma \vdash e2 : \text{rat}}{\Gamma \vdash (\text{Binop (Equalsop) } e1 \ e2) : \text{bool}} \text{Type-binop-eq}$$

This is very different from Typed Fun: a single expression, like (Num 5), can have more than one type. (In fact, (Num 5) has three different types.) If our entire program is (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:

(Lam x pos (Binop (Equalsop) (Id x) (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

## §1 Subtyping

---

assume  $x : \text{pos}$ , 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 `(Num n)` where  $n \in \mathbb{Z}$ , and  $\mathbb{Z} \subseteq \mathbb{Q}$ , so  $n \in \mathbb{Q}$ , so by rule `Type-rat`,  $\emptyset \vdash (\text{Num } n) : \text{rat}$ .

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:

$$\frac{}{\text{pos} <: \text{int}} \text{Sub-pos-int} \qquad \frac{}{\text{int} <: \text{rat}} \text{Sub-int-rat}$$

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  $S_1 \subseteq S_3$ ). Any good definition of subtyping should have these same properties. The easiest way to ensure this (at least “on paper”) is to add two more rules:

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

For now, the only useful application of `Sub-trans` is to derive `pos <: rat`:

$$\frac{\frac{}{\text{pos} <: \text{int}} \text{Sub-pos-int} \quad \frac{}{\text{int} <: \text{rat}} \text{Sub-int-rat}}{\text{pos} <: \text{rat}} \text{Sub-trans}$$

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.

### 1.2 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 values  $v$  and types  $A, B$  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

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

## §1 Subtyping

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

$$\emptyset \vdash v : \text{rat}$$

but *not*

$$\emptyset \vdash v : (\text{rat} \rightarrow \text{rat})$$

In fact, *every* value of type  $\text{rat}$  is a valid counterexample.

### 1.3 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*:

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

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:

$$\frac{\frac{\frac{(x : \text{pos})(x) = \text{pos}}{x : \text{pos} \vdash (\text{Id } x) : \text{pos}} \text{Type-id} \quad \checkmark}{x : \text{pos} \vdash (\text{Id } x) : \text{rat}} \text{Type-sub} \quad \frac{\checkmark}{x : \text{pos} \vdash (\text{Id } x) : \text{rat}} \text{Type-binop-eq}}{x : \text{pos} \vdash (\text{Binop (Equalsop) (Id } x) (\text{Id } x)) : \text{bool}} \text{Type-lam}}{\emptyset \vdash (\text{Lam } x \text{ pos (Binop (Equalsop) (Id } x) (\text{Id } x))) : \text{pos} \rightarrow \text{bool}} \text{Type-lam}$$

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 `lte` and, recently, `Tree-case`). For the variants with two rules, we could figure out which rule to try by evaluating an expression.

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

$$\frac{\frac{\frac{\vdots}{\Gamma \vdash e : \text{---}} \text{Type-sub} \quad \text{---} <: \text{---}}{\Gamma \vdash e : \text{---}} \text{Type-sub}}{\Gamma \vdash e : \text{---}} \text{Type-sub}$$

## §1 Subtyping

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

$$\frac{\frac{\frac{}{\Gamma \vdash (\text{Num } 1) : \text{pos}} \text{Type-pos} \quad \frac{}{\text{pos} <: \text{int}} \text{Sub-pos-int}}{\Gamma \vdash (\text{Num } 1) : \text{int}} \text{Type-sub} \quad \frac{}{\text{int} <: \text{rat}} \text{Sub-int-rat}}{\Gamma \vdash (\text{Num } 1) : \text{rat}} \text{Type-sub}$$

we could instead have derived

$$\frac{\frac{}{\Gamma \vdash (\text{Num } 1) : \text{pos}} \text{Type-pos} \quad \overset{\checkmark}{\text{pos} <: \text{rat}}}{\Gamma \vdash (\text{Num } 1) : \text{rat}} \text{Type-sub}$$

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:

$$\frac{\Gamma \vdash e1 : A1 \quad \Gamma \vdash e2 : A2}{\Gamma \vdash (\text{Pair } e1 \ e2) : A1 * A2} \text{Type-pair}$$

