Whirlwind Introduction to Subtyping

Cameron Wong

The purpose of this text is to provide a high-level overview of the motivations behind and basic theory of a subsumptive subtyping semantics over a simply-typed lambda calculus similar to ML. In the interest of brevity, we will elide a formal discussion of syntax forms, dynamics, etc, and simply discuss what properties this system should have if embedded into ML. As such, you should have some basic knowledge of ML and the formalities of ML-like typesystems.

Disclaimer: The few "proofs" given *are not* completely rigorous. The biggest elephant in the room is that we have not formally defined the behavior of our system, and instead nebulously pretend that we've grafted things onto OCaml (or SML). The sections marked as "proof" are intended as sketches that may be superimposed onto a formal system with similar rules. *Please* contact me if you spot an error; a major reason I set out to write this was to confirm to myself that *I* properly understand these concepts.

1 Motivation

1.1 Typesafe Lists

Consider the function `hd : 'a list -> 'a` that returns the first element of a list. Typically, this function is only defined on non-empty lists, raising some form of error when given an empty input. One way around this is to instead use a type such as `hd_opt : 'a list -> 'a option`, which forces the programmer to handle the empty case explicitly. However, this is no better than pattern matching on the list originally, and indeed only serves to delay the boilerplate in cases in which the list is provably non-empty. For example, consider the following (contrived) example:

```
let v : int list = map foo [1,2,3,4] in
match hd_option v with
| Some x -> do_something x
| None -> assert false
```

It is a feature of ML-inspired languages that, in cases like this, we generally have no choice *but* to pattern match and think about what to do when `hd_option v` is `None`. In this case, however, we *know* that the value `v` is non-empty – `map f` is a function returning a list of the same length as its input, so using this function on [1,2,3,4] gives a list of length exactly 4. However, this fact isn't actually recorded in the type of map anywhere – it is *opaque* to the typesystem. Because of this, the typesystem cannot prove that v is non-empty, and so the "safe" thing to do is pattern match and explicitly mark the extraneous case as unreachable. As you might expect, this boilerplate becomes annoying very quickly, and so nearly every standard library for these languages will provide unsafe function wrappers for this pattern (Option.valOf in OCaml, for example). But in using these functions, we remove the safety net provided by the compiler and typesystem, and invite runtime exceptions that such constructs are designed to avoid (see: option types vs pervasive use of null in other languages).

Instead, it would be ideal to *restrict* the type of hd to only act on lists that are inhabited, so the compiler will throw an error if we attempt to blindly take the first element of a potentially empty list. By *lifting* the information about whether a list is empty into the typesystem, we can then allow the typechecker to construct proofs that a given invocation of hd is definitely safe (if we adjust other functions to match, of course).

1.2 Attempt 1: Generalized Algebraic Datatypes

A somewhat commonly-seen "solution" to this issue that you might see uses GADTs to attach type information to the particular constructor. For example:

```
type empty
type nonempty

type ('a, _) list =
  | Nil : ('a, empty) list
  | (::) : 'a * ('a, 'b) list -> ('a, nonempty) list
```

Then, we can note that map does not change the emptiness of its argument by type-annotating it like so:

```
val map : ('a -> 'b) -> ('a, 'e) list -> ('b, 'e) list
```

where, because the output type 'e must be the same as the input type 'e, we know that if the input list is nonempty, the output must also be nonempty.

This works great if we know that the output emptiness is fixed, or is the same as the input list. However, we begin to run into issues where the relationship between the input and output emptiness is not constant or identity. For example, how would we encode the type of append?

```
val append: ('a, 'e1) list -> ('a, 'e2) list -> ('a, ???) list
```

In fact, we *do* know something about the output type in this case – the output of append is empty iff both inputs are. However, ML-style typesystems lack support for *type-level lambdas*, and so we have no way to express this type without employing more trickery.

Even worse, what about functions where the output has *no* relation to the input?

```
val filter: ('a -> bool) -> ('a, 'e) list -> ('a, ???) list
```

Here, we're *completely* up the creek – there's *no* way for us to know whether the filtered list is empty.