This rule makes no demands on  $A1$  and  $A2$ . We never need to use Type-sub as the last (bottommost) 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,  $\text{rat}$  or  $\text{pos} \rightarrow \text{pos}$ .

$$\frac{\Gamma \vdash e : (A1 * A2) \quad x1 : A1, x2 : A2, \Gamma \vdash e\text{Body} : B}{\Gamma \vdash (\text{Pair-case } e \ x1 \ x2 \ e\text{Body}) : B} \text{Type-pair-case}$$

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  $(\text{pos} * \text{int}) <: (\text{int} * \text{int})$ , and then tried to derive

$$\frac{\frac{\Gamma \vdash e : (\text{pos} * \text{int})}{\Gamma \vdash e : (\text{int} * \text{int})} \text{Type-sub} \quad x1 : \text{int}, x2 : \text{int}, \Gamma \vdash e\text{Body} : B}{\Gamma \vdash (\text{Pair-case } e \ x1 \ x2 \ \underbrace{(\text{App } (\text{Id } \text{pow2}) \ (\text{Id } x1)))}_{e\text{Body}}) : B} \text{Type-pair-case}$$

where  $\Gamma = (\text{pow2} : \text{pos} \rightarrow \text{pos})$ .

The idea is that  $\text{pow2}$  is a Fun function such that  $(\text{App } (\text{Id } \text{pow2}) \ (\text{Num } k))$  returns the  $k$ th 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  $\text{pos} \rightarrow \dots$ .

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

## §1 Subtyping

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:

$$e = (\text{Pair } (\text{Num } 3) (\text{Num } 4))$$

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

■ **Exercise 3.** Complete the derivation tree:

$$\frac{}{\Gamma \vdash (\text{Pair } (\text{Num } 3) (\text{Num } 4)) : (\text{pos} * \text{int})} \text{-----}$$

(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

$$\frac{\frac{\checkmark}{\Gamma \vdash e : (\text{pos} * \text{int})} \text{Type-sub} \quad \frac{}{\Gamma \vdash e : (\text{int} * \text{int})} \text{Type-sub} \quad \frac{}{x1 : \text{int}, x2 : \text{int}, \Gamma \vdash e\text{Body} : B} \text{Type-pair-case}}{\Gamma \vdash (\text{Pair-case } e \ x1 \ x2 \ (\underbrace{\text{App } (\text{Id } \text{pow2}) (\text{Id } x1)}_{e\text{Body}})) : B} \text{Type-pair-case}$$

Now we want to derive

$$x1 : \text{int}, x2 : \text{int}, \Gamma \vdash (\underbrace{\text{App } (\text{Id } \text{pow2}) (\text{Id } x1)}_{e\text{Body}}) : B$$

Trying Type-app, we get

$$\frac{x1 : \text{int}, x2 : \text{int}, \Gamma \vdash (\text{Id } \text{pow2}) : (\text{pos} \rightarrow \text{pos}) \quad x1 : \text{int}, x2 : \text{int}, \Gamma \vdash (\text{Id } x1) : \text{pos}}{x1 : \text{int}, x2 : \text{int}, \Gamma \vdash (\underbrace{\text{App } (\text{Id } \text{pow2}) (\text{Id } x1)}_{e\text{Body}}) : \text{pos}} \text{Type-app}$$

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

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  $\text{pos} * \text{int}$ , 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  $\text{rat}$ .

## 2 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:

## §2 Developing subtyping

1. Extend the concrete syntax.
2. Extend the abstract syntax.
3. Extend the dynamic semantics (e.g. 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.

### 2.1 Product types (pair types)

For products, the subtyping rule works “pairwise”:

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

For example, every pair of an int and a bool is also a pair of a rat and a bool:

$$\frac{\frac{}{\text{int} <: \text{rat}} \text{Sub-int-rat} \quad \frac{}{\text{bool} <: \text{bool}} \text{Sub-refl}}{\underbrace{(\text{int} * \text{bool})}_{A1 \quad A2} <: \underbrace{(\text{rat} * \text{bool})}_{B1 \quad B2}} \text{Sub-product}$$

### 2.2 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):

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

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

### 2.3 Trees

For trees, we can also follow the pattern of pairs.

$$\frac{A <: B}{(\text{tree } A) <: (\text{tree } B)} \text{Sub-tree}$$

For example, every tree whose keys are positive integers is also a tree whose keys are integers.