One way around this is to use higher-ranked polymorphism and continuation passing style to ‘`capture'' the output. By making the type of`filter‘ look like

```
val filter: ('a -> bool) -> ('a, 'e) list ->
            (forall b . ('a, b) list -> 'c) ->
            'c
```

we can force consumers of this function to provide a handler that is able to handle both cases. The extra function argument at the end is the “continuation” that consumes the list output by filter.

This is stylistically jarring, however, and ML-style languages generally don't support or require jumping through some hoops to encode rank N types.

1.3 Attempt 2: Sum Types

The fundamental problem with `append` and `filter` in the previous example is that, in different cases, these functions must return *different types*. This is generally forbidden[1], and so we're stuck.

On the other hand, we *have* a mechanism by which we can join two disparate types – sum types! A generic list is *either* an empty list, or an inhabited list. So we come up with

```
type 'a emptylist = nil
 and 'a inhablist = (::) of 'a * 'a list
 and 'a list = E of 'a emptylist
             | N of 'a inhablist
```

and so we can express functions like `filter` by returning an `'a list`, whereas functions like `hd` must take in `'a inhablist` as input.

This is a lot of code for a relatively simple idea, though! In addition to the mutually recursive types, here's an example of *using* this type:

```
let rec map f L =
  match L with
  | E nil -> E nil
  | N (x :: xs) -> N (f x :: map f xs)
```

[1]There are exceptions stemming from type shenanigans (oftentimes via GADTs!), but this isn't one of them.

Notice the extra uses of `E` and `N`. This adds clutter without really adding meaning, and gets increasingly cumbersome as we write more complex functions. In addition, you may have noticed that this *doesn't even solve the original problem!* When we call `map f [1,2,3,4]`, we get back a value of type `list`, which *doesn't tell us that the list is non-empty!*

Really, what we *want* is a way to express that a non-empty list is the same as a regular list, but with some *extra information* (particularly, that the list is non-empty).

2 The Subtyping Relation

Let us introduce a new judgment into our typesystem. Let $\tau_1 <: \tau_2$ (pronounced "extends", as in "τ_1 extends τ_2") be a judgment claiming pretty much exactly what we said above, that a value of type τ_1 is equivalent to a value of type τ_2, with some more specific information attached. This leads to the following typing rule:

$$\frac{\Gamma \vdash e : \tau_1 \quad \tau_1 <: \tau_2}{\Gamma \vdash e : \tau_2}$$

This is known as the *subsumption* rule, where we allow terms to automatically "upcast" from a subtype to a supertype. Using our intuition about what `<:` means, this should make sense – if a τ_1 is a τ_2 with some extra information, then we should be able to recover the τ_2 by discarding that extra specificity.

Another important property to maintain with this relation is its transitivity. In fact, if we are only concerned with typing specific expressions, then this property is irrelevant, as we can already show that $e : \tau_1$, $\tau_1 <: \tau_2$ and $\tau_2 <: \tau_3$ implies $e : \tau_3$.

Exercise: Give a derivation showing this.

When discussing the types *themselves*, however, it is often convenient to add a rule stating this directly:

$$\frac{\tau_1 <: \tau_2 \quad \tau_2 <: \tau_3}{\tau_1 <: \tau_3}$$

To solve our list problem, then, we want `'a nlist` to be a *subtype* of `'a list`, and re-structure all our functions accordingly. Before we address this directly, however, we need to explore how, exactly, this relation should behave.

3 Structural Subtyping

In any particular subtyped calculus, we might consider baking in forms of *semantic* subtyping, where the relationship between two given types is based on the fundamental properties of our primitive types. For example, convention claims that $\mathbb{N}$, the set of natural numbers, is a subset of $\mathbb{R}$, the set of reals. Then, assuming we have analogous types `nat` and `float` in our language (using floating point numbers to approximate the reals), we might decide that `nat <: float`[2].

However, in the absence of such specificity about our basic types, let us instead examine what properties might be derived *structurally*, examining only the form of the types themselves.

3.1 Products

Consider the following two types:

```
type t1 = { l1 : a
          ; l2 : b
          }
```

```
type t2 = { l1 : a
          ; l2 : b
          ; l3 : c
          }
```

(note that we're using *labeled* products, otherwise known as "records" instead of standard tuples, because it makes the semantics much clearer)

In keeping with our intuition that "the subtype is the same as the supertype but with more information", we notice that any value of type `t2` has all the fields of `t1` with the same types, but also has some data `l3`. We thus add a "width-subtyping" rule for records:

$$\frac{}{\{\ell_1 : \tau_1, \dots, \ell_n : \tau_n, \dots\} <: \{\ell_1 : \tau_1, \dots, \ell_n : \tau_n\}}$$

saying that a record ρ_1 is a subtype of another record ρ_2 when ρ_1 contains all the same fields+types as ρ_2.

3.2 Sums

Similarly, suppose we have the following two labeled variants:

```
type t1 = L1 of a
        | L2 of b
```

```
type t2 = L1 of a
        | L2 of b
        | L3 of c
```

Which direction should the subtyping go? We might claim that a value of type `t2` contains "extra" information compared to `t1` by noting that `t2` has an extra variant, so a `t2` is a `t1` but *may* be different. However, this is wrong, as demonstrated by the following snippet:

[2]This may seem alarming – `nat` is certainly not a *subset* of `float` on any real computer architecture! We will gloss over this issue with regards to subsumption for now, as it is largely inconsequential without discussion of syntax forms and dynamic semantics. Attempting to address this leads to "coercive" subtyping, where casts must be made explicit.

```
match v with
| L1 x -> do_something x
| L2 y -> do_something_else y
```

If `t2 <: t1` truly, then it would be valid for `v` to be of the form `L3 z`, as this is certainly of type `t2` and therefore type `t1`. But this would be disastrous, as the match statement does not specify what to do in the `L3` case!

We conclude by noting that the *other* direction is fine –

```
match v with
| L1 x -> do_something_1 x
| L2 y -> do_something_2 y
| L3 z -> do_something_3 z
```

If `v : t1`, then we know that it is restricted to be an `L1` or `L2` variant, so this match statement can never get stuck. The "extra information" gained from having a `t1` over a `t2`, then, can be viewed as the information that a `t1` is *not* an `L3` variant! So `t1 <: t2`.

Another way to view this is that, in many ways, sum types are "flipped around" compared to product types. Categorically speaking, we claim that sums and products are *dual* to each other (in fact, you'll often find sums referred to as "coproducts", to make this relationship explicit). It makes sense, then, that a "bigger" sum is related to a "smaller" sum in the *opposite* of the way that bigger and smaller *products* relate.

4 Variance

The next topic to consider relates to our type constructors. The exact question we will be addressing is

> When is $\tau_1\ t <: \tau_2\ t$, for a particular type constructor t?

To answer this question, we typically must examine not only the internals of the constructor t, but also the relationship between τ_1 and τ_2. The way that the relationship between τ_1 and τ_2 affects the relationship between $\tau_1\ t$ and $\tau_2\ t$ is known as *variance*.

4.1 Lists

As we began this discussion by talking about lists, it seems only right that we consider first the variance properties of the (idealized) type `'a list`.

The naive answer might be that $\tau_1\ \mathit{list} <: \tau_2\ \mathit{list}$ when $\tau_1 <: \tau_2$. This is actually the correct thing to do – if $\tau_1 <: \tau_2$, then we can view all elements of a $\tau_1\ \mathit{list}$ as a τ_2, thus giving us a $\tau_2\ \mathit{list}$, so we might gain the rule

$$\frac{\tau_1 <: \tau_2}{\tau_1\ list <: \tau_2\ list}$$

In fact, we can generalize this to “containers” in general.

Theorem (Covariance of sources) *If there exists a function of most general type $\forall a.(a\ t \to a)$ that returns a value for at least one input, then $\tau_1\ t <: \tau_2\ t$ implies $\tau_1 <: \tau_2$.*

Proof.

Let f, v, t, τ_1, τ_2 be such that $f : \forall a.(a\ t \to a)$, $v : \tau_1\ t$ and $\tau_1\ t <: \tau_2\ t$.

By subsumption, $v : \tau_2\ t$, so $f\ v : \tau_2$.

Alternatively, $f\ v : \tau_1$.