Notice that for products, lists and trees, the subtyping in the premise(s) “goes the same way” as the subtyping in the conclusion: In Sub-tree,  $A$  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, Sub-list and Sub-tree are said to be *covariant*.

## 2.4 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

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

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

$$\frac{\frac{}{\text{int} <: \text{rat}} \text{Sub-int-rat} \quad \frac{}{\text{bool} <: \text{bool}} \text{Sub-refl}}{(\text{int} \rightarrow \text{bool}) <: (\text{rat} \rightarrow \text{bool})} \text{??Sub-arr}$$

This should mean that, if we expect to be given a function of type  $(\text{rat} \rightarrow \text{bool})$ , we should be happy with a function of type  $(\text{int} \rightarrow \text{bool})$ . But a function of type  $(\text{int} \rightarrow \text{bool})$  is only half as good as one of type  $(\text{rat} \rightarrow \text{bool})$ , because a function whose domain is  $\text{rat}$  can be applied to any rational number, while a function whose domain is  $\text{int}$  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  $\Phi$  we can essentially use type safety: a property of functions  $f$  of type  $\text{rat} \rightarrow \text{bool}$  is that, when applied to any value of type  $\text{rat}$ , certain errors will not occur.

However, this property is *not* true of all functions  $g$  of type  $\text{int} \rightarrow \text{bool}$ : type safety tells us that, for any function  $g : (\text{int} \rightarrow \text{bool})$ , if we apply  $g$  to any value of type  $\text{int}$ , 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. We added an expression  $\text{Nth}$  that returned the  $n$ th character in a string:

$$\frac{eS \Downarrow (\text{Str } s1) \quad eIdx \Downarrow (\text{num } n) \quad n \in \mathbb{Z} \quad n \geq 0 \quad n < \text{len}(s1)}{(\text{Nth } eS \ eIdx) \Downarrow (\text{Str } s1_n)} \text{Eval-nth}$$

$$\frac{\Gamma \vdash eS : A1 \quad A1 = \text{string} \quad \Gamma \vdash eIdx : A2 \quad A2 = \text{num rat}}{\Gamma \vdash (\text{Nth } eS \ eIdx) : \text{string}} \text{Type-nth}$$

Since we only had a generic  $\text{num}$  type, whenever we evaluated  $\text{Nth}$  we had to check that the index evaluated to a number that was (1) an integer, (2)  $\geq 0$ , and (3) less than the length of the string.

Now that we have a type  $\text{int}$ , we can remove the check for  $n$  being an integer from  $\text{Eval-nth}$ , provided  $\text{Type-nth}$  checks that  $eIdx$  is an  $\text{int}$  and not merely a  $\text{rat}$ :

$$\frac{eS \Downarrow (\text{Str } s1) \quad eIdx \Downarrow (\text{num } n) \quad n \in \mathbb{Z} \quad n \geq 0 \quad n < \text{len}(s1)}{(\text{Nth } eS \ eIdx) \Downarrow (\text{Str } s1_n)} \text{Eval-nth}$$

$$\frac{\Gamma \vdash eS : A1 \quad A1 = \text{string} \quad \Gamma \vdash eIdx : A2 \quad A2 = \text{num rat int}}{\Gamma \vdash (\text{Nth } eS \ eIdx) : \text{string}} \text{Type-nth}$$



## §2 Developing subtyping

---

Let  $g$  be the function

$$(\text{Lam } x \text{ int } (\text{Nth } (\text{Str } \text{"hello"}) (\text{Id } x)))$$

which has type  $\text{int} \rightarrow \text{bool}$ . 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  $\text{Sub-arr}$  doesn't work. (Some people call the relation described by  $\text{Sub-arr}$  “naïve subtyping”. I do not approve: subtyping that uses  $\text{Sub-arr}$  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  $B1$ :

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