Because f is universally quantified, it cannot rely on any properties of τ_1 or τ_2. But then, $f\ v$ is arbitrary. In particular, any object of type τ_1 can be written as $f\ v$ for an appropriate v.

But $f\ v$ is the same object regardless of whether v is viewed as a $\tau_1\ t$ or a $\tau_2\ t$, so we must be able to view arbitrary objects of type τ_1 as objects of type τ_2.

□

4.2 Functions

Functions are a strange case, because a function arrow really has *two* type variables, the input and the output. We’ll examine these one at a time.

Consider types $\tau \to \tau_1$ and $\tau \to \tau_2$, where τ is fixed. If τ is a type inhabited with an object σ, then certainly $\lambda f : f\sigma$ is a function of type $\forall a.((\tau \to a) \to a)$, so by the above we know that function arrows should be covariant in their second argument. In fact, by a slightly modified argument, we can show this even when τ is the empty type $\bot$ (hint: suppose that there existed some function of type $(\bot \to \tau_1) \to \rho$ and consider what behavior this function could possibly have).

The first argument, on the other hand, actually *flips* this behavior. If we have a function of type $\tau_1 \to \tau$, we’re eventually going to want to *call* it on a value of type τ_1 (formally we might say that we are allowed to *eliminate* this function by passing it an input). For this to be safe, we must know that any subtype of the type $\tau_1 \to \tau$ must be able to safely handle a τ_1. But this means that subtypes of $\tau_1 \to \tau$ must have *supertypes* of τ_1 on the left (as otherwise they might perform an operation that is only implemented by the subtype). This gives us the rule

$$\frac{\tau_2 <: \tau_1}{\tau_1 \to \tau <: \tau_2 \to \tau}$$

This phenomenon, in which we *reverse* the relationship between τ_1 and τ_2 to $\tau_1\ t$ and $\tau_2\ t$ is known as *contravariance.*

By duality, we might expect to be able to produce a similar theorem to the one given above, perhaps by the existence of a function of type $\forall a.(a \to a\ t)$. Unfortunately, this doesn't quite work – the function $\lambda x.[x]$ has type $\forall a.(a \to a\ list)$, but lists are definitely not contravariant.

Proposition (Contravariance of sinks) *If there exists a type c such that there exists a function of type f such that $f : \forall a.(a \to a\ t \to c)$, then $\tau_1\ t <: \tau_2\ t$ implies $\tau_2 <: \tau_1$.*

This is marked as a proposition, rather than a theorem, because it is false as stated. For example, the function $\lambda\_\_1.\lambda\_\_2.()$ has type $\forall a.\forall b.(a \to b \to unit)$, which of course specializes to $\forall a.(a \to a\ t \to c)$ for *any* t. It may be enough to assert that f "meaningfully uses" its arguments, but this is difficult to formalize. That said, I'm not actually sure that *this* is true either – if you can prove it (even semi-informally in our ML-like world), *please* let me know!

4.3 Mutable References

It turns out that mutable state (important to imperative programming) interacts in an interesting way with subtyping variance. Consider the types $\tau_1\ ref$ and $\tau_2\ ref$, referring to the types of *mutable reference cells.*

Notice that the dereference function `!` has type $\forall a.(a\ ref \to a)$, so by covariance of sources (I promise I will explain the terms "source" and "sink" shortly), we have that $\tau_1\ ref <: \tau_2\ ref$ implies $\tau_1 <: \tau_2$.

However, $\tau_1 <: \tau_2$ is *not* sufficient to conclude that $\tau_1\ ref <: \tau_2\ ref$. Let `A <: B` be types, and consider the following code:

```
let magic (r: A ref) (v: B) : A =
  (r : B ref) <- v;
  !r
```

Remember that the rule of subsumption means that annotating `r` as type `B ref` is a no-op (just a hint to the compiler). But `!r` (= `v`) is definitely of type `A`, as `r : A ref`, even though `v` is already of type `B`! The only way this is safe is if `B <: A`.

In fact, we need *both directions* to assert that $\tau_1\ ref <: \tau_2\ ref$. This can only happen if τ_1 and τ_2 are the same type up to some definition of "same" (reordering of labels, etc)[3]. This property is known as *invariance.*

One way to think of covariance and contravariance is to discuss what you can *do* with an `'a t`. If having an `'a t` possibly *gives* you an `'a`, it is called a *source*, and is therefore covariant by the proof above. Otherwise, if an `'a t` *consumes* an `'a`, then it is called a *sink*, which is contravariant using the same

[3]We should be careful that we don't claim that $\tau_1 <: \tau_2$ and $\tau_2 <: \tau_1$ implies $\tau_1 \cong \tau_2$ (isomorphism) unduly, as isomorphism often works strangely with subsumptive subtyping. For example, $(a, (b, c)) \cong ((a, b), c)$, but in these cases the term $v.0$ is ambiguous. The difference is technical and difficult to reason about without formally defining semantics.

reasoning as with function arguments. However, references are *both* a source (by dereferencing) *and* a sink (by writing a value to the cell). We don't run into this issue with *immutable* containers because growing a container necessarily changes the type of the (resulting) container, but writing a value to a ref cell *doesn't* retroactively change its type.

4.4 Bivariance

Thus far, we've seen examples of both covariant and contravariant type constructors, along with a type constructor that is both (invariant). You may then ask whether there exists a type of variance that is *neither* – that is, if there exists some t such that $\tau_1\ t <: \tau_2\ t$ regardless of the relation between τ_1 and τ_2.

In fact, there is! This is the case when an object of type $\tau\ t$ is entirely unrelated to objects of type τ. This is most often seen with phantom type variables intended to aid typechecking. Such types are known as *bivariant* in the given input.

We can show that this is the *only* possibility for a type to be bivariant by appealing to the source theorem and sink proposition above. Regarding the latter case, if $\tau_1\ t <: \tau_2\ t$ always, then we can sink *any value* into a value of type $\tau\ t$ (for any τ!). This can only possibly be safe if sinking a value is a no-op.

Exercise: Provide a similar argument to the above without the source theorem to show that a bivariant type constructor cannot be a source. (Hint: let $\tau = \bot$)

5 Bounded Quantification

You may notice that, after all that, we *still* haven't solved the problem we initially set out to! Even if we construct a type `'a nlist` such that `'a nlist <: 'a list`, we *still* can't tell the typechecker that `map` only returns an `nlist` if given an `nlist`!

The solution is to adjust how we approach polymorphism. Previously, ML-style languages supported only *unbounded quantification* in their types. Polymorphism was all-or-nothing; either you had a specific type or you must be generic over *all* possible types. This is denoted via the implicit $\forall$ seen in polymorphic ML types. However, now we have interesting things to say about types and how they relate to each other, which in turn enrichens our type language.

In particular, we can now have types of the form $\forall\{t : t <: s\}.\tau$ and $\forall\{t : s <: t\}.\tau$, where we can *upper- or lower-bound* the range of types we are quantifying over.

This allows us to solve our list problem relatively elegantly via a combination of subtyping and GADTs by declaring types *unknown*, *nonempty* $<:$ *unknown* and *empty* $<:$ *unknown*. Then `map` can have type $\forall\{e : e <: unknown\}.\forall a.((a \to b) \to (a, e)list \to (b, e)list)$.

Exercise: Show that you can express the type of `append` under this scheme. You may assume any reasonable extension of the syntax to, say, provide an upper and lower bound at the same time, or to allow multiple bounds to apply at once.

Many modern-day subtyping schemes allow quantification in this way. For example, in Java, there exists a type `List<? extends T>` for $\forall\{t : t <: T\}.t\ list$ and `List<? super T>` for $\forall\{t : T <: t\}.t\ list$. However, adding expressiveness to a type-level language often leads to trouble. In fact, typechecking a system with bounded quantification in both directions is *undecidable*.

References

[1] Pierce, Benjamin C. Types and Programming Languages. The MIT Press, 2002.

[2] Reynolds, John C. "Design of the Programming Language Forsythe." Algol-like Languages, 28 June 1996.

[3] Grigore, Radu. "Java Generics Are Turing Complete." ACM SIGPLAN Notices, vol. 52, no. 1, 1 Jan. 2017, pp. 73–85.